CPU SERVICE CLASSES: A SOFT REAL TIME FRAMEWORK FOR MULTIMEDIA APPLICATIONS

BY

HAO-HUA CHU

B.S., Cornell University, 1994
M.E., Cornell University, 1994

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1999

Urbana, Illinois

CPU SERVICE CLASSES: A SOFT REAL TIME FRAMEWORK FOR MULTIMEDIA APPLICATIONS.

Hao-hua Chu, Ph.D.
Department of Computer Science
University of Illinois at Urbana-Champaign, 1999
Klara Nahrstedt, Advisor

The recent development of inexpensive high speed networking technology enables a computing environment where home applications and appliances can be connected to and run on shared general-purpose computing server(s), replacing traditional computing environments where each application or appliance has its own dedicated special purpose computing hardware. The advantages of this new computing environment over the traditional computing environment are *lower cost*, *better resource utilization*, and *better manageability*.

However, there are many technical challenges that need to be solved in this new computing environment. This thesis is an effort to identify some of the technical challenges and to provide solutions.

One key technical challenge is the level of computing resource guarantee that a shared general purpose server can provide to different devices, which have diverse computing resource requirements and varying performance requirements. This thesis presents the design and implementation of a reservation-based CPU management system, called *dynamic soft real time (DSRT) system*, on such a shared general purpose server. Even though it cannot provide a level of computing resource guarantee that is as good as dedicated special purpose hardware, the DSRT system can provide an acceptable level of guarantee to most *soft real time (SRT)* devices which can tolerate occasional violations of computing resource guarantees.

The DSRT system distinguishes itself by providing new concepts: CPU service classes, multiprocessor partitions, and three-phase execution flow. It also provides probing service, adaptation service, advance reservation, distributed reservation, access control capability, and remote monitor to the client devices. In addition, the DSRT system has a platform-independent middleware design with multiprocessors support.

This thesis further expands on the DSRT system to a multi-resource management system, called *QoS-Aware Resource Management Framework (QualMan)*. QualMan manages various types of shared resources including the CPU, memory, and the network.

To my dear mother

Acknowledgements

I have many people to thank for the completion of this dissertation. First and foremost, I would like to thank my advisor, Professor Klara Nahrstedt, for motivating and guiding my research. I am especially thankful for her rescue efforts on occasions when I came close to failure. I am grateful to her for her willingness to devote time to listen and talk to me irrespective of her extremely busy schedule.

I would like to thank the other members of my thesis committee for their suggestions to improve the quality of this thesis: Professor Vaduvur Bharghavan, Professor Roy Campbell, and Professor Lui Sha.

I would like to thank my colleagues in the MONET group who have cooperated with me on the DSRT system and QualMan: Goutham Garimella, Roland Geisler , Manish Gupta, Kihun Kim, Chih-han Lin, Srinivas Narayan, Jiang Qian, Vignesh Venkataramani, Duangdao Wichadakul, and Dongyan Xu. I would like to thank Lintian Qiao for his friendship, wise advise, and companionship on numerous late nights at the office. I would like to thank other members of the MONET group who have generously shared their ideas with me: Alex Chen, Shigang Chen, Baochun Li, Sergio Servetto, and Mukul Chawla.

I would like to thank my past and present apartment mates: John Chen, John Lee, and Alex Hsu. They made my life at Urbana Champaign one of my most memorable experience. I would like to thank my friends for their support: Ben-Chung Cheng, Alex Chen, Le-chun Wu, Tai-yi Huang, Chi-li Sung, Mae-yen Tsai, Jennifer Chang, Ann Wei, and Mark Gardner. I am especially grateful to Szu-wen Kuo, who has been like a sister to me.

I would like to thank my lovely fiancee, Emily Hung, who has brought great happiness to my life. I would like to thank my family members: Yung-hua, Paul, Yang-hua, father, and grandfather. Last, my deepest gratitude goes to my dear mother for her love, and for making me the person I am.

Table of Contents

Chapter

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

The recent development of inexpensive high speed networking technology enables a new computing environment where home appliances or applications can be connected to and run on shared general-purpose computing server(s), replacing traditional computing environments where each appliance or application has its own dedicated special purpose computing hardware. This new shared computing environment holds several advantages over the traditional computing environment.

- *Higher resource utilization*:
  A general purpose computing hardware can be time shared by several appliances simultaneously. On the other hand, dedicated special purpose computing hardware cannot be shared and stays idle when it is not active.

- *Better manageability*:
  An upgrade on the shared computing server is much easier than upgrades on many different dedicated computing hardware.

- *Lower cost*:
  A shared computing server is cheaper than the sum of the costs of many dedicated computing hardware.

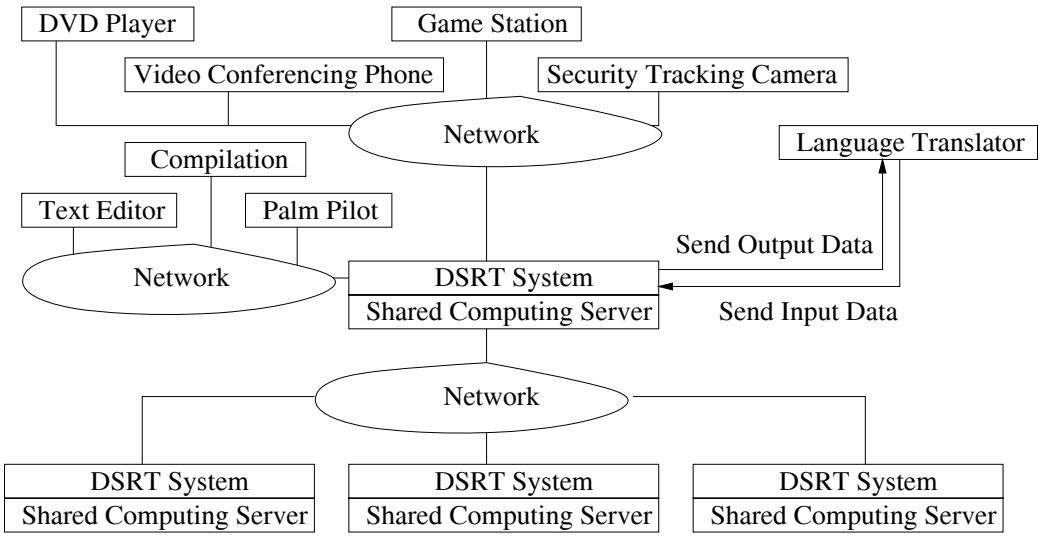


Figure 1.1: A new computing environment with shared computing servers that run DSRT system to support the computing needs of several multimedia appliances running simultaneously.

We show an example of this new computing environment in Figure 1.1. It contains a shared computing server that supports the combined computing needs of several multimedia appliances or applications. One of the appliances can be a light weight language translator which sends a stream of voice data in a foreign language, over a wireless network, to a shared server. The shared server performs the computationally intensive language translation from a foreign language to a native language. The translated voice stream is sent back to the language translator.

If one shared computing server cannot satisfy the computing needs of several appliances, additional computing servers can be added into the network. It is also possible to access existing computing servers on a neighborhood network.

We identify the following issues in this new computing environment:

- Appliances have varying *performance requirements*. Some appliances, such as a language translator and a DVD player, need to provide *real time (RT)* interaction with human beings and they have *RT performance requirements*. A voice translator needs to have its voice data stream translated continuously without interruption and within some known delay bound. A DVD player needs to have its MPEG video/audio data stream uncompressed continuously according to the playback rate. On the other hand, some traditional applications, such as a text editor, a compilation program, or a palm pilot down-loading e-

mails, have only *time sharing(TS)* performance requirements. A shared computing server must be able to support several RT or TS appliances and applications simultaneously.

- Appliances have diverse *computing resource requirements*. A language translator, a DVD player, or a security tracking camera require varying amount of continuous CPU time. Some of their CPU usage patterns may also be constant or variable, or they may be periodic or aperiodic. The shared computing server must be able to support diverse computing resource requirements.

- A shared general purpose computing server cannot provide a level of performance guarantee that is as good as dedicated special purpose hardware, because it uses a general purpose operating system and hardware. As a result, not all RT appliances are suited to run on a shared general purpose computing server. *Soft real time (SRT)* appliances, or applications that can tolerance occasional performance violations, are suited to this new environment. Most multimedia appliances, including those shown in Figure 1.1, are in the SRT category. For example, people will not notice a few dropped frames by a DVD player or on a video conference phone, and they can tolerate a few milliseconds of delay in a language translator. On the other hand, *hard real time (HRT)* appliances, or applications that may produce catastrophic errors given any performance violations, are not suited to run in this new environment. HRT appliances may require dedicated special purpose hardware and software for their guaranteed performance.

This thesis addresses these issues with a reservation-based CPU management system, called the *dynamic soft real time (DSRT) system* for soft real time applications. The DSRT system is designed to run on top of shared general purpose server(s) as shown in Figure 1.1. It provides a management layer on top of the shared servers so that several SRT or TS multimedia appliances or applications with varying degree of performance requirements and computing resource requirements can be supported simultaneously.

1.2 DSRT System

The DSRT system [12] provides the following basic features to SRT clients. We will use the general term *client* to represent different types of multimedia appliances, applications, or pro-

cesses. We consider the following features as *basic* because they are the minimum requirements to support coexistence of both SRT and TS clients in a general purpose operating system environment.

- *Processor time reservation* and *guarantee*:

  The SRT client first enters a reservation with the DSRT system. The DSRT system performs an admission control test to check if there are sufficient processor resources to satisfy the reservation request. Then the DSRT system uses a RT scheduling algorithm to guarantee the reserved processor time to all the SRT clients.

- *Overrun protection*:

  An overrun is a condition when a SRT client needs more processing time than what it has reserved to complete its job[1] at a given iteration. The DSRT system provides protection such that overruns from one SRT client cannot cause violations to the contracts of other SRT clients.

- *Guaranteed TS allocation*:

  The DSRT system also allocates a fixed percentage of processor time to the traditional TS processes so that they are not starved in the presence of SRT clients.

These basic features are also supported by most of the related systems described in detail in Chapter 6. The DSRT system distinguishes itself from the other related systems by introducing the following new concepts and services.

- *CPU Service Classes*:

  Multimedia clients exhibit a wide range of processor usage patterns. For example, some may require constant or variable processor usage time, or they may have fixed or dynamic periods. The DSRT system defines multiple CPU Service Classes that can accommodate SRT clients with a wide variety of processor usage patterns.

- *Multiprocessor Partitions*:

  Multiprocessor partitions are designed to support the processor time guarantees for real time clients using the CPU service classes, and for the traditional time sharing clients.

[1] A client can release a stream of *jobs*, e.g. one job per period.

It contains *RT Partitions* for scheduling the guaranteed parts of reservations, *Overrun Partitions* for scheduling the statistically/non guaranteed parts of reservations, and *TS Partitions* for time sharing clients.

- *Execution Flow:*

  In the DSRT system, a client goes through three different phases. The first phase is the *probing phase*, when reservation parameters are extracted from the client. The second phase is the *reservation phase*, when a client submits its extracted reservation to the DSRT system. The third phase is the *execution phase*, when a client is scheduled by the DSRT server in real time.

- *Adaptation Service*:

  Multimedia clients may change their processor usage patterns over time. For example, the amount of processor time used by a software MPEG decoder may depend on the background scene and the amount of actions in the scene, which can change over time. Another example is object rendering in a game. The amount of processor time may depend on the number of rendered objects on the screen. The DSRT system provides *system-initiated adaptation* which can automatically adjust the parameters in the processor reservation on behalf of a SRT client based on its actual processor usage.

- *Probing Service*:

  It is a challenge for the SRT client developers and the users to determine the most suitable reservation parameters for their SRT clients. Many multimedia applications are written to be platform-independent, and it is intended to be compiled and executed on as many hardware platforms and operating systems as possible. As a result, it is not appropriate to measure and hard-code specific reservation parameters in programs. For example, the processor time to decode a MPEG frame differs significantly between a SUN Sparc 10 and a faster SUN Ultra 2. The DSRT system provides a *smart probing service* to a SRT client. It automatically determines the most suitable service class and parameters so that the SRT process can use it to make a reservation. In addition, the DSRT system provides *profiling service* to record the previously measured reservation parameters for later retrieval.

- *Multiprocessors Support*:

  The DSRT system provides *multiprocessor support* for symmetric multiprocessor (SMP) machines, e.g., the SUN Enterprise servers. The DSRT system can efficiently utilize all processors for real time scheduling and reservation.

- *Middleware Implementation*:

  The DSRT system is designed and implemented as middleware. In this context, the DSRT system is a system software that sits between operating system and applications to provide soft real time support without any modifications to the operating system. This middleware design, shown in Figure 1.2, extends its portability to most of the existing POSIX-compliant UNIX operating systems. We will show in Chapter 4 that we implemented the DSRT system on top of both Solaris and Irix operating systems.

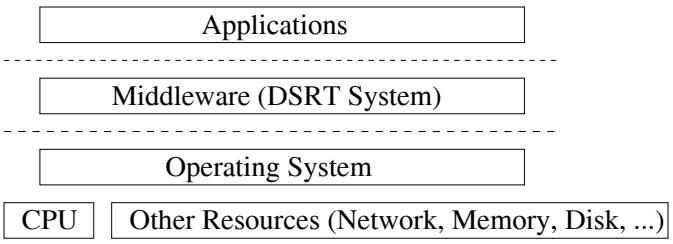


Figure 1.2: Middleware.

1.3 DSRT System Enhancements

We describe several enhancements that are built on top of the DSRT system. These enhancements provide the DSRT system and its clients with the following additional services and capabilities:

- *Advance Reservation:*

  It allows clients to make advance CPU reservations for a specific time interval in the future. This service is useful for applications that make use of expensive instruments which are allocated strictly according to a time table.

- *Distributed Reservation:*

  It allows distributed and collaborating clients to synchronize their CPU reservations (or

other resource types) on multiple hosts across a network. This service is useful for large distributed computing applications that need to use multiple resources across a network.

- *Access Control Capability:*

  It enhances the DSRT system with an access control and accounting mechanism to control the resources end users can acquire through advance or immediate reservations. This solves the problem of unlimited use of reservations by greedy users. The access control service can be incorporated into a *pricing model* for resource providers to charge clients.

- *Remote Monitor:*

  It provides a centralized information database about resource availability on shared servers, and the usage patterns of clients over a distributed environment. The remote monitor can be used in a *distributed resource discovery protocol* to match resource requests with shared servers that have equal resource availability.

1.4 QualMan

We expand the DSRT system into a multi-resource management system called *Quality of Service Aware Resource Management System (QualMan)* [49]. QualMan provides the end-to-end quality of service (QoS) for various distributed multimedia clients that require reservations and RT scheduling on shared resources along their execution path. The considered shared resources are CPU, memory, and network. The DSRT system serves as the CPU resource server in QualMan.

1.5 Contributions

This thesis makes contribution in the development of a novel dynamic soft real time scheduling framework for continuous media applications. In specific, it makes contributions in the following aspects:

- *DSRT System Concepts:*

  The DSRT system introduces three innovative concepts. The *CPU Service Classes* concept allows the specification of different usage patterns by clients. The *Multiprocessor Partitions* concept provides a multi-level scheduling framework that supports guarantees

to the CPU service class reservations. The *Execution Flow* concept divides the client's execution into three phases: reservation parameters are extracted during the probing phase, a reservation is submitted during the reservation phase, and it is executed in real time during the execution phase.

- *DSRT System Implementation:*

  We have implemented and verified the DSRT System concepts through a middleware implementation on top of several operating system platforms, including Solaris, Irix, and Windows NT. We have also made a software release of the DSRT system, which has been widely used among numerous researchers or research groups all over the world because of its middleware design (e.g., Globus Project at Argonne National Lab, Catalase Project at Beckman Institute, GRASP lab at University of Pennsylvania, Electrotechnical Laboratory in Tsukuba Science City, Japan).

- *DSRT System Enhancements:*

  The DSRT system is further enhanced (1) to accommodate a distributed computing environment with *distributed reservation*, and *remote monitor* services, (2) to control unlimited resource reservations by greedy users with *access control* capability, and (3) to allow for *advance reservations*.

- *QualMan Design and Implementation:*

  We have designed and implemented a general multi-resource management platform, called QualMan, for SRT distributed clients that have end-to-end performance requirements. The considered resources are CPU, memory, and network bandwidth. QualMan provides a general and uniform resource specification model and a general resource control model for CPU, memory, and communication resources.

- *Extensive Literature Survey:*

  In order to bring out the various innovative features of the DSRT system, we have conducted an extensive literature survey of related systems and have compared the related systems with the DSRT system.

- *The DSRT System Applications:*

  We have integrated the DSRT system as a resource management component in the Globus and Catalase projects.

We also give a brief summary on the following innovations made by the DSRT system with respect to other soft real time systems. The details are described in Chapter 6.

- The DSRT system provides a *rich set* of QoS specification for the QoS-aware CPU management, expressed through the CPU service classes. It allows the clients to specify their static and *dynamic processor usage patterns* in the form of average/peak processor usage time, and an accumulative burst tolerance in either a fixed or dynamic period. Other related systems allow the clients to specify their QoS only in the form of a static processor usage specification, e.g., fixed amount processor usage time in a fixed period.

- The DSRT system provides guarantees within a bounded system jitter in an *overloaded situation* if the applications conform to their reservations. Other soft real time systems do not provide guarantees in overloaded situation; instead they degrade the application performance.

- The DSRT system provides an *enhanced measurement-based probing service* (smart probing) for an automatic extraction of the QoS-related reservation specification from a client process. Other related systems suggest other methods to extract usage patterns from client processes, e.g., simulation, instruction counting, or benchmarking.

- The DSRT system provides *hierarchical multiprocessor partitions* to schedule client processes with the CPU service class specification. The goal of the hierarchical multiprocessor partitions is to isolate clients' reserved/guaranteed usages from their best-effort/non-guaranteed usages. The best effort/non-guaranteed usages occur when applications violate their QoS contracts, i.e., exceeding their reserved CPU time.

- The DSRT system divide the processor bandwidth according to *reserved runs and overruns* from applications. Other hierarchical scheduling systems divide the processor bandwidth according to applications' classes or different scheduling algorithms.

- The DSRT system is implemented at the *user-level* on top of a POSIX-compliant UNIX system; whereas other related systems are implemented at the kernel-level. The user-level implementation enhances its portability to other POSIX compliant UNIX systems and its deploy-ability because it can run as a daemon process.

1.6 Organization

We organize the remainder of the thesis as follows. In chapter 2, we explain the three innovative concepts of the DSRT system: CPU service classes, multiprocessor partitions, and execution flow.

In Chapter 3, we describe DSRT system components which include smart probing, admission control, conformance test, adaptation, and partition scheduling.

In Chapter 4, we present the middleware (user-level) implementation of SRT system on top of several operating system platforms including Solaris 2.6, Irix 6.5, and Windows NT 4.0 operating systems [2]. We describe the application program interfaces (APIs) provided by the DSRT system to SRT clients. We also discuss limitations of the DSRT system.

In Chapter 5, we run several experiments with different mixtures of SRT clients, and evaluate the performance results on the DSRT system.

In Chapter 6, we discuss work related to the DSRT system. We divide the related work into two parts: (1) systems that provide RT support for soft/hard RT clients in the general purpose operating system environment, and (2) theoretical frameworks that provide statistical guarantees to SRT clients with variable execution time and dynamic processor usage patterns. We compare the DSRT system to these related systems and frameworks, and carefully state their differences.

In Chapter 7, we present QualMan, QoS-Aware Multi-Resource Management framework, which is an extension of the DSRT system. QualMan manages various types of shared resources including CPU, memory, and network. It employs the DSRT system as its CPU resource manager.

[2] We have implemented the dynamic SRT system on top of Solaris 2.6 and Irix 6.5 systems. However, we have only implemented the static SRT system, a previous version of the DSRT system, on Windows NT 4.0 operating system.

In Chapter 8, we describe the DSRT system enhancement services, including advance reservations, distributed reservations, security and access control, and remote monitor.

In Chapter 9, we present other applications of the DSRT system, including the Globus project and the Catalase project.

In Chapter 10, we summarize this thesis, state its contributions, and suggest some possible future work.

Chapter 2

DSRT System Concepts

The DSRT system provides three fundamental concepts: (1) *Three-Phase Execution Flow* of SRT clients, (2) *CPU Service Classes* for the SRT clients to specify their processor usage patterns, and (3) mapping between CPU service classes and the *Multiprocessor Partition*. We explain each of three concepts in this section.

2.1 Execution Flow

In the DSRT system, a SRT client goes through three different phases as shown in Figure 2.1. The first phase is the *probing phase* in which the DSRT system runs a few iterations of the SRT client without any reservation. At the same time, the DSRT system monitors and records the processor usage of the SRT client iteration by iteration. At the end of the probing phase, the DSRT system analyzes the processor usage history and configures a *reservation* with the most suitable CPU service class and parameters. Then, the DSRT system returns the reservation to the SRT client.

The second phase is the *reservation phase* in which the SRT client submits the probed reservation to the DSRT system. The DSRT system performs an *admission control test* to check if there is sufficient processor resource to satisfy this reservation. If the reservation is accepted, it becomes a *contract*. In a multiprocessor system, the admission control test also determines the *processor binding* of the SRT client, meaning which processor has the capacity to schedule this SRT client.

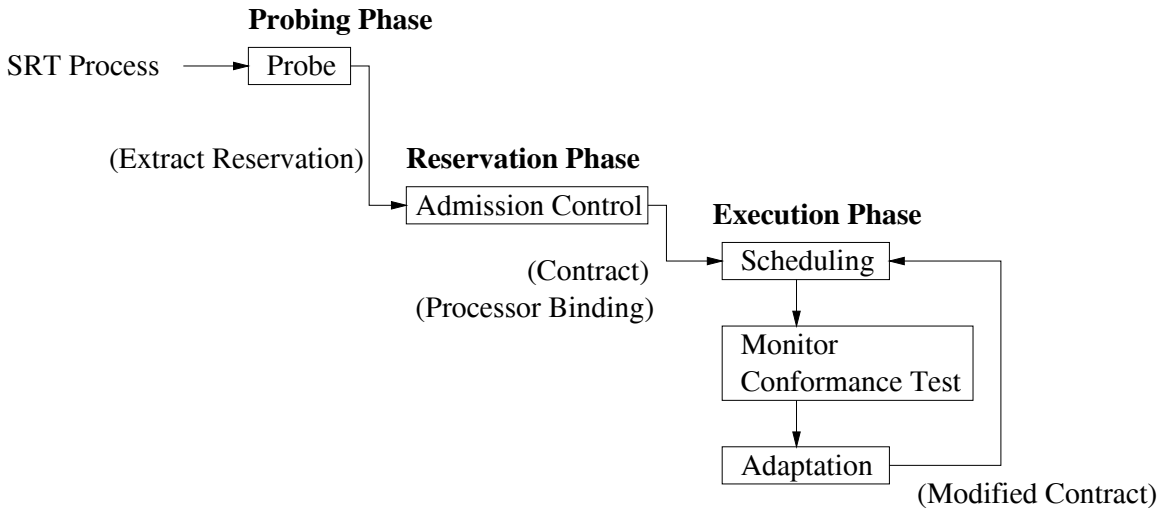


Figure 2.1: Execution flow of a SRT client.

The third phase is the *execution phase* in which the SRT client enters its real time execution. The DSRT system schedules the SRT client and performs a regular *conformance test* to check if the actual processor usage of the SRT client conforms to its contract. If there is a mismatch between the usage and contract (over-reserved or under-reserved), the system can perform *adaptation* to adjust the parameters in the contract on behalf of the SRT client, in order to close the mismatch.

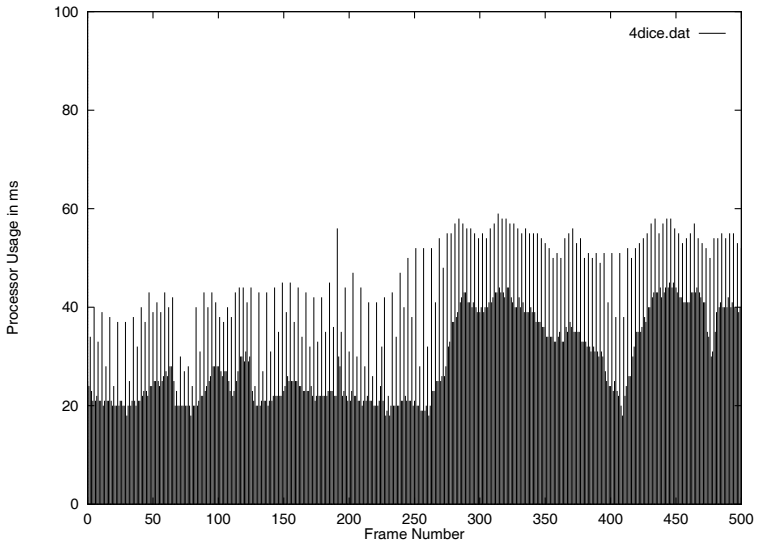


Figure 2.2: Frame-by-frame processor time for a MPEG video decoder decoding a 352x240 stream.

Figure 2.2 shows a processor usage pattern of a MPEG decoder program that requires adaptation. In frames (0-270), the sustainable processing time is around $28ms$. However, there is a major scene change at frame 270, where the sustainable processing time per frame is increased from $28ms$ to $48ms$. As a result, the contract needs to be adjusted accordingly to match the change in the processor usage time pattern. Otherwise, the MPEG decoder will have overruns. Since the overruns are not guaranteed, they can cause missed deadlines and dropped frames.

2.2 CPU Service Classes

Service Class	Specification Parameters	Soft Guarantees
PCPT (Periodic Constant Processing Time)	P = Period PPT = Peak Processing Time	PPT
PVPT (Periodic Variable Processing Time)	P = Period SPT = Sustainable Processing Time PPT = Peak Processing Time BT = Burst Tolerance	SPT $BT^{Statistical\ Guarantees}$
ACPU (Aperiodic Constant Processing Utilization)	PPU = Peak Processing Utilization	PPU
Event	P = Period PPT = Peak Processing Time	PPT

Table 2.1: CPU service classes.

The DSRT system delivers soft processor time guarantees to a set of CPU service classes shown in Table 2.1. The CPU service classes specification is used by SRT clients to describe how they plan to use the processors in the DSRT system so that the system can allocate and schedule processor time for SRT clients during their execution.

The PCPT (Periodic Constant Processing Time) class specification of period(P) and peak processing time(PPT) can be applied to a SRT client process that needs at most PPT amount of processor time every period P. A PCPT client is guaranteed PPT processor usage every period P. Any processor usage exceeding PPT in a period is considered an *overrun* and is not guaranteed. Figure 2.3 shows a processor usage pattern that exemplifies the PCPT class. It is generated by a sampling program that performs a fixed amount of computation every $50ms$. The processing time per iteration stays approximately constant with the peak processing

time of $10ms$. We can still observe some small variations in the processing time. This is the result of using the general purpose operating system and hardware that do not provide any timing guarantees. For example, the unpredictable performance in the memory subsystem, I/O interrupts, and the unbounded service time for system calls in the kernel all contribute to these variations.

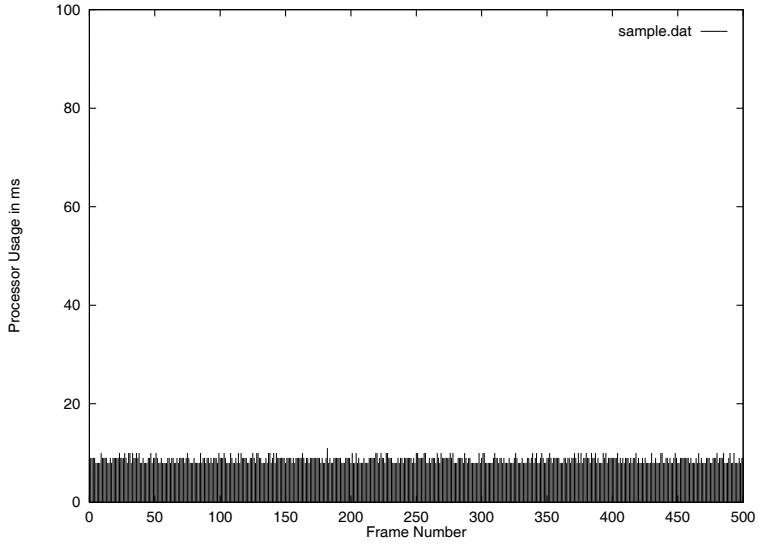


Figure 2.3: Iteration-by-iteration processor usage history for a sampling program that performs a fixed amount of computation.

The PVPT (Periodic Variable Processing Time) class specification of period (P), sustainable processing time (SPT), peak processing time (PPT), and burst tolerance (BT), can be applied to a SRT client that needs on average SPT amount of processor time but no more than PPT every period P. In addition, a PVPT client may generate a processor usage *burst* in excess of SPT but no more than BT. Any processor usage beyond BT is an overrun. A PVPT client receives a soft guarantee of SPT processor usage every period. However, a PVPT client receives only *statistical guarantees* (defined in section 2.2.2) on its bursts. Figure 2.4 shows a processor usage pattern that exemplifies the PVPT class. It is generated from a software MPEG decoder program that plays a MPEG stream at 10 frames per second. The peak processing time is $21ms$, the sustainable processing time is $15ms$, and the burst tolerance is $7ms$. The variations in processing time are caused by a number of factors including the type of frame (I, P, B), the composition of macro-blocks in a frame, the background scene, and the amount of action in the scene. In other words, the processor usage pattern is content dependent. It is undesirable

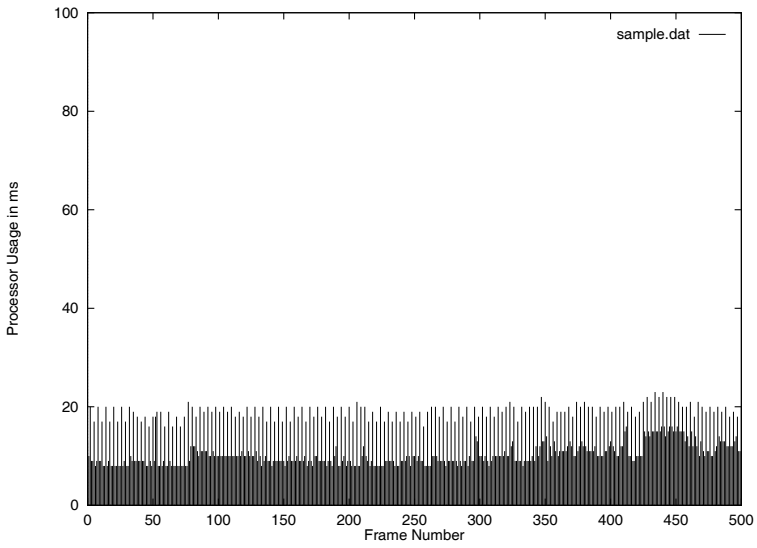


Figure 2.4: Frame-by-frame processor usage history for a MPEG video decoder decoding a 192x144 stream.

to apply the constant processing time class to the MPEG decoder program because it would require the MPEG decoder program to make reservation close to the worst-case level, which is $30ms$ and considerably higher than the average $15ms$. Since the worst-case rarely occurs, it leads to a waste of processor resources and low processor utilization.

The ACPU (Aperiodic Constant Processing Utilization) class specification of peak processing time (PPU) can be applied to a SRT client that does not have a fixed period but that requires PPU percentage of processor time. Given that ACPU class does not have a period specification, an ACPU client needs to set a relative deadline at the start of each iteration during its execution.

The Event class specification of period (P) and peak processing time (PPT) can be applied to a SRT client that needs PPT processor time for only one period P.

We can draw an analogy between our CPU service classes specification in the processor domain and the constant/variable bit rate (rt-CBR/rt-VBR) traffic classes specification in the ATM network domain [1]. It is well known in field of networks that a constant bit rate (rt-CBR) traffic class is insufficient to handle more dynamic network applications like transmitting MPEG streams.

2.2.1 Definition of Soft Guarantees

The DSRT system provides *soft guarantees* on reserved processor time to client processes according to the CPU service class specification defined in Table 2.1. For example, a $PCPT$ client process that passes the admission control test is allocated PPT amount of processor time continuously every period. However, the processor time allocation may be occasionally violated under the following two conditions:

- Some I/O interrupts occur during the time when a client process is dispatched on the processor. These I/O interrupts are not generated for or caused by the dispatched client process.
- Some page faults occur during the time when a client process is dispatched on the processor. These page faults are not generated for or caused by the dispatched client process.

These interrupts and faults occur at the hardware interrupt priority level which the kernel would preempt the dispatched client process. The interrupt/fault service time (the non-preemptable part) is typically small, in the range of us. Since the DSRT system runs on top of the general purpose operating system, it cannot mask the interrupts/faults. As a result, interrupts/faults can violate the processor time allocation to the dispatched client process. They are the causes of *softness* of *guarantees* in the DSRT system. If interrupts/faults service time is denoted as Δ [1] and the pure processor usage time is t, the DSRT system provides soft guarantees (R) in processor time allocation equals to $t + \Delta$.

$$R_i = \begin{cases} PPT + \Delta & \text{if process} \in PCPT, Event \text{ classes} \\ SPT + \Delta & \text{if process} \in PVPT \text{ class} \end{cases}$$

We also distinguish between *resource level* guarantees and *application level* guarantees. The DSRT system provides only resource level guarantees, e.g. continuous processor usage time to a client process. It does *not* provide application level guarantees, e.g. a constant frame rate for a MPEG decoder. The reason is that if a client process needs more processor time than what it has reserved (e.g., an overrun), the DSRT system may not have enough free processor time to

[1] Δ is equal to (PPT or SPT) * $SSBTR * SGP$. $SSBTR$ and SGP are described in section 3.1.1.

schedule the overrun. Only if a client uses processor time within its reservation, its application level performance can be guaranteed.

2.2.2 Definition of Statistical Guarantees

The DSRT system provides *statistical guarantees* to bursts from client processes according to the $PVPT$ class specification defined in Table 2.1. A burst for a $PVPT$ client process is defined as processor usage time in exceed of SPT but no more than BT. The statistical guarantees on bursts (within BT) are different than the soft guarantees on reserved runs (within SPT) defined in the previous section.

Statistical guarantees are quantified by a parameter called SGP (statistical guarantee probability). For example, $SGP = 90\%$ means that the DSRT system will schedule at least 90% of the bursts generated by all client processes before their deadlines. The DSRT system employs a dynamic re-configuration scheme to maintain this 90% statistical guarantee level. This scheme requires dynamic resize on the overrun partitions, which is described in details in section 3.3.2.1.

Assume that the system is in a non-transient period, we quantify the *soft processor time guarantees* for a PVPT process as follows:

$$SPT + (1 - SGP) * BT, \quad 0 \leq SGP \leq 1$$

Because the re-configuration scheme is based on a monitor-and-adapt approach, the DSRT system does not guarantee SGP during the transient period when the system is re-configuring itself under a change in burst load. However, the DSRT system can guarantee SGP after the re-configuration is complete. For example, after the system accepts a new $PVPT$ process, the burst load in the system may increase. As a result, the probability of bursts being scheduled may be less than SGP at the transient period before the system re-configures itself to meet the new burst load. However, after the re-configuration is complete, the system can again provide SGP level of statistical guarantees.

Figure 2.5 plots the percentage of usages being scheduled before their deadlines (% probabilistic processor time guarantees) at different processor usage value. For usages within the reserved run, the DSRT system provides the soft guarantees defined in the previous section. So the probabilistic processor time guarantee is 100%. For usages within burst tolerance, the

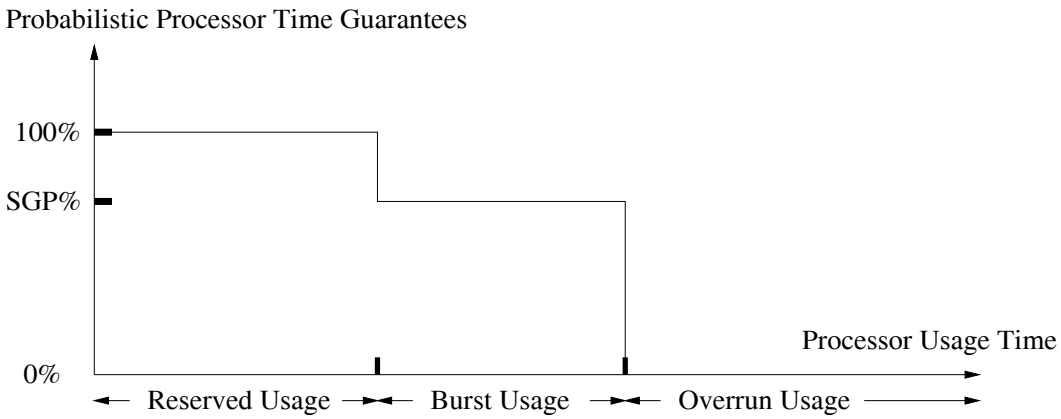


Figure 2.5: Probabilistic processor time guarantees at different processor usage value in an iteration.

DSRT system provides the probabilistic processor time guarantee at the $SGP\%$ level in the non-transient period. For usages exceeding its burst tolerance, the DSRT system does not provide any guarantees. So the probabilistic processor time guarantee is 0%.

2.3 Multiprocessor Partitions

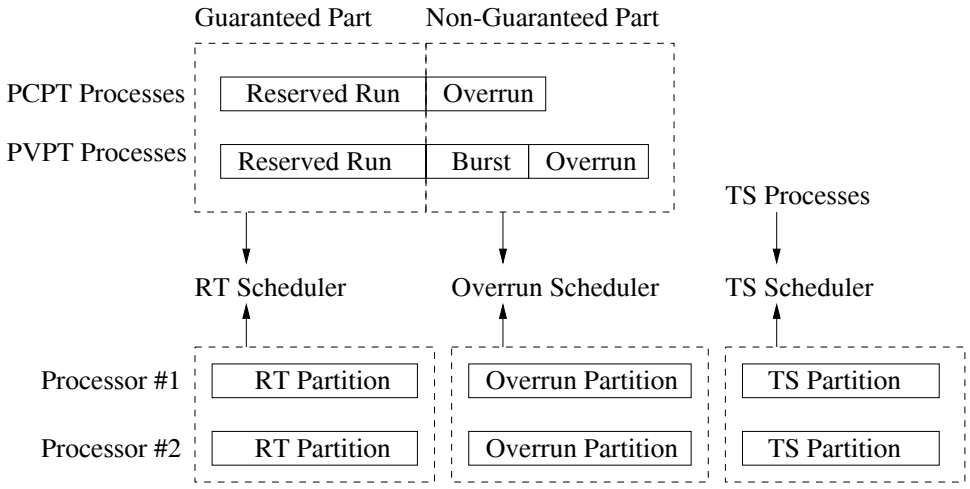


Figure 2.6: Mapping of CPU service classes into processor partitions.

In order to support soft processor time guarantees for our CPU service classes specification, the DSRT system multiplexes the soft real time requirements of CPU service classes into three different processor partitions: RT partition, overrun partition, and TS partition on each processor in a multiprocessor system. As shown in Figure 2.6, the DSRT system maps the guaranteed

and non-guaranteed parts of processor usage in the CPU service classes to different partitions. The guaranteed and the non-guaranteed parts of CPU service classes are defined in Table 2.1.

The RT partitions are dedicated to schedule the reserved runs (guaranteed part) of processor usage from SRT clients, and they are managed by a centralized *RT Scheduler*[2]. The overrun partitions are dedicated to schedule the burst (the statistically guaranteed part of a reservation) and overrun (the non-guaranteed part of a reservation) processor usages from the SRT clients, and they are managed by a centralized *Overrun Scheduler*. The overrun partition acts as a shared resource pool where bursts and overruns from all SRT clients are multiplexed. The TS partition is dedicated to schedule only TS processes so that traditional TS processes do not starve because of SRT clients. The TS partition is managed by the underlying UNIX TS scheduler. The RT and overrun schedulers are described in detail in chapter 3.

The relative sizes of the partitions are initially configurable by the system administrator. The sizes of the RT and TS partitions depend on the relative weights of the RT and TS workloads, while the size of the overrun partition is adjusted according to the burstiness of SRT clients in the system. The dynamic resizing of the overrun partition is described in detail in section 3.3.2.1. It is necessary to provide statistical guarantees to the bursts in the SRT clients.

We denote the sum of all RT partitions in the N multiprocessor system as $RtRatio$. It represents the total processor(s) capacity, counting each processor as 100%. We denote the size of a RT partition on a processor j as $RtPartition_j$, which is a portion of a single processor capacity. The same notation applies to the overrun and TS partitions.

$$RtRatio + OverrunRatio + TsRatio = N * 100\%$$

$$\sum_{j=1..N} RtPartition_j = RtRatio$$

$$\sum_{j=1..N} OverrunPartition_j = OverrunRatio$$

$$\sum_{j=1..N} TsPartition_j = TsRatio$$

[2]We have a centralized dispatcher on top of SMP architecture which schedules the underlying SRT clients on different processors within their RT partitions.

Processor 1	$RtPartition_1 = 70\%$	$OverrunPartition_1 = 20\%$	$TsPartition_1 = 10\%$
Processor 2	$RtPartition_2 = 70\%$	$OverrunPartition_2 = 20\%$	$TsPartition_2 = 10\%$

Table 2.2: Sample multiprocessor partitions.

In our current implementation, we distribute $RtPartition$ equally among all the processors in a multiprocessor system. This also applies to the overrun and TS partitions.

$$RtPartition_j = RtRatio/N$$

$$OverrunPartition_j = OverrunRatio/N$$

$$TsPartition_j = TsRatio/N$$

One possible partitioning of a $N = 2$ multiprocessor system is shown in Figure 2.2. It has $RtRatio = 140\%$, $RtParition_1 = RtParition_2 = 70\%$, $OverrunRatio = 40\%$, $Overrun\ Partition_1 = OverrunPartition_2 = 20\%$, $TsRatio = 20\%$, and $TsPartition_1 = TsPartition_2 = 10\%$.

Chapter 3

DSRT System Components

In this chapter, we describe various components in the DSRT system that are used to implement three concepts described in chapter 2. The order of introduction is according to the execution flow order in Figure 2.1. Section 3.1 describes the smart probing service during the probing phase which extracts the reservation parameters from a SRT client. Section 3.2 presents the admission control test during the reservation phase which decides if there are sufficient resources to meet the reservation request. Section 3.3 explains the multiprocessor partition scheduling during the execution phase. Two other services are also active during the execution phase. The conformance test, which checks if the processor usage pattern of a SRT client is within its reservation, is described in section 3.1.1. Section 3.4 describes the adaptation service which automatically adjusts the reservation on behalf of a SRT client according to its on-going processor usage pattern.

3.1 Smart Probing

Given a SRT client, it is non-trivial to determine the most suitable CPU service class and its usage parameters for reservation. This is further complicated by the fact that the processor usage may depend on input data, on the hardware platform, and on the underlying operating systems. For example, the processor usage for a MPEG decoder is dependent on the amount of action in the input stream. In addition, running the decoder on a SUN Ultra 2 machine is faster than on a SUN Sparc 10 machine.

The DSRT system solves this problem by providing a smart probing service. A SRT client enters a *probing phase* in which the DSRT system runs a few iterations of the client process without any reservation. At the same time, the DSRT system monitors the client's processor usage iteration by iteration, and records them. At the end of the probing phase, the DSRT system analyzes the processor usage history and derives a reservation with the most suitable CPU service class and parameters. Note that the client simply decides the number of iterations in the probing phase, and all other hard work of configuring the reservation is done by the DSRT system.

One important assumption in using the smart probing service is that the processor usage pattern collected during the probing phase of a client process will resemble the processor usage pattern during its real time execution phase. This assumption is reasonable because the probing phase of a client process runs the same binary code on the same machine as in its execution phase. If the usage pattern changes over time (as shown in Figure 2.2), the smart probing service is still useful in the sense that it can give the adaptation service a good initial reservation to start adjusting from. We also assume that the higher the number of iterations that the client process specifies during the probing phase, the more accurate the derivation.

This monitor-and-analysis estimation technique used in smart probing is a straight-forward and accurate method to automatically extract processor usage pattern and reservation parameters from a SRT client process. However, this technique requires a SRT client to endure a small duration of unpredictable performance during the probing phase because it is executed without any reservation. This probing phase (e.g., the first 30 seconds of a movie) is relatively short and should be tolerable in comparison to the length of its execution phase (e.g., 1 hour movie).

Other estimation techniques [28] suggest simulation, instruction counting, or benchmarking which may be too complex and often infeasible. For example, instruction counting may require some compilation and runtime support to trace the execution path (e.g., branches) of a program in order to arrive at an accurate estimation. However, this method is not always feasible because the execution path may be dependent on input data which is dynamic, and execution time may depend on memory/cache performance under different system configurations and work loads.

Based on the usage history collected, the smart probing analysis extracts reservation parameters that satisfy the following two constraints.

- The reservation must conform to the processor usage history collected during the probing phase.
- The reservation should use the minimum amount of processor resources.

These two constraints ensure that the derived reservation does not over-estimate or under-estimate its processor requirement which can cause underruns/overruns during its execution phase. The smart probing analysis uses the conformance test to check if its derived reservation satisfies these two constraints. Conformance test is also used during the execution phase (shown in Figure 2.1) to check if the processor usage conforms to its reservation. However, we need the readers to have knowledge about the conformance test in order to explain the reservation configuration. As a result, we describe the conformance test in the following section.

3.1.1 Conformance Test

The conformance test checks if the processor usage of a client process conforms to its reservation (contract) during its execution phase. The conformance test is invoked for a client process under the following conditions:

- A client process finishes executing a time slice.
- A client process completes its iteration.
- A client process misses its deadline.

The conformance test defines a parameter, $SSBTR$ (system-specific burst tolerance ratio), which is a percentage value that quantifies the variations caused by the scheduling overhead, the underlying general purpose operating system, and the hardware. This *system-generated variation* can occur even if the amount of computation for a SRT process is exactly the same on all iterations[1]. The $SSBTR$ parameter is configured by the system administrator. For example, if $SSBTR = 10\%$ and a processor usage time is $100ms$, the variation caused by the underlying system can be within $\pm 10ms$.

[1]The DSRT system does not control every resource in the underlying operating system. The DSRT system is implemented in the user space; therefore events such as context switching, I/O hardware interrupts, and memory allocation/de-allocation can cause non-deterministic variations in the processing time. This fact leads to the provision of soft real time guarantees instead of hard real time guarantees.

3.1.1.1 Conformance Test for PCPT class

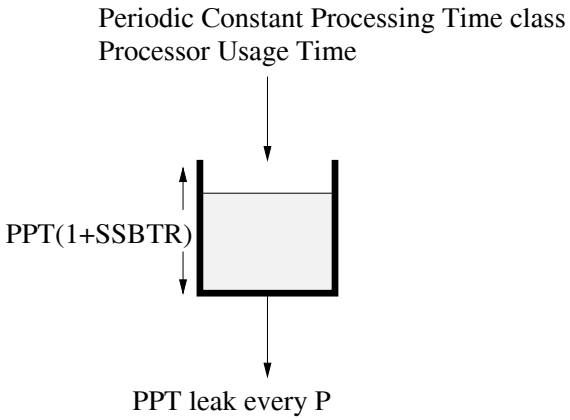


Figure 3.1: Conformance test for the PCPT class.

Conformance test for the PCPT class can be represented by a leaky bucket as shown in Figure 3.1. The bucket has depth $PPT*(1+SSBTR)$. The amount equivalent to the processor usage time is poured into the bucket after each time slice consumption. At the end of each period, the bucket is drained for the amount PPT. A PCPT process conforms to its contract when its bucket does not overflow. It allows a system-generated burst in the amount of $(PPT* SSBTR)$.

Iteration Number	1	2	3
Processor Usage	$53ms$	$56ms$	$47ms$

Table 3.1: Processor usage for 3 consecutive iterations of a PCPT process.

We illustrate the conformance test for a PCPT process using the following example. Assume $SSBTR = 10\%$. A process makes a PCPT class reservation with $PPT = 50ms$. Its processor usage during three consecutive iterations is shown in Table 3.1. The status of its leaky bucket is shown in Figure 3.2 at the end of each iteration. The depth of the bucket is calculated as $50ms * (100\% + 10\%) = 55ms$. In the first iteration, the process consumes $53ms$ of processor time, so $53ms$ is poured into the bucket. At the end of first iteration, the bucket is drained by $50ms$, and $3ms$ of burst is accumulated in the bucket. We assume that the system has enough processor resource to schedule all bursts/overruns before their deadlines in all three iterations.

Note that the leaky bucket in the DSRT system is different from the leaky bucket in the network traffic shaper [13], where it acts as a buffer to store packets during a burst.

In the second iteration, the process consumes $56ms$ of processor time, so $56ms$ is poured into the bucket, causing the bucket to overflow at a height of $59ms$. As a result, the process does not conform at the second iteration. At the end of second iteration, the bucket is drained by $50ms$, and $9ms$ of burst is accumulated in the bucket. In the third iteration, the process consumes $45ms$ of processor time. The bucket reaches a height of $54ms$ and it again conforms.

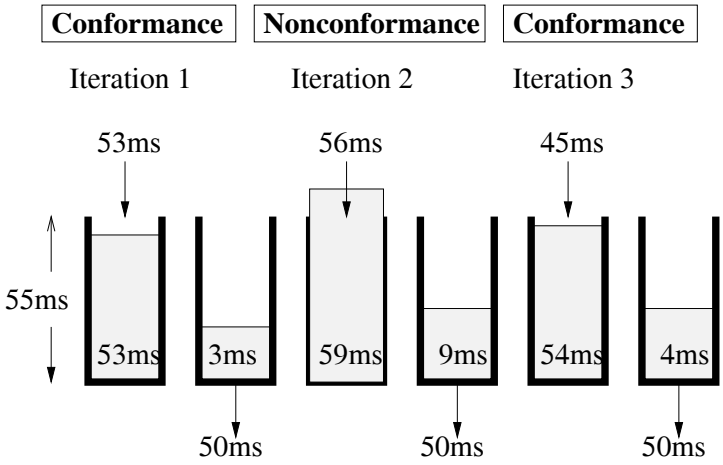


Figure 3.2: The status of leaky buckets for a PCPT process during 3 iterations.

3.1.1.2 Conformance Test for PVPT class

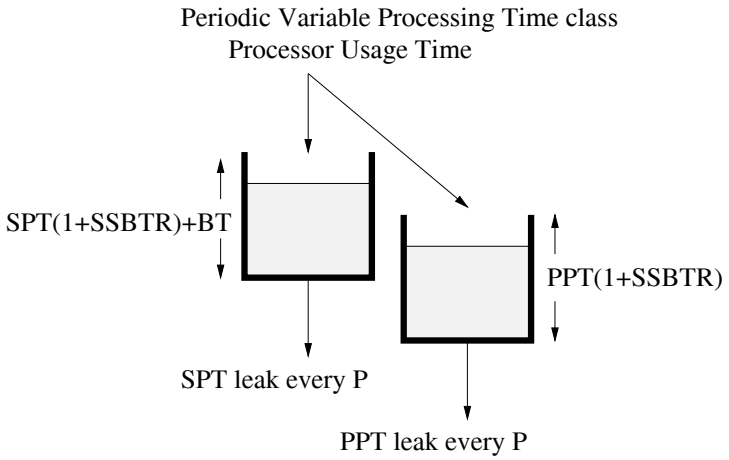


Figure 3.3: Conformance test for the PVPT class.

Conformance test for the PVPT class can be represented by two leaky buckets as shown in Figure 3.3. The left bucket has a depth $SPT * (1 + SSBTR) + BT$, and it is used to check for conformance of the parameters SPT and BT. The right one has depth $PPT * (1 + SSBTR)$, and it is used to check for conformance for the parameter PPT. The amount equivalent to the processor usage time is poured into both buckets simultaneously. During every period P, the left bucket is drained for the amount SPT, and the right one PPT. A PVPT process conforms to its contract when both buckets do not overflow, meaning that it does not violate any of the parameters SPT, BT, and PPT.

We illustrate the conformance test for a PVPT process using the following example. Assume $SSBTR = 10\%$. A process makes a PVPT class reservation with $PPT = 60ms, SPT = 50ms$, and $BT = 5ms$. Its processor usage during three consecutive iterations is shown in Table 3.1. The status of its leaky bucket at the end of each iteration is shown in Figure 3.4. The depth of the left bucket is calculated as $50ms * (100\% + 10\%) + 5ms = 60ms$. The depth of the right bucket is calculated as $60ms * (100\% + 10\%) = 66ms$. In the first iteration, the process consumes $53ms$ of usage time, so $53ms$ is poured into both buckets. At the end of first iteration, the left bucket is drained by $50ms$, and the right bucket is drained by $66ms$. As a result, the left bucket accumulates $3ms$ of burst. Again we assume that the system has enough processor resource to schedule all bursts and overruns before their deadlines in all three iterations.

In the second iteration, the process consumes $56ms$ of usage time, so $56ms$ is poured into two buckets which causes the left(right) bucket to reach a height of $59ms(56ms)$. The left bucket has enough depth to accommodate the additional $6ms$ of burst so the process still conforms. At the end of second iteration, the bucket is drained by $50ms$, and $9ms$ of burst accumulates in the left bucket. In the third iteration, the process consumes $45ms$ of usage time. The left(right) bucket reaches a height of $54ms(45ms)$.

3.1.1.3 Conformance Test for ACPU class

Conformance test for the ACPU class is also represented by a leaky bucket as shown in Figure 3.5. The bucket has depth $PPU * deadline * (1 + SSBTR)$. Since the deadline may be reset by the SRT process at the start of each iteration, the bucket depth changes accordingly. The amount $(PPU * deadline)$ is drained from the bucket after passing each deadline. An ACPU process conforms to its contract when its bucket does not overflow.

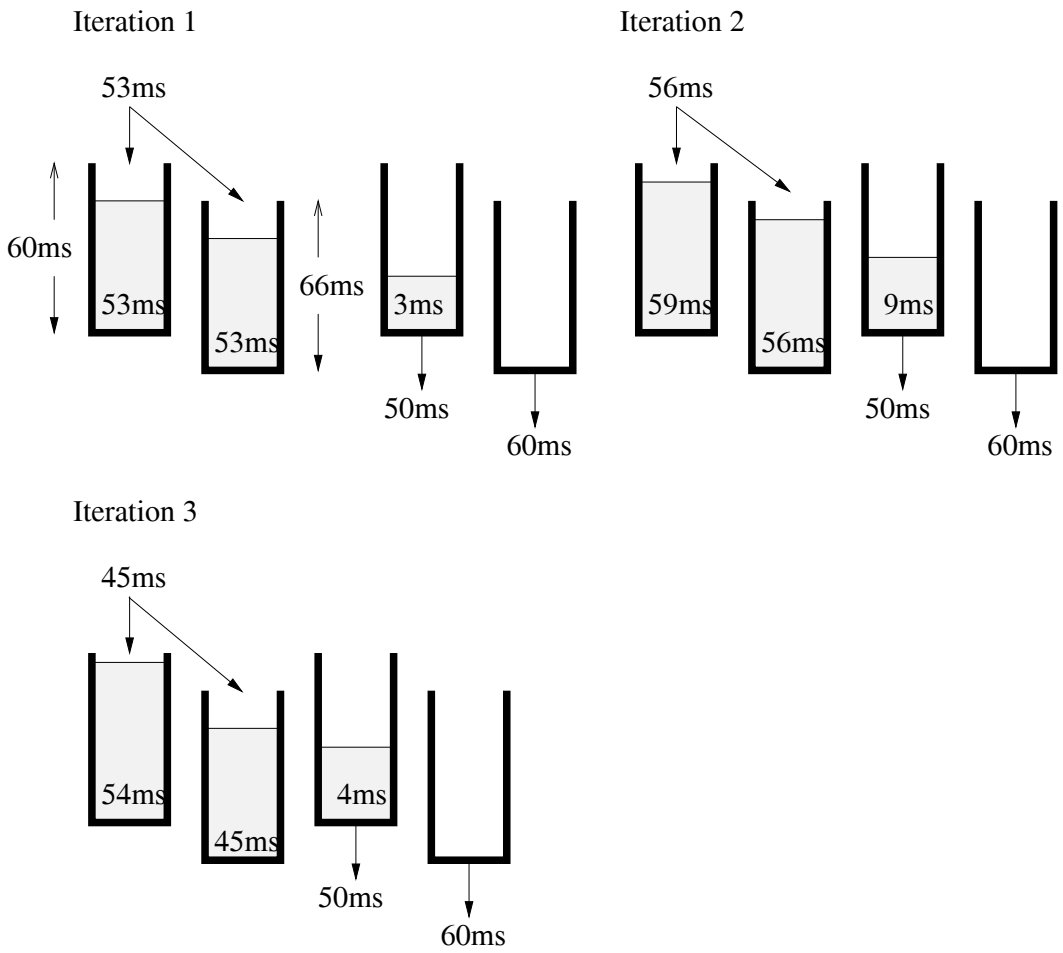


Figure 3.4: The status of leaky buckets for a PVPT process oduring 3 iterations.

We illustrate the conformance test for a PCPT process using the following example. Assume $SSBTR = 10\%$ A process makes a ACPU class reservation with $PPU = 50\%$. Its processor usage and relative deadlines in 3 consecutive iterations are shown in Table 3.2. The status of its leaky bucket at the end of each iteration is shown in Figure 3.2.

In the first iteration, the process sets a relative deadline of $100ms$, and the depth of the bucket is calculated as $50\% * 100ms * (100\% + 10\%) = 55ms$. It consumes $53ms$ of processor time, so $53ms$ is poured into the bucket. At the end of first iteration, the bucket is drained by $(100ms * 50\%) = 50ms$, and $3ms$ of burst is accumulated in the bucket. Again we assume that the system has enough processor resource to schedule all bursts/overruns before their deadlines in all three iterations.

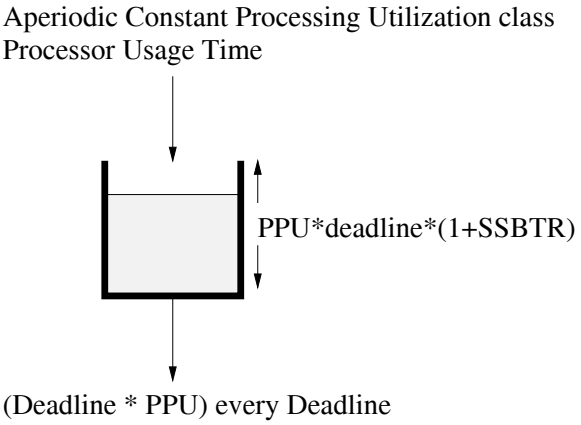


Figure 3.5: Conformance test for the ACPU class.

Iteration Number	1	2	3
Processor Usage	$53ms$	$26ms$	$94ms$
Relative Deadline	$100ms$	$50ms$	$200ms$

Table 3.2: Processor usage for 3 consecutive iterations of a ACPU process.

In the second iteration, the process sets a relative deadline of $50ms$, and the depth of the bucket is calculated as $50\% * 50ms * (100\% + 10\%) = 27.5ms$. It consumes $26ms$ of processor time, and $26ms$ is added into the bucket, causing the bucket to overflow at a height of $29ms$. As a result, the process does not conform at the second iteration. At the end of the second iteration, the bucket is drained by $(50ms * 50\%) = 25ms$, and $4ms$ of burst remains in the bucket. In the third iteration, the process sets a relative deadline of $200ms$, and the depth of the bucket is calculated as $50\% * 200ms * (100\% + 10\%) = 110ms$. It consumes $94ms$ of processor time, so $94ms$ is poured into the bucket. The bucket reaches a height of $98ms$, and it conforms again.

3.1.2 Reservation Configuration

With the understanding of how the conformance test works in section 3.1.1, we are ready to discuss how the smart probing analysis computes a reservation given a processor usage history, while satisfying the conformance and minimum processor resource constraints. For a periodic client process, the followings steps are taken to compute a PCPT or PVPT reservation.

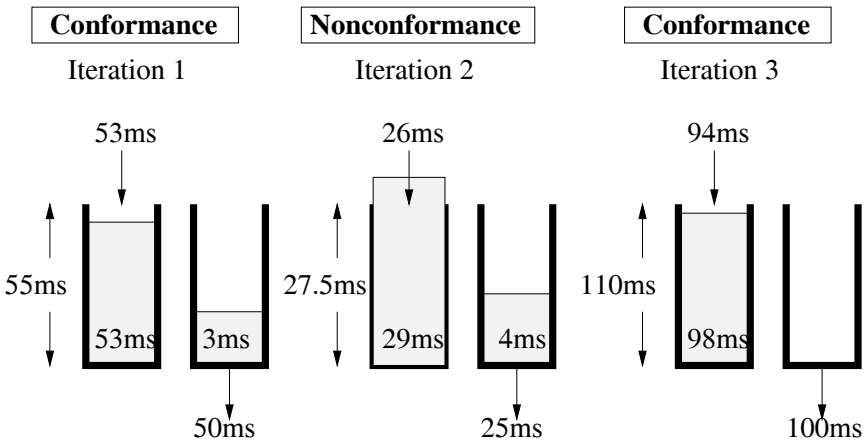


Figure 3.6: The status of leaky buckets for an ACPU process during 3 iterations.

- Compute the *average processor usage* and the *peak processor usage* from the processor usage history.

- Find the maximum height reached in the leaky bucket by running the processor usage, iteration by iteration, through a leaky bucket algorithm at the leaking rate of *average processor usage* every fixed period. The maximum height reached in the leaky bucket immediately after each leakage is the *maximum burst*.

- If the *maximum burst* is less than or equal to the system specific burst tolerance = *(average processor usage * SSBTR)*, the DSRT system configures the reservation as constant processing time (PCPT) class with PPT = *average processor usage*. Otherwise, the DSRT system configures the reservation as variable processing time class (PVPT) with PPT = *peak processor usage*, SPT = *average processor usage*, and BT = *(maximum burst - average processor usage * SSBTR)* .

For example, assume $SSBTR = 10\%$. Table 3.3 shows five iterations of processor usage collected during the probing phase.

Iteration Number	1	2	3	4	5
Processor Usage	$40ms$	$50ms$	$62ms$	$43ms$	$55ms$

Table 3.3: Processor usage history measured during 5 iterations of a periodic client process.

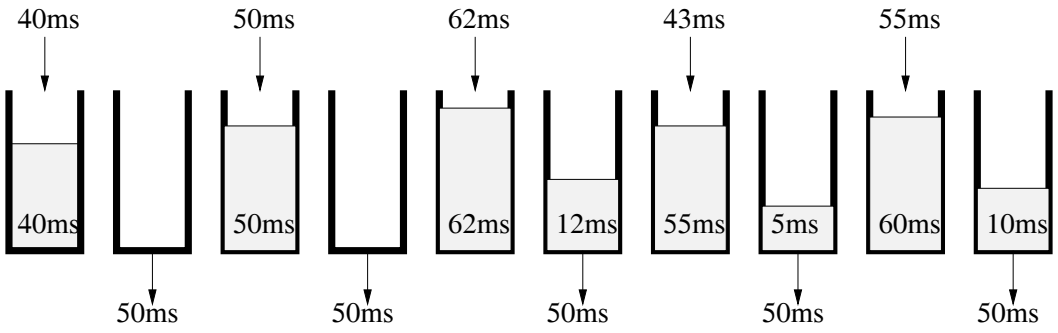


Figure 3.7: The height of leaky buckets in smart probing.

The average processor usage is $50ms$. The peak processor usage is $62ms$. The height of the leaky bucket is shown iteration by iteration in Figure 3.7. The maximum height of $62ms$ in the bucket is reached at the third iteration. The maximum burst is $12ms$ immediately after a leakage of $50ms$. Since the maximum burst is greater than the system specific burst tolerance ($50ms * 10\% = 5ms$), the client is configured as a periodic variable processing time (PVPT) class with $SPT = 50ms$, $PPT = 62ms$, and $BT = 12ms - 5ms = 7ms$.

For an aperiodic client process, we compute the *processor utilization* at each iteration, which is the processor usage time divided by the relative deadline. Then we find the *peak processor utilization (PPU)*. The reservation is configured as an aperiodic constant utilization class with PPU = *peak processor utilization*.

For example, assume $SSBTR = 10\%$. Table 3.4 shows five iterations of processor usage collected during the probing phase. The processor utilization in the first iteration is computed as $25ms/50ms = 50\%$. The maximum processor utilization of 52% occurs in the fourth iteration.

Iteration Number	1	2	3	4	5
Processor Usage	$25ms$	$45ms$	$35ms$	$104ms$	$30ms$
Relative Deadline	$50ms$	$100ms$	$75ms$	$200ms$	$75ms$
Processor Utilization	50%	45%	47%	52%	40%

Table 3.4: Processor usage history measured during 5 iterations of an aperiodic client process.

3.1.3 Profiling

The extracted reservation can also be recorded in a *profile* associated with the client process running on that particular hardware platform. For example, the DSRT system may store a profile

for a MPEG decoder called $mpeg\_decoder.profile$ with the following entry: (platform=Ultra-1, resolution= 352x240, class=PVPT, $SPT = 50ms, PPT = 62ms, BT = 7ms$). The profile can be retrieved for future reservation.

3.2 Admission Control Test

A SRT client enters the reservation phase by submitting the configured reservation to the DSRT system. The DSRT system performs an admission control test (1) to check if there is enough processor resource in the RT Partitions to satisfy the guaranteed part of the reservation, and (2) to determine which processor to allocate this reservation to. The RT scheduler is based on the *preemptive earliest deadline first* [45] (p-EDF) algorithm. We assume that all SRT processes in the system are preempt-able and migrate-able to other processors[2].

Let p_i be a SRT client process, R_i be its processor utilization, N be the number of processors in the multiprocessor system, $RtRatio$ be the sum of all RT Partitions in the system, and $RtPartition_j$ be size of RT Partition on processor j. The admission control test must always satisfy the following two conditions before accepting a new reservation request.

$$(1) \quad \sum_{all\ SRT\ p_i} R_i \leq RtRatio, \quad and$$

$$(2) \quad For\ each\ processor\ j = 1..N, \quad \sum_{p_i\ bound\ to\ j} R_i \leq RtPartition_j, \quad where$$

$$R_i = \begin{cases} PPT/P & \text{if } p_i \in PCPT, Event \text{ classes} \\ SPT/P & \text{if } p_i \in PVPT \text{ class} \\ PPU & \text{if } p_i \in ACPU \text{ class} \end{cases}$$

Condition (1) states that the sum of utilization from all SRT processes must be less than or equal to the sum of the capacity of the RT partitions in the multiprocessor system. Condition (2) states that the sum of utilization from SRT processes bound to each processor must be less or equal to the capacity of RT partition on that processor.

[2]This is generally true for processes running on top of general purpose operating systems and SMP architecture.

The admission control test first checks if condition (1) is true. If not, it rejects the request due to insufficient processor capacity in the multiprocessor system. Otherwise, it tries to locate a processor that has sufficient capacity to accommodate this new request without changing processor bindings of existing SRT processes. Note that condition (1) does not imply condition (2), because of potential resource fragmentations on the RT partitions of processors. For example, processor 1 and processor 2 may each have 10% available capacity in their RT partitions, but neither can accommodate a new reservation request with 20% utilization.

To address this fragmentation problem, the admission control test applies an exhaustive search algorithm, which is equivalent to the NP-complete bin packing problem [14], to try to re-pack existing processes and the new request into the RT partitions of processors. If the exhaustive search also fails, the admission control test rejects the request. In the current implementation, the computational cost of such exhaustive search is small given that it is executed only when new requests arrive (not periodically) and the maximum number of processors in today's top of line SMP machines is small, e.g., the Sun Ultra 450 can support up to 4 UltraSPARC-II processors, and the SGI 540 workstation can also support up to 4 Intel Pentium II Xeon processors [3]. Re-packing existing processes requires the SRT system to modify their processor bindings. We have measured this overhead (a `processor_bind()` system call on Solaris or a `sysmp(MP_MUSTRUN_PID)` system call on Irix) and found it to be small.

After the reservation request is accepted by the admission control test, it becomes a *contract*. The reservation contract is effective immediately until the client process frees the reservation.

3.3 Partition Scheduling

The DSRT system employs two levels of schedulers. The *top-level* scheduler allocates the processor time to the bottom-level schedulers which are the RT, overrun, and TS partition schedulers. The processor time allocated to each partition scheduler is proportional to the relative sizes of its partitions on each processor.

The top-level scheduler uses a very simple credit/debit scheme, similar to the RT Mach [42], to schedule three partition schedulers. Let $(C_{RT_j}, C_{Overrun_j}, C_{TS_j})$ be the credits of the

[3] If the number of processors becomes large in future SMP machines, we will apply a polynomial time approximation algorithm to the bin packing problem.

RT, overrun, and TS partitions on processor j. Let T be the length of time slice (ms) used by the top-level scheduler to schedule a partition at one time. When a partition has accumulated a credit greater than T units, the top-level scheduler will invoke its partition scheduler to run for one time slice T on its corresponding processor and its credit subtracted by T. If there are no processes runnable in that partition scheduler, its credit is still subtracted by T and its processor time is transfered to other partition schedulers that have runnable processes. This places an upper bound on the amount of credits that can accumulate on any partition scheduler. It is possible that none of the partitions may have accumulated a credit greater than T units. For example, all partitions are initialized with 0 credit at the start of the system. Then the top-level scheduler chooses the RT partition first if $C_{RT} \geq 0$, or it chooses either the overrun or TS partition with a larger credit.

At the end of one time slice, the top-level scheduler debits the scheduled partition for T units and credits each partition for the amount proportional to their relative sizes:

$$C_{RT_j} = C_{RT_j} + T * RtPartition_j$$

$$C_{Overrun_j} = C_{Overrun_j} + T * OverrunPartition_j$$

$$C_{TS_j} = C_{TS_j} + T * TsPartition_j$$

3.3.1 RT Partition

The RT partition schedules only *reserved-runs* from SRT processes. A reserved-run is defined as the amount of guaranteed processing time specified in the contracts (see Table 2.1 for what is guaranteed in each class). The RT scheduler must schedule all SRT processes that have been admitted to the RT partitions so that their reserved-runs of SRT processes are always satisfied.

The RT scheduler maintains two queues for each processor as shown in Figure 3.8: the *run queue* and the *wait queue*. The run queue contains processes that are not overrunning their reserved processing time, and they are sorted according to their deadlines in ascending order. The deadline of a periodic process is the end of its current period. The deadline of an aperiodic process is specified by the client at the beginning of its iteration. Processes which have completed their jobs in the current period but have not yet been released for the next

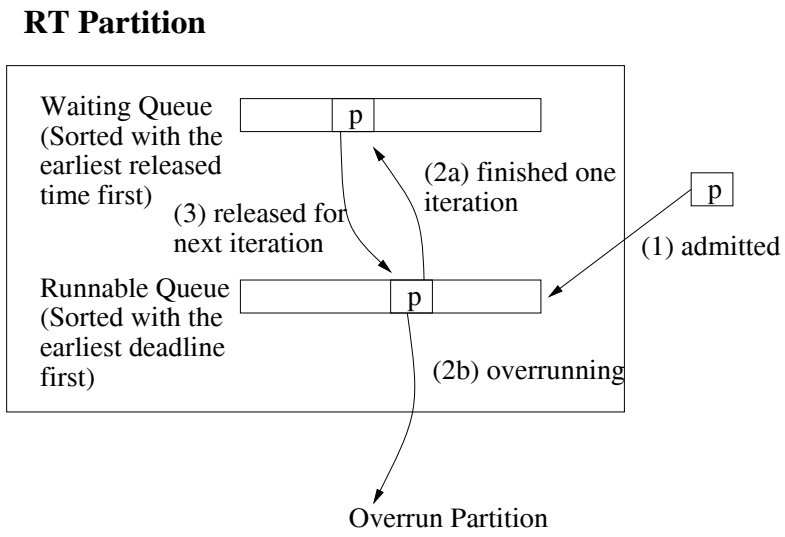


Figure 3.8: Process transition among the queues in the RT partition.

period are placed in the *wait queue*, which is sorted on ascending order according to released time.

When the RT scheduler is invoked by the top-level scheduler, it checks the wait queue to see if there are processes that need to be released. If so, the RT scheduler moves them to the run queue. Then the RT scheduler dispatches the first process from the run queue for one time slice. At the end of one time slice, the RT scheduler can perform overrun detection and protection. If the dispatched process hasn't completed its job and it has used up all its reserved processing time, an overrun is detected. The dispatched process is removed from the run queue and inserted into one of the queues in the overrun partition. Otherwise, the dispatched process is re-inserted into the run queue. This is a straight-forward implementation of the preemptive EDF algorithm. The preemptive EDF algorithm guarantees that the reserved runs - rounding down to the closest increments of T - of all SRT processes that are accepted by the admission control test are always satisfied.

3.3.2 Overrun Partition

The overrun partition schedules *bursts* and *overruns* from SRT processes. The overrun partition acts as a shared backup resource which bursts and overruns from all SRT processes are multiplexed. The overrun partition is managed by the overrun scheduler. The overrun scheduler distinguishes between *bursts* and *overruns*. Bursts are defined as processor usages that exceed

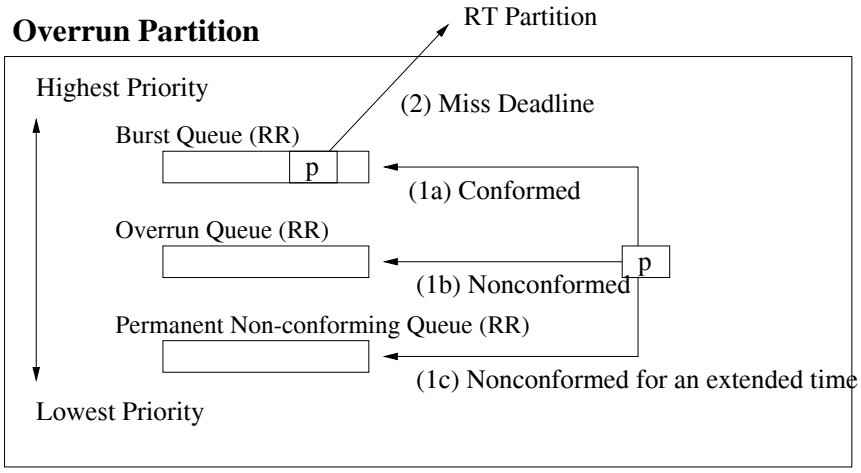


Figure 3.9: Process transition among the queues in the overrun partition.

the guaranteed processor time specified in its contract, but they stay within the burst tolerance of its contract according to the conformance test described in section 3.1.1. For example, a PVPT process can generate a usage burst that exceeds SPT but still within BT in an iteration. Unlike the RT scheduler, the overrun scheduler makes only statistical guarantees on the bursts.

Overruns are defined as processor usages that do not conform to their contracts. The overrun scheduler makes no guarantee on the amount of processor time that each overrun process receives, or that each overrun will be scheduled in time to meet its deadline.

3.3.2.1 Dynamic Resize of Overrun Partition

In order to provide statistical guarantees to the bursts, the DSRT system needs to adjust the size of the overrun partition to accommodate varying degree of burstiness from the SRT clients. It is based on the monitor-and-adjust approach. It actively monitors the percentage of any bursts being scheduled before their deadlines and within the current overrun partition. Then it compares the monitored percentage with a statistical guaranteed level set by the system administrator. If bursts are not being scheduled as often as the statistical guaranteed level, the size of the overrun partition will be increased to accommodate more bursts.

The statistical guaranteed level is quantified by the parameter, SGP (statistical guaranteed probability), which defines the ideal probability that any bursts will be scheduled by overrun partitions to meet their deadlines in the system. The system administrator sets SGP which has

a value between 0 and 1. The system administrator also sets the time window size parameter, ws, which represents how often (in number of time slices) the DSRT system should check if resizing is needed in overrun partitions.

Two counters are used to perform monitoring: one counter (numBursts) counts the number of bursts from all SRT clients in the current time window, and the other counter (numScheduledBursts) counts the number of bursts that meet their deadlines in the current time window. The monitored percentage of bursts being scheduled before their deadlines is simply,

$$\overline{SGP} = \frac{numScheduledBursts}{numBursts}$$

When bursts are scheduled below the statistical guaranteed level $SGP < \overline{SGP}$, the overrun partition size is incremented by 10%. However, there is one constraint prior to resizing. Given that total processor resource is constant in a system, increase(decrease) in an overrun partition requires an equivalent amount of decrease(increase) in a RT Partition. The DSRT system increases an overrun partition only when it is possible to decrease a RT partition by an equivalent amount without violating the existing contracts of p_i on a selected processor j. That is the following equation must be satisfied,

$$\sum_{p_i \; bound \; to \; processor \; j} R_i \leq RtPartition_j - 10\%$$

When the above equation cannot be satisfied, the DSRT system attempts to make room for the increase in overrun partition by (1) moving an existing process to other processors on a multi-processor SMP machine, (2) moving an existing process to other machines by a distributed resource discovery protocol.

3.3.2.2 Overrun Scheduler

The overrun scheduler maintains three queues for each processor in the system as shown in Figure 3.9: *burst queue* for bursts, *overrun queue* for overruns, and *permanent non-conforming queue* for overruns that occur frequently over a long period of time. Permanent non-conformance is defined by two parameters in the DSRT system: *tolerance frequency* and *tolerance window size*. A process is permanently non-conforming when it has accumulated more than *(tolerance*

*frequency * tolerance window size)* the number of overruns in the past *tolerance window size* number of iterations.

When the overrun scheduler is invoked by the top-level scheduler, it first schedules bursts. When there are no bursts, it schedules overruns and then the permanent non-conforming overruns. The overrun scheduler is based on the *multi-level priority round-robin queue algorithm.* Within each queues, the overrun scheduler uses a *round robin* algorithm which gives each process an equal share of the overrun partition. The emphasis is on *fairness.* All queues are RR (round robin). The overrun scheduler inserts a new overrun at the end of the queue regardless of its deadline and dispatches from the head of the queue.

At the end of every time slice (during the interrupt point), the overrun queues need to be checked for any overrun processes that have missed their deadlines. When this occurs, a flag is set to notify a client process of a missed deadline.

Contrary to the RT scheduler, the overrun scheduler can schedule a process on any idle processor which it is not bound to. Since overruns are not guaranteed by the DSRT system, it is desirable for the overrun scheduler to utilize as much otherwise idle processor time as possible to satisfy the overruns. However, this requires an additional overhead to temporarily bind the overrun process to another processor. This overhead has been found to be small.

3.3.2.3 Probabilistic Processor Time Analysis for Overruns

The DSRT system does not provide any processor time guarantees for overruns. However, we provide the following analysis on the worst-case probability that an overrun of length t^{ov} is scheduled before its deadline. Let $OverrunRatio$ be the sum of all Overrun Partitions in the system (also defined in section 2.3). Let U_i, BT_i and P_i be the utilization, the burst tolerance, and the period of a client process i in the system. Let P be the period of the considered overrunning client process. Let M be the number of client processes in the system. Let T be the size of the time quantum. The worst-case probability that an overrun of length t^{ov} is scheduled in the system is defined as:

$Pr(t^{ov}\ is\ schedulable) =$

$$\begin{cases} 1, & \text{if } max(0, OverrunRatio - \sum \frac{BT_i}{P_i}) * max(0, P - \sum_i U_i * P_i) \geq \lceil \frac{t^{ov}}{T} \rceil * T * M \\ 0, & \text{otherwise} \end{cases}$$

We assume the worst-case when all $PVPT$ processes are generating the maximum possible conforming bursts equal to their BT. After the system schedules all bursts, $(OverrunRatio - \sum \frac{BT_i}{P_i})$, measured the available processor utilization in percentage that is left for scheduling the non-conforming overruns. This utilization is multiplied by $max(0, P - \sum U_i * P_i)$, which measured the worst-case completion time of the reserved run of the considered client process based on the worst-case assumption that all other processes have shorter deadline and they are all in phase. This product computes the minimum amount of credit generated in the overrun partitions before the deadline of the considered client process.

We assume all M client processes in the system are overrunning. The considered overrunning client process is queued at the last position in the overrun queue. Given that the overrun queue is scheduled in a round robin fashion with a time quantum of T, the amount of credit needed in the overrun partition to schedule the considered overrun is $\lceil \frac{t^{ov}}{T} \rceil * T * M$.

Stochastic analytical methods [64, 24] can be used to bound the percentage of overruns that can be scheduled prior to their deadlines. However, these stochastic analytical methods are applicable to only simple *fixed-priority systems*. For dynamic-priority systems, such as the DSRT system, stochastic analysis becomes too complex and simulations are used instead. We describe the details of the stochastic analytical methods in section 6.2.

3.4 Adaptation

The DSRT system provides system-initiated adaptation to automatically adjust the parameters in the contracts for SRT client processes based on their actual processor usage time. What the SRT client needs to do is to simply pick the most suitable *adaptation strategy* to adjust its reservation/contract. The DSRT system provides two different adaptations strategies: exponential average strategy and statistical adaptation. Adaptation strategies are applicable to periodic and aperiodic classes (PCPT, PVPT, ACPT) only. They are not applicable for the event class because an event process runs for only one iteration. Our current adaptation strategies adjust the guaranteed parameter in the contract, which is denoted as $\overline{X}$.

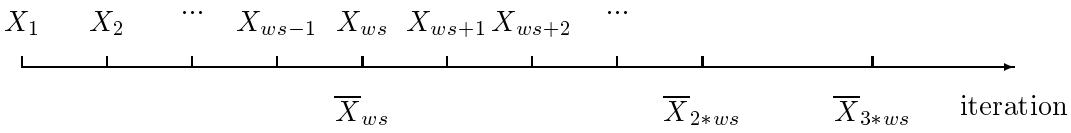


Figure 3.10: Adaptation points.

$$\overline{X} = \begin{cases} PPT, & \text{if process} \in \text{PCPT} \\ PPR, & \text{if process} \in \text{ACPT} \\ SPT, & \text{if process} \in \text{PVPT} \end{cases}$$

All adaptation strategies require a client to specify the *window size(ws)* parameter which represents how often (in number of iterations) the DSRT system should perform adaptation for a client. For example, a PCPT process with *window size = 10* means that adaptation will adjust its PPT parameter every 10 iterations. Note that it is undesirable to perform adaptation too frequently because adaptation involves some computational overhead.

A client can also specify a *range* $(\overline{X}_{max}, \overline{X}_{min})$ for the adjustment of $\overline{X}$, such that $\overline{X}$ must fall within the range $[\overline{X}_{max}, \overline{X}_{min}]$. The default value for $\overline{X}_{max}$ is its period (the maximum possible value), and the default value for $\overline{X}_{min}$ is 0 (the minimum possible value).

We denote $\overline{X}_i$ as the guaranteed parameter in the contract at the i-th iteration. The adaptation points occur at $\overline{X}_{ws}$, $\overline{X}_{2*ws}$, $\overline{X}_{3*ws}$, .. $\overline{X}_{k*ws}$ as shown in Figure 3.10. We denote X_i as the actual processor usage value measured at iteration i, which can be different from the value $\overline{X}$ specified in the contract. Therefore, the first adaptation point $\overline{X}_{ws}$ is computed based on the measured processor usage values $[X_1, .. X_{ws}]$ during the iterations $[1 .. ws]$.

3.4.1 Exponential Average Strategy

When the exponential average strategy is invoked at the first adaptation point, it computes a processor usage value $\widetilde{X}_{ws}$ based on $[X_1, .. X_{ws}]$ during the first ws number of iterations.

$$\widetilde{X}_{ws} = \begin{cases} MAX_{i=1..ws} X_i, & \text{if process} \in \text{PCPT or ACPT} \\ \frac{\sum_{i=1..ws} X_i}{ws}, & \text{if process} \in \text{PVPT} \end{cases}$$

It uses the following exponential average formula to compute the new value $\overline{X}_{k*ws}$ in the contract. This formula is commonly used in many control systems:

$$\overline{X}_{k*ws} = (1-\alpha) * \overline{X}_{(k-1)*ws} + \alpha * \widetilde{X}_{k*ws}$$

The SRT client selects a scaling parameter $0 \leq \alpha \leq 1$ that represents the relative weight between the current measured $\widetilde{X}_{k*ws}$ value and the previous $\overline{X}_{(k-1)*ws}$ value to determine the new $\overline{X}_{k*ws}$ value. The default value for α is 0.5. After the new value is calculated, the contract is adjusted accordingly.

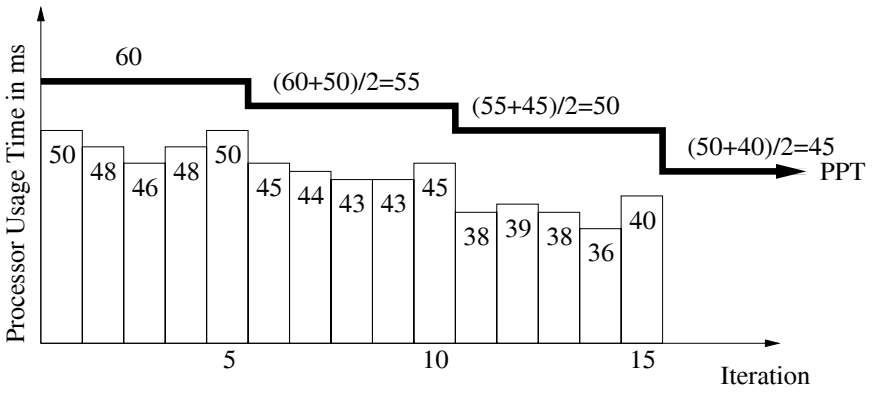


Figure 3.11: Exponential adaptation strategy.

We illustrate with an example in Figure 3.11. A PCPT process enters an initial contract of ($PPT = 60ms$) with the DSRT server. It sets the adaptation strategy with parameters *(α=0.5, ws=5 iterations)*. Figure 3.11 shows its processor usage history measured during 15 consecutive iterations, and the adapted peak processing time (PPT) parameter. Adaptation occurs every 5 iterations. The maximum processor usage time in the first 5 iterations is $50ms$, and PPT is adjusted to $(0.5*60ms+0.5*50ms) = 55ms$. The maximum processor usage time in the next 5 iterations is $45ms$, and PPT is adjusted to $(0.5*55ms+0.5*45ms) = 50ms$. The processor usage time in the last 5 iterations is $40ms$, and PPT is adjusted to $(0.5*50ms+0.5*40ms) = 45ms$.

3.4.2 Statistical Strategy

The statistical strategy requires a SRT client to specify the parameter *overrun frequency(f)*. It will adjust $\overline{X}$ at the adaptation points described in the previous section, to a level where no

more than *(f*ws)* overruns can occur within *ws* iterations. A smaller overrun frequency results in fewer overruns and better quality for clients.

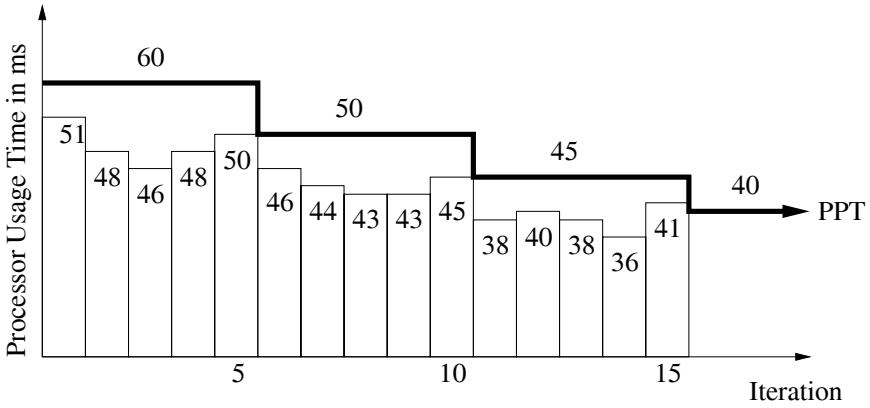


Figure 3.12: Statistical adaptation strategy.

We illustrate with an example in Figure 3.12. A PCPT process enters an initial contract of ($PPT = 60ms$) with the DSRT server. It sets the adaptation strategy with parameters *(f=0.2, ws=5 iterations)*. Figure 3.11 shows its processor usage history measured during 15 consecutive iterations, and the adapted peak processing time (PPT) parameter. Adaptation occurs every 5 iterations, and it adjusts the PPT parameter such that only ($0.2 * 5 = 1$) overrun can occur in the preceding five iterations. In the first five iterations, PPT is adjusted to $50ms$ to allow for at most one overrun ($51ms$ at the 1st iteration) to occur. In the next five iterations, PPT is adjusted to $45ms$ to allow for at most one overrun ($45ms$ at the 6th iteration). In the last five iterations, PPT is adjusted to $40ms$ to allow for at most one overrun ($41ms$ at the 15th iteration).

When the statistical strategy is invoked, it sorts the processor usage [X_1, .. X_{ws}]. Then $\overline{X}_{ws}$ is the $(f * ws + 1)$-th element in the sorted list. The first $(f * ws)$ elements in the sorted list are the allowable overruns.

Chapter 4

DSRT System Implementation

A novel feature of the DSRT system is that it is implemented as middleware and runs entirely in the user space without any modifications to the kernel. This user-level implementation requires the underlying operating system to provide the following three services (1) a real time interval timer, (2) fixed priorities, and (3) process-to-processor binding. These three services are available in one form or another in almost all UNIX systems with the POSIX.4 real time extension.

To show that it is portable to different UNIX platforms, we have implemented the DSRT system on top of SUN Solaris and SGI Irix operating systems. We have also implemented a previous version of the DSRT system, called SSRT (static soft real time system), on top of the Windows NT operating system.

This chapter is outlined as follows. Section 4.1 describes the implementation layout. Section 4.2 explains a user-level process dispatch mechanism. Section 4.3 presents the C++ and Java application program interfaces of the DSRT system. Sections 4.4 and 4.5 describe the Sun Solaris and SGI Irix implementation of the DSRT system. Section 4.6 presents the Windows NT implementation of the SSRT system. Section 4.7 describes how the DSRT system behaves when a SRT client is in a *self-blocking* condition. Section 4.8 discusses the limitations of this user-level implementation.

The implementation layout, user-level priority dispatch mechanism, and APIs are applicable to different operating system platforms.

4.1 Layout

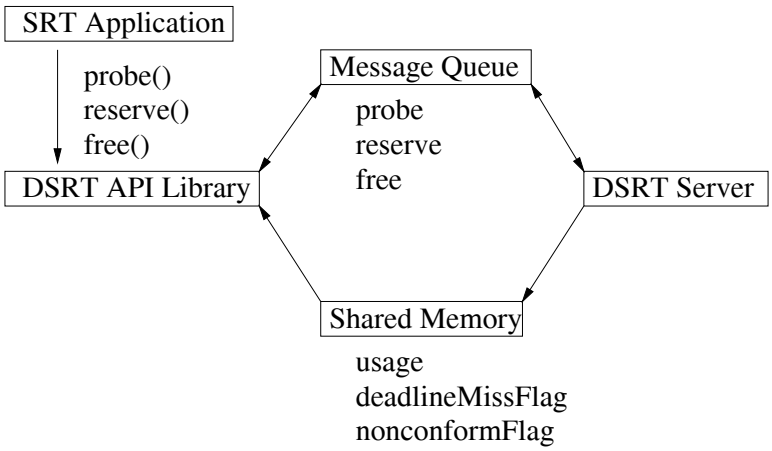


Figure 4.1: The implementation layout of the DSRT system.

The implementation layout of the DSRT system is shown in Figure 4.1. It contains a DSRT server process and a client side library that provides a set of application program interfaces (APIs). The SRT clients access services provided by the DSRT server, such as probing, making a reservation, or freeing a reservation, through the API library. There are two communications channels between the SRT client and the DSRT server: *message queues* which pass service calls and parameters, and *shared memory* which stores various SRT process status (e.g. number of deadline misses, number of non-conformance, usage history, ...). The DSRT server periodically polls the message queues for service calls, performs the requested services (e.g., probing, partition scheduling, adaptation, and etc.), and periodically writes the status of SRT processes into the shared memory.

4.2 Priority Dispatching

The essence in implementing the DSRT server is in the *priority dispatch mechanism*. After the partition scheduler selects a RT process *pid* to be dispatched on a processor *prid* for one time slice, the dispatch mechanism must ensure that process *pid*, if runnable, will be scheduled on processor *prid* for at most one time slice and no more.

The priority dispatch mechanism is shown in Figure 4.2. The DSRT server process must have root privilege so that it can manipulate the fixed RT priorities and can change the priority

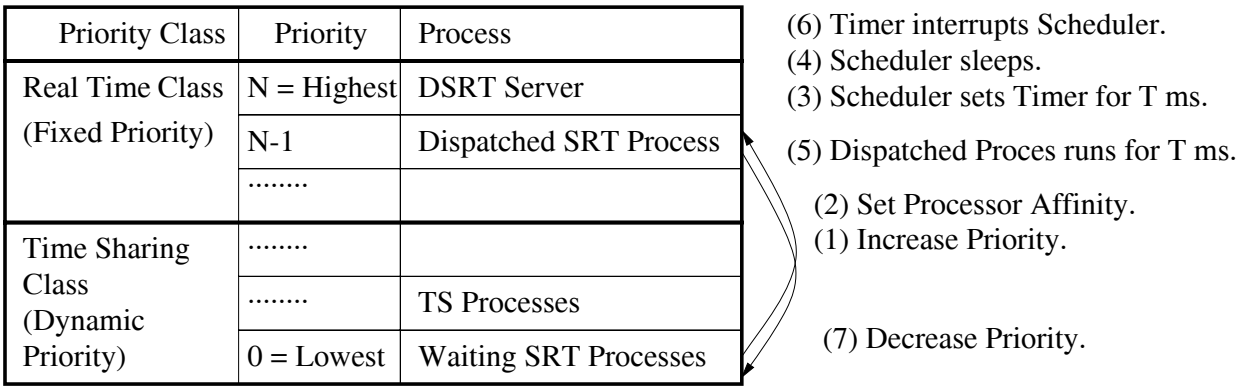


Figure 4.2: The priority dispatch mechanism.

of SRT client processes. We also assume that the two highest priorities in the system (N and $N-1$) are reserved for the exclusive use of the DSRT server.

The server process runs at the highest possible global(and fixed) priority N. At the system startup time, the server sets the real time interval timer equal to the time slice T, which means that the timer will wake up (signal) the server process every T time units.

After a SRT process pid is selected to be dispatched on a processor $prid$, the server promotes $pid's$ priority from the lowest to the 2nd highest global (and fixed) priority $N-1$. If the processor binding is not set already, the server binds process pid to processor $prid$. Then the server sleeps, and pid is scheduled by the underlying kernel given that it is the runnable process with the highest priority. At the end of one time slice, the interval timer wakes up the server, which also preempts pid. The server stops (suspends) pid and lowers pid's priority from $N-1$ to the lowest priority 0. This dispatch cycle repeats.

When we say that the server sleeps after dispatching SRT processes, the server is actually blocking on a semaphore shared with the SRT process. The interval timer will interrupt the semaphore blocking on the server when the timer expires. When the SRT process finishes its computation in its current iteration and prior to the timer interrupt, it unblocks this semaphore which wakes up the server immediately.

Overrun, burst, and underrun detections are implemented as follows. When the server is interrupted, the server calculates the amount of processor usage for the dispatched SRT process. If the dispatched SRT process has finished its computation and it hasn't used up all its reserved processor time, it has a underrun. The server calculates the amount of underrun and allocates

it to other SRT or TS processes. If the dispatched SRT process hasn't finished its computation and it has used up all its reserved processor time (its actual processor usage is greater than its reserved processor usage), it has an overrun or a burst.

The server schedules the TS partitions by not promoting any SRT processes to the dispatch priority. As a result, the kernel will schedule the TS processes according to its TS scheduler.

4.3 Application Programming Interfaces

The DSRT system provide C++ and Java application programming interfaces (APIs). We first illustrate the use of the APIs through a sample program. Then we give the API definitions.

4.3.1 Sample Program

A SRT application calls `reserve()` to make a reservation and to specify the desired service class and parameters. It calls `setAdaptStrategy()` to select an adaptation strategy. Then it calls `start()` to begin the real time execution phase. After it completes its job, it calls `yield()` to mark the end of its job. The `yield()` call is a powerful primitive because it allows the DSRT system to detect and to react to (1) an overrun/burst when the SRT process does not call `yield()` after it has used up the reserved processing time in its contract, (2) an underrun when the SRT process calls `yield()` before it has used up all the reserved processor time in its contract. In the underrun case, the DSRT system reacts by allocating the un-used but reserved processing time to other SRT or TS processes.

The `yield()` call also performs timing control for a SRT process and enforces its periodic or aperiodic behavior. Between each consecutive `yield()s`, the SRT process is scheduled for at least the amount of guaranteed processing time in its contract. It calls `free()` to release the reservation.

```
// Reservation Phase
CpuApi cpu;
cpu.reserve(reservation);
cpu.setAdaptStrategy(strategy);

// Execution Phase
```

```
cpu.start();
for (int i=0; i < numIterations; i++) {
   doJob();
   cpu.yield(...);
}
cpu.free();
```

The probing phase is marked by running `numProbeIterations` iterations between `probe()` and `probeEnd()`. Note that the probing phase should be started prior to the reservation phase described above. At the end of the probing phase, the application calls `probeMatch()` which returns a configured reservation.

```
// Probing Phase
cpu.probe(...);
cpu.start(...);
for (int i=0; i < numProbeIterations; i++) {
   doJob();
   cpu.yield(...);
}
cpu.probeEnd();
cpu.probeMatch(&reservation);
```

4.3.2 C++ APIs

The C++ APIs are encapsulated in a class called `CpuApi`.

- `CpuApi::CpuApi();`

  Setup a connection with the DSRT server.

- `int CpuApi::reserve(CpuReserve* cr, int pid=0);`

  Make a reservation specified in `cr` for the SRT process `pid`. The calling process must have the right privilege to make a reservation for the SRT process `pid`. After the `reserve()` call, the SRT process `pid` still runs at normal TS priority. The real time execution is started by calling `start()`. Return 1/0 for success/failure.

- `int CpuApi::modify(CpuReserve* cr, int pid=0);`

  Modify the current reservation specified in `cr` for the SRT process `pid`. The calling process must have the right privilege to modify the reservation for the SRT process `pid`. Return 1/0 for success/failure.

- `int CpuApi::free(int pid=0);`

  Free the current reservation for the SRT process `pid`. The calling process must have the right privilege to free the reservation for the SRT process `pid`. Return 1/0 for success/failure.

- `int CpuApi::start(timeval begin = TIMEVAL_INIT, int pid=0);`

  Start the real time execution for the SRT process `pid` at a future time `begin`. If `begin` is not specified, it is the current time. The calling SRT process blocks until time `begin`. A reservation must be made prior to `start()`. Return 1/0 for success/failure.

- `int CpuApi::stop(int pid=0);`

  Temporarily stop the real time execution for the SRT process `pid`. The calling process must have the right privilege to modify the reservation for the process `pid`. The process `pid` becomes a normal time sharing priority process. However, its reservation is still valid. The process must have called `start()` prior to `stop()`. The SRT process `pid` can be restarted by calling `start()` again. Return 1/0 for success/failure.

- `void CpuApi::yield();`

  Mark the end of an iteration (or period) for the calling SRT process. This applies to both periodic and aperiodic reservations. The calling SRT process must have started the real time execution by calling `start()`. The SRT process is blocked until the beginning of the next period.

- `int CpuApi::setAdaptStrategy(AdaptStrategy* as, int pid=0);`

  Set the adaptation strategy specified in `as` for a SRT process `pid`. The calling process must have the right privilege to set the adaptation strategy for the SRT process `pid`. Return 1/0 for success/failure.

- `int CpuApi::setApDeadline(int deadline);`

  This is for a CPU_RT_ACPU process to set its relative deadline (in us) for its next iteration. Return 1/0 for success/failure.

- `int CpuApi::probe(int period = 0);`

  Mark the beginning of the probing phase for the calling SRT process. The calling process must not have any reservation. If the calling process is a periodic process (with fixed period), then it must specify period in μs. If the calling process is an aperiodic process, it does not specify any period. Return 1/0 for success/failure.

- `int probeEnd();`

  Mark the end of the probing phase for the calling process. Return 1/0 for success/failure.

- `int probeMatch(CpuReserve* cr);`

  Determine the suitable CPU reservation for the probing process. It can be called after `probeEnd()`. The CPU reservation is returned in `cr`. Return 1/0 for success/failure.

- `int setRecordProcStats(char* fname=NULL);`

  Record the process usage information for the calling process in a file "fname-ps.dat". If fname is NULL, the file name is "pid-ps.dat". Return 1/0 for success/failure.

- `int setRecordSchedStats(char* fname=NULL, Long samplingInterval=1000000);`

  Record CPU server usage information in a file "fname-ss.dat" every interval samplingInterval. If fname is NULL, the file name is "pid-ss.dat". Return 1/0 for success/failure.

- `ProcStats* getProcStats(int pid=0);`

  Return the process usage information in the previous iteration for a real time process `pid`.

- `SchedStats* getSchedStats();`

  Return the CPU server status.

- `int isPermNonConform();`

  Return 1/0 if the calling process is/is not permanently non-conforming.

The C++ APIs also define four structures. `CpuReserve` structure represents the CPU service class specification in Table 2.1. `ServiceClass` specifies one of four CPU service classes:

CPU_RT_PCPT for periodic constant processing time class, CPU_RT_PVPT for periodic variable processing time class, CPU_RT_ACPU for aperiodic constant processing utilization class, and CPU_RT_EVENT for event class. Depending on the service class, other parameters need to be specified according to the CPU service class definition. Period, peakProcessingTime, sustainableProcessingTime, and burstTolerance are in units of μs.

```
typedef struct CpuReserve {
  int serviceClass;
  Long period;
  Long peakProcessingTime;
  Long sustainableProcessingTime;
  Long burstTolerance;
  double peakProcessingUtil;
} CpuReserve;
```

The adaptation strategy specification is represented by the AdaptStrategy structure. The strategy field specifies one of two adaptation strategies: ADAPT_EXPO for exponential adaptation strategy and ADAPT_STAT for statistical adaptation strategy. The windowSize and alpha values are for exponential adaptation strategy. The windowSize and nonconformFreq values are for statistical adaptation strategy. The max and min values specify upper and lower bounds on the guaranteed parameter that can be adjusted by the DSRT system.

```
typedef struct AdaptStrategy {
  int strategy;
  int windowSize;
  double alpha;
  double max;
  double min;
  double nonconformFreq;
} AdaptStrategy;
```

The process status is represented by the ProcStats structure. The DSRT server updates this structure at the end of each iteration during the execution phase of a SRT process. clock is

the CPU server clock time (μs) when this status is recorded. `usage` is the actual usage time in μs in this iteration. `nonconformFlag` is 1/0 when this iteration is non-conforming/conforming. `cpuReserve` is the amount of the CPU reservation in this iteration, which can change if adaptation is used. `apDeadline` is the relative deadline for an ACPU process. `permNonconformFlag` is 1/0 depending on whether or not the SRT process is permanently non-conforming or not. `overrunItr` is the number of periods that the SRT process misses its deadline. We can compute the amount of time that the SRT process missed its deadline as `(overrunItr * period)`. `accUsage` is the accumulated usage (μs). `accNumNonconform` is the cumulative number of iterations that are non-conforming. `accNumPermNonconform` is the cumulative number of iterations that are permanently non-conforming. `accNumDeadlineMiss` is the cumulative number of iterations that have deadline misses. `accItr` is the number of iterations since the start of the execution phase.

```
typedef struct ProcStats {
  Long clock;
  Long usage;
  int nonconformFlag;
  CpuReserve cpuReserve;
  Long apDeadline;
  int permNonconformFlag;
  int overrunItr;
  Long accUsage;
  int accNumNonconform;
  int accNumPermNonconform;
  int accNumDeadlineMiss;
  int accItr;
} ProcStats;
```

The server status is represented by the `SchedStats` structure. It contains information such as the available permanent partition on each processor and the usage loads on the permanent and overrun partitions. The DSRT server updates this structure at the end of each time slice. Interested readers can visit the DSRT system software release webpage [32] for further detail.

4.3.3 Java APIs

The Java APIs are encapsulated in a class called CpuJApi, which are almost identical to that of the C++ APIs. The Java APIs are implemented using Java JNI, which calls the C++ APIs. Therefore, their definitions are omitted here. Interested readers can go to the DSRT system software release webpage [32] for further detail.

4.4 Solaris 2.6 Implementation

We have implemented and tested the DSRT system on top of the Solaris 2.6 operating system and on uni-processor and multi-processor machines with the different hardware configurations shown in Table 4.1. We use the following system services: `setitimer(ITIMER_REAL)` for the real time interval timer, `priocntl()` for manipulating the real time priority of SRT processes, and `processor_bind()` to bind SRT processes to processors.

Machine Name	Machine Type	Processors	Memory	Dispatch Latency
boston.cs.uiuc.edu	SUN Ultra-1	1xUltraSPARC 143MHz	128MB	$< 0.8ms$
prague.cs.uiuc.edu	SUN Ultra-2	2xUltraSPARC 200MHz	256MB	$< 0.4ms$

Table 4.1: Solaris 2.6 machine configurations and dispatch latency measurement.

Solaris 2.6 operating system has a default global priority range 0 (lowest) to 159 (highest). There are 3 priority classes: RT class, System class, and TS class. The RT class contains fixed priority range 0-59, which maps to global priority range 100-159. The DSRT server runs at the highest global priority 159. The dispatched SRT process runs at the second highest global priority 158. Other SRT processes wait at the lowest global priority 0.

The highest resolution for the interval timer is $10ms$. This determines the lower bound for the *time slice (T)*.

We have measured the *dispatch latency*, which is the amount of processor time to switch execution from one SRT process to the next dispatched SRT process. This includes the following overheads: (1) two context switches from one SRT process to DSRT server and from the DSRT server to the next SRT process, (2) execution of the partition scheduling algorithm, and (3) system calls in the priority dispatch mechanism. The dispatch latency measurement is shown in Table 4.1. Given a time slice of $10ms$, the DSRT system overhead is less than 8% on a

Ultra-1 and 4% on a Ultra-2. The DSRT system overhead is counted as processor usage of the dispatched SRT processes.

4.5 Irix 6.5 Implementation

We have implemented and tested the DSRT system on top of SGI Irix 6.5 operating system and on uni-processor and multi-processor machines with the different hardware configurations shown in Table 4.2. We use the following system services: `setitimer(ITIMER_REAL)` for the real time interval timer, `sched_setscheduler()` for manipulating the real time priority of SRT processes, and `sysmp(MP_MUSTRUN_PID)` to bind SRT processes to processors.

Machine Name	Machine Type	Processors	Memory	Dispatch Latency
urbana.cs.uiuc.edu	SGI O2	1x175MHz IP32	64MB	$< 0.45ms$
jack.isl.uiuc.edu	SGI Onyx2	4x250MHz IP27	1024MB	$< 0.09ms$

Table 4.2: Irix 6.5 machine configurations and dispatch latency measurement.

Irix 6.5 operating system has a RT priority range 0 (lowest) to 255 (highest). The DSRT server runs at the second highest RT priority 254. The highest RT priority 255 is reserved for the real time interval timer. The dispatched SRT process runs at the third highest RT priority 253. Other SRT processes wait at the lowest TS priority 0.

We have measured the dispatch latency in Table 4.2. Given a time slice of $10ms$, the DSRT system overhead is less than 4.5% on an O2 and 0.9% on an Onyx2.

4.6 Windows NT 4.0 Implementation

We have implemented a static version of SRT system [44, 11] on top of Windows NT 4.0 operating system and on uni-processor and multi-processor machines with different hardware configurations shown in Table 4.3. We use the following system services: `SetTimer()` for the real time interval timer, `SetPriorityClass()` for manipulating the real time priority of SRT processes, and `SetProcessAffinityMask()` to bind SRT processes to processors.

The priority level in Windows NT 4.0 is shown in Table 4.4. There are four possible priority classes for a process and seven possible priority levels within each priority class. The main

Machine Name	Machine Type	Processors	Memory	Dispatch Latency
rome.cs.uiuc.edu	HP Vectra Xu	2x200MHz Pentium	96MB	$0.64ms$
moscow.cs.uiuc.edu	HP Vectra Xu	1x200MHz Pentium	64MB	$0.64ms$

Table 4.3: Windows NT 4.0 machine configurations and dispatch latency measurement.

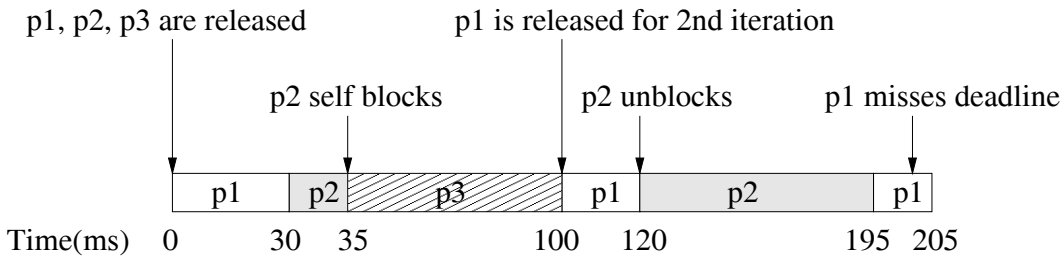
Process Priority Level	Thread Priority Level
IDLE_PRIORITY_CLASS	THREAD_PRIORITY_IDLE
NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_LOWEST
HIGH_PRIORITY_CLASS	THREAD_PRIORITY_BELOW_NORMAL
REALTIME_PRIORITY_CLASS	THREAD_PRIORITY_NORMAL
	THREAD_PRIORITY_ABOVE_NORMAL
	THREAD_PRIORITY_HIGHEST
	THREAD_PRIORITY_TIME_CRITICAL

Table 4.4: Process and thread priority structure in Windows NT 4.0.

thread in each NT process has a scheduling priority range 0 (lowest) to 31 (highest). Each thread's priority is a combination of the priority class of its process and the priority level of the thread within the priority class. The REALTIME_PRIORITY_CLASS is a fixed priority class in the range of $(16 \sim 31)$.

The priority structure in Windows NT 4.0 can be easily mapped to the priority structure required by the user-level priority dispatch mechanism. The static SRT server runs at the highest priority 31 (`REALTIME_PRIORITY_CLASS` + `THREAD_PRIORITY_TIME_CRITICAL`). The dispatched SRT process runs at RT priority 26 (`REALTIME_PRIORITY_CLASS` + `THREAD_PRIORITY_TIME_HIGHEST`). Other SRT processes wait at the lowest priority 0 (`IDLE_PRIORITY_CLASS` + `THREAD_PRIORITY_IDLE`).

We have measured the *average* dispatch latency, over 10000 dispatches, to around $0.64ms$. The time slot is set to 20ms, and the average overhead is 3.2%. However, we have encountered a problem in the periodic timer (`SetTimer()` system call), which may fail to wake up the dispatcher on time when the dispatched SRT process overruns its assigned slot. When an overrun occurs, the invocation of the dispatcher may be delayed for as long as the dispatched SRT process requires to finish its overrun. This means that one SRT process overrun can delay the dispatching of the process in the next time slot.



	Utilization	Period
p1	30%	100ms
p2	40%	199ms
p3	20%	300ms

Figure 4.3: A self-blocking process may cause violations to guarantees of other processes.

4.7 Self-Blocking Condition

A *Self-blocking condition* occurs when a client process is selected for dispatching on a processor, but it is blocked on some input data or a semaphore. A typical example of a self-blocking condition is when a process is waiting on a blocking read call from a network socket whose data does not arrive on time. In the current implementation, the DSRT server does not detect any self-blocking condition. It will do a priority dispatch on a self-blocking SRT process that has the earliest deadline for one time slice regardless of whether it is self-blocking or not. Given that the self-blocking process cannot run, the kernel scheduler may select a runnable system thread (hopefully it will unblock the self-blocking condition), or a TS process to run instead. If the dispatched client process becomes unblocked before the time slice ends, the dispatched process immediately preempts the running process and runs for the reminder of the time slice. However, the self-blocking client process is charged for the amount of self-blocking time.

The DSRT server assumes that it is the responsibility of the application developers to make sure that their SRT processes are always eligible to run at the beginning of each period. Application developers can minimize the probability of self-blocking occurrences by using asynchronous unblocking system calls. For example, a SRT process can use the asynchronous nonblocking call, e.g., `aio_read()` on Solaris, to read from a network socket.

It is possible to implement self-blocking detection on SRT processes in the DSRT system, and then to modify the DSRT scheduling algorithm so that it dispatches only non-blocking

SRT processes. However, we decide against such modifications. The reason is that under some special circumstances, it may cause violations to the contracts of SRT processes. Figure 4.3 shows an example where a violation occurs for $p1$ if the scheduler replaces a self-blocking process $p2$ with a non self-blocking processes $p3$. At time $30ms$, p_2 is dispatched but it becomes self-blocked at time $35ms$. As a result, a non self-blocking process p_3 is dispatched instead. At time $120ms$, p_2 is unblocked. However, it causes $p1$ to miss its deadline at time $200ms$.

4.8 Limitations

There are several features in the general purpose operating systems and hardware that can potentially cause violations to the guarantees that the DSRT system makes to SRT clients, even if SRT clients have passed the admission control test. The *hardware interrupt* is one such limitation. A hardware interrupt is usually serviced at the hardware interrupt priority level, which is higher than the priority level that the DSRT server can manipulate in the user-level. If a hardware interrupt occurs at the time when the DSRT server is executing the scheduling algorithm, it will preempt the DSRT server and will adversely lengthen the dispatch latency and the dispatch precision. If a hardware interrupt occurs at the time when a dispatched process is running, it will preempt the dispatched process and will adversely delay the processing time of the preempted dispatched process.

Solving the hardware interrupt problems requires non-trivial changes in the structure of the general purpose operating system. Instead of servicing the interrupts whenever they occur, the operating system would periodically poll them. In addition, the operating system would need to distinguish interrupts generated by TS applications, which it will service only when there is free time, and interrupts generated by SRT client processes, which it will service immediately.

The other limitation that can cause unpredictable performance is the layered memory subsystem. The layered memory subsystem improves the average performance at the expense of higher worst case performance. As a result, many hard real time systems turn the cache off.

We do not advocate for changes in the general purpose operating systems or hardware. It would have been an overkill for SRT systems and applications. Instead, the DSRT system quantifies these adverse effects in a parameter called the *system-specific burst tolerance ratio (SSBTR)*, which is described in section 3.1.1. $SSBTR$ accounts for processing time variability

caused by the use of underlying general purpose operating system and hardware. We also recommend that the client applications re-do probing when adding/removing memory in the system to accurately account for the memory performance.

Chapter 5

Performance Evaluation

5.1 Experimental Setup

All experiments are performed on a Sun Ultra-2 machine with two processors (2xUltraSPARC 200MHz processors) and 256MB of memory, running Solaris 2.6 operating system. Each experiment contains a different set of periodic SRT processes running concurrently with a set of TS processes. These SRT processes are programmed according to the *Three-Phase Execution Flow* described in section 2.1. They first use the probing service to extract the reservation parameters, and then make reservations based on the extracted parameters.

The performance metric is the *inter-frame time*. The inter-frame time measures the amount of time between the *job completion time* in each consecutive iteration. Figure 5.1 shows the inter-frame time of a MPEG video decoder that runs at 10 frames per second (fps). A decoding job is released at the beginning of every $100ms$, called the *job released time*, e.g., $0ms$, $100ms$, $200ms$, $300ms$, ... For example, the second job is released at $100ms$ (starting to decode the second frame) with a deadline at $200ms$. It is completed at $160ms$ (displaying the second frame) before its deadline. The third job is released at $200ms$ (starting to decode the third frame) with a deadline at $300ms$. It is completed at $270ms$ (displaying the third frame) and before its deadline. The inter-frame time between the second and third frames is $110ms$. The ideal inter-frame time is $100ms$. However, the ideal time is rarely seen because (1) the amount of decoding time is different from one frame to another, and (2) the amount of system load (jobs from other processes with earlier deadlines) may vary from one frame to another. As a

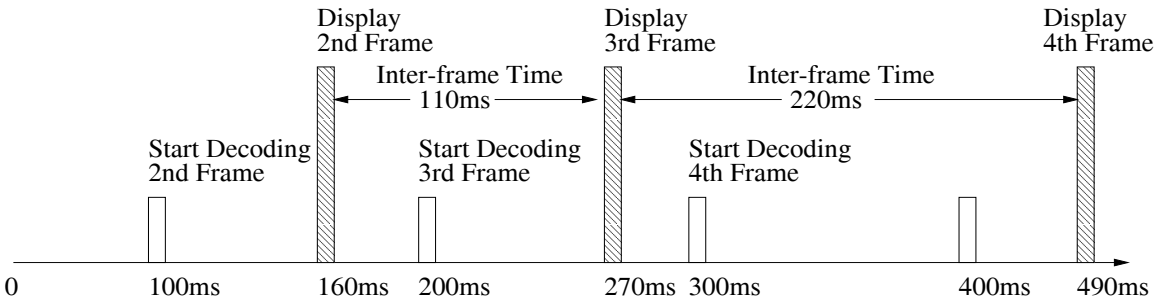


Figure 5.1: The inter-frame time for a MPEG video decoder running at 10 frames per second.

result, we may observe some small variance in the inter-frame time, which is hardly noticeable by the viewers at high frame rate.

If the system makes no provisions for resource allocation and real time scheduling (no DSRT system running), it is possible for some frames to miss their deadlines, causing large and noticeable variance in inter-frame time. As shown in Figure 5.1, the third job is released at $300ms$ (starting to decode the third frame) with a deadline at $400ms$. However, it is completed at $490ms$ (displaying the fourth frame), passing its deadline. The inter-frame time between the third and fourth frames is $220ms$. If the MPEG video decoder is scheduled by the DSRT system and it conforms to its contract, this should not occur. It should be able to decode and display a frame in every $100ms$ interval.

In order show the resource guarantees by the DSRT system, we introduce the following set of TS processes ($TS_{1..8}$) in the experiments. They act as the TS loads and they run concurrently with the measured SRT processes. We will show that they do not cause any performance degradations to the measured SRT processes[1]. We also introduce a set of misbehaving SRT processes ($SRT^{mb}_{1..2}$) which try to grab as much processor time as possible. The misbehaving SRT processes are used to show the *overrun protection* that their overruns do not cause violations on the guarantees to other behaving SRT processes.

- $TS_{1..6}$: Six identical TS processes repeatedly compute sin and cos tables using an infinite series formula.

- $TS_{7..8}$: Two identical TS processes repeatedly compile the Berkeley MPEG video decoder.

[1]We assume that these TS processes do not generate too many hardware interrupts or kernel service requests.

		Processor Usage Pattern	
MPEG Video Streams	Resolution	First 20 Frames	500 Frames
4dice.mpg	352x240	$PPT = 40ms$ $SPT = 28ms$ $BT = 11ms$	See Figure 2.2
Twister.mpg	352x240	$PPT = 38ms$ $SPT = 25ms$ $BT = 17ms$	See Figure 5.2(a)
Simpsons-temple.mpg	192x144	$PPT = 21ms$ $SPT = 14ms$ $BT = 6ms$	See Figure 2.4
Simpsons-dentist.mpg	192x144	$PPT = 23ms$ $SPT = 18ms$ $BT = 3ms$	Figure 5.2(b)
Simpsons-mcbain.mpg	192x144	$PPT = 21ms$ $SPT = 15ms$ $BT = 6ms$	See Figure 5.2(c)
MPEG Audio Stream	Sampling Frequency		
Ifeelgood.mp2	44.1 kHz	$PPT = 14ms$[2]. $SPT = 13ms$ $BT = 2ms$	See Figure 5.2(d)

Table 5.1: MPEG video and audio streams.

- $SRT^{mb}_{1..2}$: Two misbehaving PCPT class processes make PCPT reservations with parameters ($P = 100ms, PPT = 10ms$). They perform computation (sin and cos tables) forever and never yield (do not call `yield()`).

5.2 First Experiment

The first experience consists of the following six SRT processes ($SRT_{1..6}$) running *concurrently* with the TS loads ($TS_1..8$) and the misbehaved SRT loads ($SRT^{mb}_{1..2}$). All six SRT processes are MPEG video decoders, playing three different MPEG streams at different framerates (period) from 20 fps ($50ms$) to 5 fps ($200ms$). The tested MPEG streams and their processor usage patterns are described in Table 5.1.

- SRT_1: a PVPT class MPEG video decoder plays the "Simpsons-temple.mpg" stream at 20 frames per second. Its processor usage pattern is shown in Figure 2.4. It uses

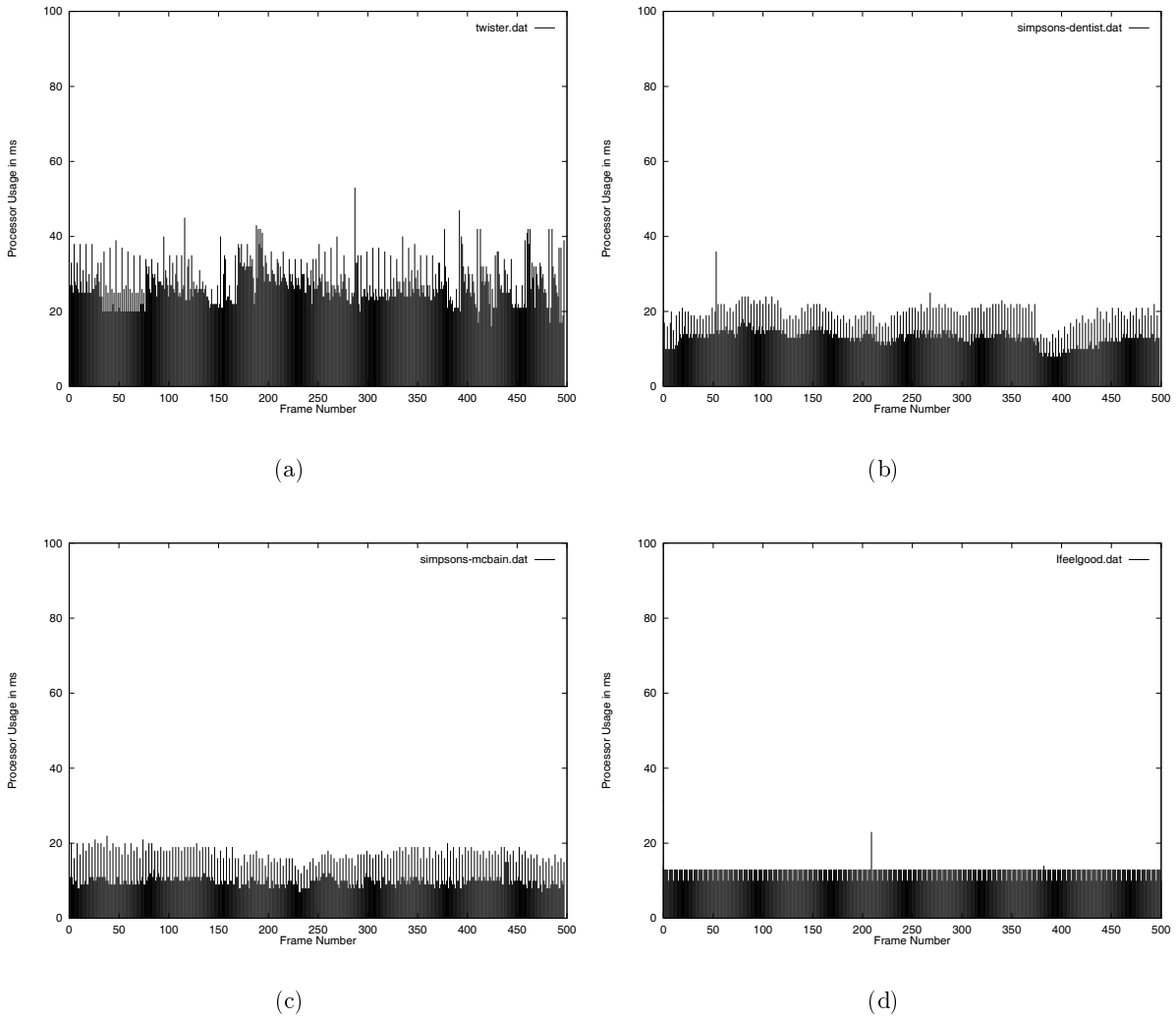


Figure 5.2: Iteration-by-iteration processor usage history for three MPEG video and audio streams described in table 5.1.

the probing service (20 iterations) to configure a PVPT reservation with parameters $(P = 50ms, PPT = 21ms, SPT = 14ms, BT = 6ms)$.

- SRT_2: a PVPT class MPEG video decoder plays the "Simpsons-dentist.mpg" stream at 13.3 frames per second. Its processor usage pattern is shown in Figure 5.2(b). It uses the probing service (20 iterations) to configure a PVPT reservation with parameters $(P = 75ms, PPT = 23ms, SPT = 18ms, BT = 3ms)$.

- SRT_3: a PVPT class MPEG video decoder plays the "Simpsons-mcbain.mpg" stream at 10 frames per second. Its processor usage pattern is shown in Figure 5.2(c). It uses

the probing service (20 iterations) to configure a PVPT reservation with parameters $(P = 100ms, PPT = 21ms, SPT = 15ms, BT = 6ms)$.

- SRT_4: a PVPT class MPEG video decoder plays the "Simpsons-temple.mpg" stream at 8 frames per second. Its processor usage pattern is shown in Figure 2.4. It uses the probing service (20 iterations) to configure a PVPT reservation with parameters $(P = 125ms, PPT = 21ms, SPT = 14ms, BT = 6ms)$.

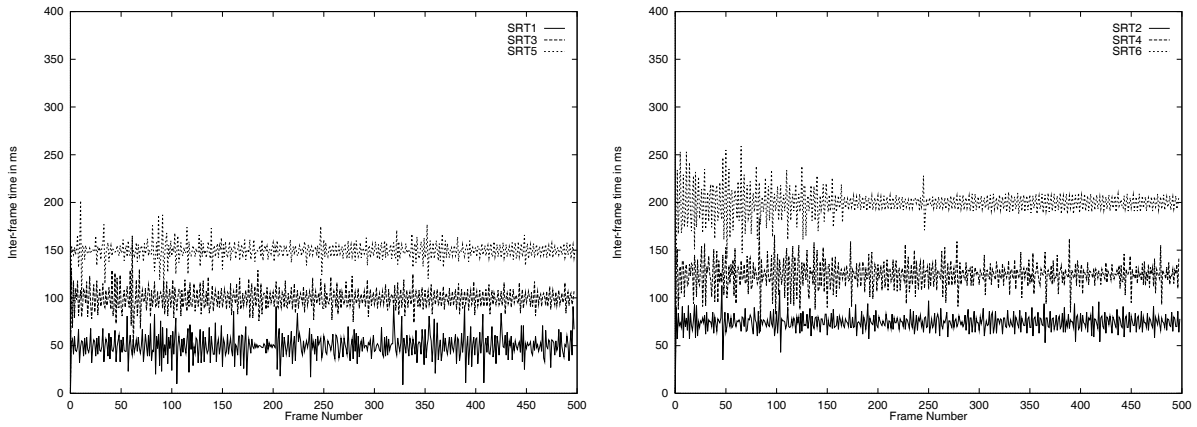
- SRT_5: a PVPT class MPEG video decoder plays the "Simpsons-dentist.mpg" stream at 6.6 frames per second. Its processor usage pattern is shown in Figure 5.2(b). It uses the probing service (20 iterations) to configure a PVPT reservation with parameters $(P = 150ms, PPT = 23ms, SPT = 18ms, BT = 3ms)$.

- SRT_6: a PVPT class MPEG video decoder plays the "Simpsons-mcbain.mpg" stream at 5 frames per second. Its processor usage pattern is shown in Figure 5.2(c). It uses the probing service (20 iterations) to configure a PVPT reservation with parameters $(P = 200ms, PPT = 21ms, SPT = 15ms, BT = 6ms)$.

Figures 5.3 show the inter-frame time for the SRT processes $SRT_{1..6}$. There are no deadline misses for any of the SRT processes. The inter-frame time for SRT processes stays within the bound of their perspective one period time. This experiment shows that SRT processes can perform well on a shared server running the DSRT system, given the following circumstances: (1) a variety of periodicity from the SRT processes, (2) the presence of a heavy TS load, and (3) the presence of a misbehaving SRT load. We would like to draw a reasonable analogy that each of these SRT process represents a task sent from a client device to the shared computing server.

5.3 Second Experiment

The second experiment consists of the following four SRT processes ($SRT_1..4$) running concurrently with the TS loads ($TS_1..8$) and the misbehaved SRT loads ($SRT^{mb}_{1..2}$). The four SRT processes include a variety of applications: a MPEG video decoder, a sampling program, a MPEG audio decoder, and a Java game. The tested MPEG streams and their processor usage patterns are described in Table 5.1.



(a) (b)

Figure 5.3: The inter-frame time for six SRT processes ($SRT_{1..6}$) in the first experiment.

- SRT_1: a PVPT class MPEG video decoder plays a 352x240 "Twister.mpeg" stream at 20 frames per second. Its processor usage pattern is shown in Figure 5.2(a). It uses the probing service (20 iterations) to configure a PVPT reservation with parameters ($P = 50ms, PPT = 38ms, SPT = 25ms, BT = 17ms$).

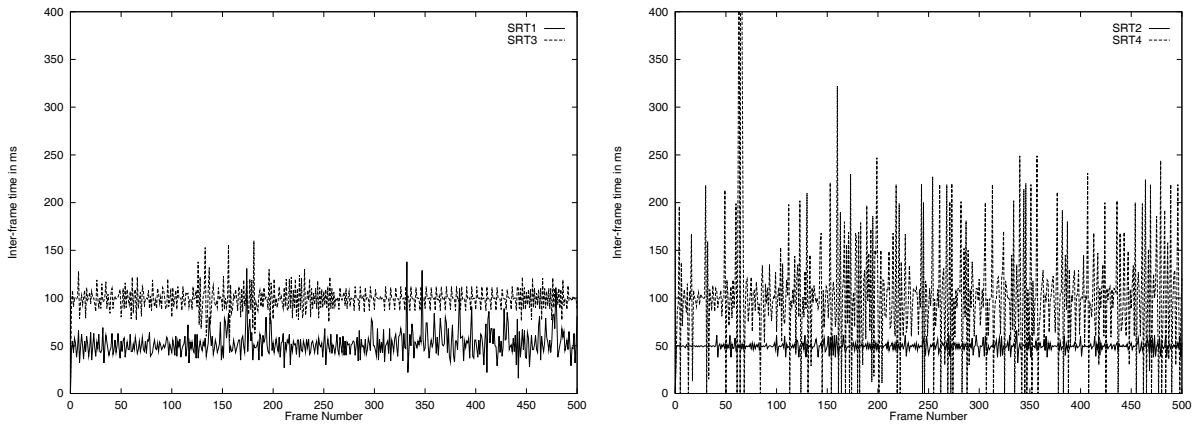
- SRT_2: a PCPT class sampling program performs a fixed amount of computation every $50ms$. Its processor usage pattern is shown in Figure 2.3. It makes a PCPT reservation with parameters ($P = 50ms, PPT = 10ms$) without any adaptation.

- SRT_3: a PVPT class MPEG audio decoder plays the "Ifeelgood.mp2" stream at 10 frames per second. Its processor usage pattern is shown in Figure 5.2(d). It uses the probing service (20 iterations) to configure a PVPT reservation with parameters ($P = 100ms, PPT = 14ms, SPT = 13ms, BT = 2ms$). This audio stream contains 38 samples per second. The processor usage pattern is based on decoding on average 3.8 samples and writing these samples to the audio device every $100ms$.

- SRT_4: a Java RocksInSpace game controls a space ship that can shoot missiles at incoming rocks. The game updates movements of objects (ship, rocks, missiles) every $100ms$. It makes a PCPT reservation with parameters ($P = 100ms, PPT = 30ms$).

Figures 5.4 show the inter-frame time for the SRT processes $SRT_{1..4}$. There are no deadline misses for $SRT_{1..3}$. Their inter-frame time stays within the bound of their perspective one period time. However, some deadline misses are recorded on the Java RocksInSpace game SRT_4. The causes are: (1) Java virtual machine performs garbage collection at random interval which freezes the thread that is running the animation and (2) the Java virtual machine also makes a lot of blocking system calls to the kernel.

This experiment shows that the non-Java SRT processes can perform well on a shared server running the DSRT system, given the following circumstances: (1) a different variety of SRT processes with different periodicity and processor usage patterns, (2) the presence of a heavy TS load, and (3) the presence of a misbehaving SRT load.



(a) (b)

Figure 5.4: The inter-frame time for four SRT processes ($SRT_{1..4}$) in the second experiment.

5.4 Third Experiment

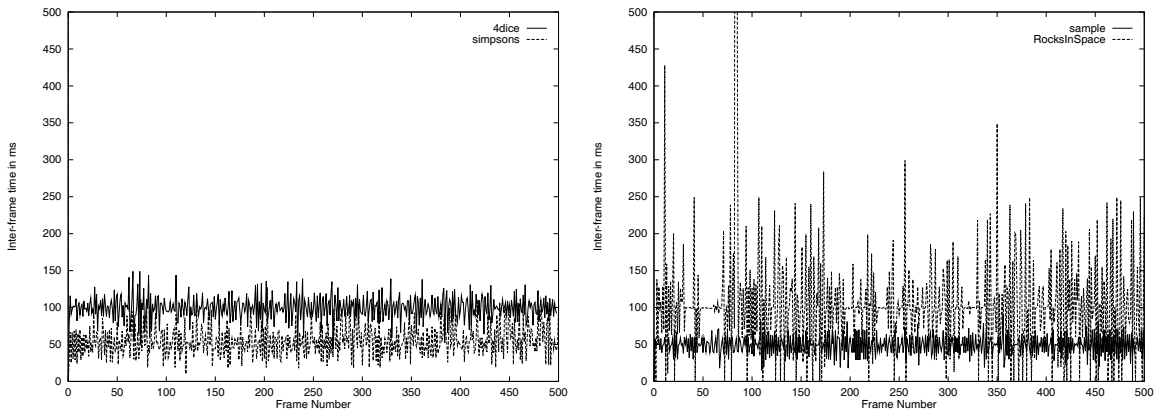
The third experiment consists of the following four SRT processes ($SRT_1..4$) running concurrently with the TS loads ($TS_1..8$) and the misbehaved SRT loads ($SRT^{mb}_{1..2}$). These four SRT processes include a similar mixture as in the second experiment. However, we test a MPEG video stream where adaptation is needed. The tested MPEG streams and their processor usage patterns are described in Table 5.1.

- SRT_1: a PVPT class MPEG video decoder plays a 352x240 "4dice.mpg" stream at 10 frames per second. It uses the probing service (20 iterations) to configure a PVPT reservation with parameters ($P = 100ms, SPT = 28ms, PPT = 40ms, BT = 11ms$). It also selects the statistical adaptation strategy (*non-conforming overrun frequency=0.2, window size=20*).

- SRT_2: a PVPT class MPEG video decoder plays a 192x144 "Simpsons-temple.mpg" stream at 20 frames per second. It uses the probing service (20 iterations) to configure a PVPT reservation with parameters ($P = 50ms, PPT = 21ms, SPT = 14ms, BT = 6ms$).

- SRT_3: a PCPT class sampling program performs a fixed amount of computation every $50ms$. Its processor usage pattern is shown in Figure 2.3. It makes a PCPT reservation with parameters ($P = 50ms, PPT = 10ms$) without any adaptation.

- SRT_4: a Java RocksInSpace game controls a space ship that can shoot missiles at incoming rocks. The game updates movements of objects (ship, rocks, missiles) every $100ms$. It makes a PCPT reservation with parameters ($P = 100ms, PPT = 30ms$).



(a) (b)

Figure 5.5: The inter-frame time for the four SRT processes ($SRT_{1..4}$) in the third experiment.

Figures 5.5 show the inter-frame time for four SRT processes $SRT_{1..4}$. There are no deadline misses for two MPEG video decoders SRT_1 and SRT_2 and the sampling program SRT_3. Their inter-frame time stays within the bound of their perspective one period time. However,

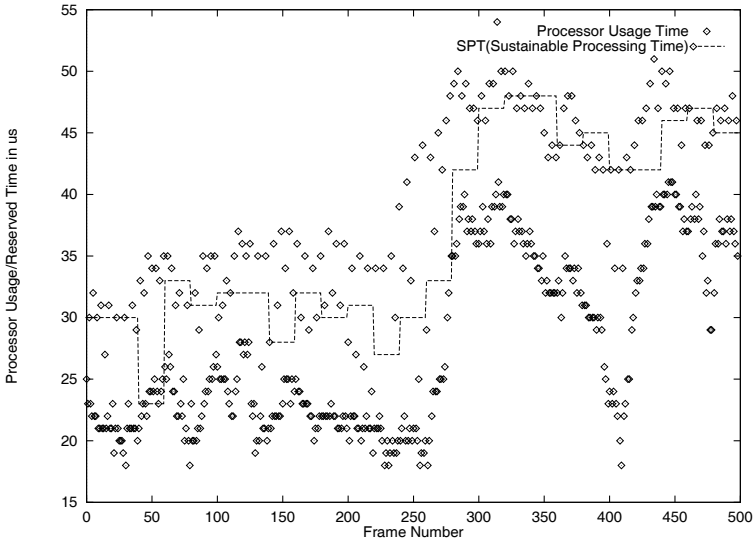


Figure 5.6: Statistical adaptation strategy adjusts SPT (sustainable processing time) shown as the line for SRT_1. The dots represent its actual processor usage.

some deadline misses are recorded on the Java RocksInSpace game SRT_4 for the same reason explained in the second experiment.

Figure 5.6 shows how the statistical adaptation strategy adjusts the SPT parameter for the SRT_1 MPEG video decoder. The SPT value is adjusted from the initial $28ms$ to approximately $45ms$ when the processor usage time increases after frame 270.

Chapter 6

Related Work

In this chapter, we describe related work to the DSRT system. We divide it into the following two sections. Section 6.1 describes related RT systems that provide hard/soft RT support for RT applications in the general purpose operating system environment. Our literature survey finds 16 creditable systems that have implementation with well-known publications. We describe each of them with an emphasis on the differences to the DSRT system.

Section 6.2 describes related work that uses analytical methods to validate probabilistic performance guarantees for hard/soft applications with dynamic processor usages. Again, we describe each of them with an emphasis on the differences to the DSRT system.

6.1 General Purpose RT Systems

We use the following classification criterions to both differentiate among and describe various related general purpose RT systems with respect to the DSRT system. These criterions are defined below:

- *Reservation-Based System or Best-Effort-Based System*

  We define a reservation based system as one that provides performance or resource guarantees to applications. In order to support guarantees, a reservation-based system must use some forms of admission control test (schedulability test) on reservation requests. For example, the DSRT system is a reservation-based system. On the other hand, a best-effort

based system does not provide performance or resource guarantees. A best-effort based system does its best to allocate its resources to satisfy resource requests of applications.

- *Usage Specification:*

  Different systems provide different usage specification for RT applications to describe their processor usage patterns. If systems are priority-based, RT applications can also describe their relative priorities to other applications. For example, the DSRT system provides a rich set of CPU service classes to RT applications. Other related systems may provide a combination of priority, weights, execution time (equivalent to processing time), or other metrics.

- *Dynamic Processor Usage*

  Dynamic processor usage is defined as processor usage variations across different iterations of a RT application. Bursts and overruns are examples of these variations and they are common character of multimedia applications running on top of general purpose operating systems. Different systems have different methods to deal with the dynamic processor usage. For example, the DSRT system contains specification for usage bursts, and it provides overrun protection by scheduling overruns/bursts in a separate overrun partition.

- *Scheduling algorithm*

  Different systems use different scheduling algorithms. For example, the DSRT system has a hierarchical partitioning scheduling design where the RT partition is scheduled by an EDF queue, and the overrun partition is scheduled by multi-level round robin priority queues.

- *Implementation*

  Different systems are implemented in different ways. They require different amount of kernel modifications, from significant changes to no changes. The DSRT system is implemented as middleware which requires no changes to the kernel.

6.1.1 Real Time Mach

RT Mach [47] implements the *Processor Capacity Reserves* abstraction for the RT threads. RT Mach provides processor time reservation and guarantees for RT threads. A new RT thread first

requests its CPU QoS in the form of a period, and requested CPU time in a percentage during the reservation phase. Once the request is accepted, a *capacity reserve* of CPU processing time is established and it is bound to this new RT thread. Any computation time done on behalf of this RT thread, including all service time from various system threads, is charged to the capacity reserve of this RT thread. The capacity reserve is replenished periodically and for the amount equal to its reserved CPU time. A resource monitor/allocation tool [48] is implemented to display and to change the reservation dynamically.

RT Mach also supports three dynamic QoS control mechanisms[42]. (1) Each RT thread specifies a *quality range* [max, min, desirable], and *a quality adjustment policy* on how to adjust its reservation in case of dynamic behavior. RT Mach offers five different quality adjustment policies: adjusting its period while keeping its computation time per period constant, adjusting its computation time while keeping the period constant, maintaining the computation time within a range, adjusting the computation time by a specified discrete step, or allowing the RT threads to make adjustments by giving it a maximum value. (2) Each RT thread is associated with a *quality adjustment priority* which decides the order which the system performs the adjustment. (3) RT Mach has different admission control policies on how to adjust the reservation of existing RT threads while admitting new RT threads. For example, an admission control policy can adjust the existing RT threads to their minimum quality to try to accommodate the newly admitted RT threads. RT Mach is implemented in the Mach Kernel.

The DSRT system differs from RT Mach in the following aspects. (1) The DSRT system supports a variety of CPU services classes for RT applications with different processor usage patterns, e.g., PVPT, PCPT, and ACPU classes. RT Mach supports only one static type of reservation which is equivalent to DSRT's PCPT class. RT Mach does not accommodate bursts and does not provide any statistical guarantees on the bursts. (2) The DSRT system supports the probing service which can extract the reservation parameters from the processor usage history of RT applications. RT Mach requires users to configure the reservation based on previous experience or by trial-and-error. (3) The DSRT system has specifications for the adaptable range on reservation parameters [max, min], which is similar to the RT Mach quality adjustment policies. In addition, the DSRT system has specification for the algorithmic adaptation strategies (e.g., exponential average, and overrun bounding algorithms) on how to adjust the reservation parameters in the range, which are not found in RT Mach. In the

DSRT system, the RT applications can tune the parameters (e.g. frequency, alpha, or overrun frequency) in the algorithms that best suit their dynamic processor usage patterns. (4) The DSRT system is implemented in the user-space whereas the RT Mach is implemented in the kernel. The kernel implementation allows RT Mach to have much smaller scheduling overhead and to accurately account for kernel service time on behalf of the RT threads. However, the user-level implementation of the DSRT system is more portable to different UNIX operating systems.

The DSRT system shares many similar features with RT Mach, e.g., processor time reservation and guarantees, overrun protection, and guaranteed TS allocation. These features are considered essential in any RT systems in the general purpose environment.

6.1.2 Adaptive Rate Controlled Scheduler

Adaptive Rate-Controller (ARC) scheduler [68, 67] is an elegant modification of the virtual clock scheduling algorithm [70] to schedule processes instead of network packets. A RT process specifies two reservation parameters: a reserved rate r ($0 < r \leq 1$), and a time interval p. The admission control test checks that the sum of all rates is less than or equal to 1, $\sum r \leq 1$. RC scheduler computes the *RC* values (similar to the expected finish time in the virtual clock algorithm) for RT processes. RC values are incremented by the amount of processor usage in proportional to their reserved rate (usage/r). A RT process that tries to run ahead of its reserved rate (tries to overruns) will have a larger RC values than a RT process that runs at its exact reserved rate, whereas a RT process that runs behind its reserved rate (underruns) will have a smaller RC values. RC scheduler dispatches a process with the smallest RC value first. It can guarantee that all punctual RT processes are scheduled for (k*r*p) processor time over time interval (r*p). At the same time, it provides overrun protection among RT processes. When multiple processes all try to run ahead of their reserved rate (overruns), competing for limited processor time in an overloaded situation, the RC scheduler behaves as proportional sharing.

RC scheduler uses a rate adaptation algorithm to adjust the reserved rate. Rate adaption computes two values for an adapting RT process: *lag* measures how far a process is running ahead of its reserved rate, and *lax* measures the unused reserved processor time. If the monitored lag is greater than a *lag tolerance*, the RT process is notified to increase its reserved rate. If the

monitored lax is greater than a *lax tolerance*, the RT process is notified to decrease its reserved rate. Rate adaptation is an elegant extension of the RC algorithm. The ARC scheduler is implemented as a loadable module in the Solaris kernel.

Except for the handling of adaptation which is innovative to RC algorithm, the ARC scheduler provides similar functionalities as RT Mach. Hence, the DSRT system differs from ARC scheduler in similar aspects as it differs from RT Mach. However, it is possible to augment the ARC scheduler with CPU service classes and other advanced services in the DSRT system such as probing, adaptation strategies, overrun scheduling, and multiprocessor support.

There are also differences in the handling of overruns between the ARC scheduler and the DSRT system. For example, when the system is under-loaded and a RT process runs ahead of its reserved rate, ARC scheduler will schedule all overruns. As a result, the overrunning RT process will accumulate a large RC value, which grows proportionally with processor usage. However, when the system becomes over-loaded at a later time, this RT process will be penalized by its large RC value even if it is no longer overrunning. In other words, the penalty for overrunning may be carried over to a later time in the ARC system. This carry-over penalty does not exist in the DSRT system. Overruns, which are scheduled in the overrun partitions, have no adverse effect on the future reserved runs scheduled in the RT partitions.

6.1.3 SMART

In SMART [53, 54], a RT task specifies both a *priority* value and a weighted *share* of the processor bandwidth. There are two types of RT tasks, one with a deadline (called deadline task), and one without a deadline (called conventional task). SMART schedules RT tasks based on a tuple value consisting of a combination of *importance* (or priority) and *urgency* (or deadline). A task has a higher priority if its tuple value has a higher priority, or the same priority but earlier urgency. Since a conventional task does not have a deadline, its tuple contains only importance. The SMART scheduling algorithm can be summarized as follows. If a task with the highest priority is a conventional task, then that task is scheduled. Otherwise, a task with the earliest deadline, whose execution does not cause any higher priority tasks to miss their deadlines, is scheduled. SMART scheduling algorithm computes *urgency* of a RT task as a function of its share of processor time in proportional to sum of shares from all RT tasks at the same priority. This is similar to the weighted fair sharing algorithm [17].

Unlike the DSRT system, SMART is not a reservation-based system. There is no admission control test in SMART to check if deadlines requested by RT tasks can all be met. In SMART, a RT task can specify its QoS request. However, the amount of processor time that a task receives is dependent on the processor usage of other higher priority tasks. For example, if a high priority task exhibits bursty processor usage, it may cause variable processor allocation to all lower priority tasks. There is no overrun protection.

SMART is designed to accommodate different RT applications with different properties running simultaneously. A deadline task requires a deadline driven scheduler, whereas a conventional task requires a proportional sharing scheduler. As a result, we classify SMART as a hybrid of proportional sharing and priority based system.

SMART has a multiprocessor design [55]. Each processor has a separate dispatch queue. For a new deadline task, SMART assigns a processor that can best satisfy its deadline while minimizing the occurrences of deadline misses by the existing RT tasks. This assigned processor should not contain any conventional tasks. For a new conventional task, SMART computes the priority of processors as the maximum priority of its assigned tasks, and it assigns a processor with the minimum priority. SMART is implemented inside the the Solaris kernel.

The multiprocessor design in the DSRT system is different from SMART multiprocessor design. The DSRT system processor assignment is computed during the admission control test, and it ensures that the assigned processor has enough processor resource to satisfy the new RT process. The SMART thread-to-processor assignment is based on other factors, such as the processor separation of conventional tasks and deadline tasks, and load balancing.

6.1.4 Rialto at Microsoft Research

Rialto [35] at Microsoft Research brings real time support in the Windows NT. A RT activity is represented as a time constraint (processor time estimate, start time, and deadline), and priority (critical, or non-critical). Rialto realizes that figuring out the required processor time of any piece of code is a hard problem in a general purpose operating system and hardware environment. As a result, processor time is estimated through feedback from previous execution of the same code of an activity. For activities of different priority, Rialto schedules higher priority activities. For activities of the same priority, Rialto uses a minimal laxity first scheduling

algorithm. Laxity is defined as the amount of time from the current time to the latest time at which a time constraint's code could be initially scheduled without missing its deadline.

A recent version of Rialto [37] adds continuous CPU reservations to RT activities. CPU reservations are in the form reserve X units of time out of every Y units. A CPU reservation is specified alone with a time constraint. X/Y can be more or less than the processor time estimate/(deadline - start time). It also adds a feasibility test (admission control test), in the form of a pre-computed scheduling graph, to check if a new CPU reservation can be accepted. A new CPU reservation at a critical priority can steal CPU resource from an existing CPU reservation at a non-critical priority. The recent version also replaces the laxity-first scheduling algorithm at runtime with an EDF algorithm. Through the feasibility test and the EDF scheduling algorithm, it can provides guarantees to all CPU reservations. We classify Rialto as a reservation based system.

The DSRT system differs from Rialto in the following aspects. (1) Rialto does not support dynamic CPU usage patterns. It support only one static type of reservation, which is equivalent to the PCPT class in the DSRT system. (2) Rialto does not deal with changing processor usage patterns. It makes no provisions for reservation adaptation. (3) The processor usage estimation in Rialto is different from the probing service in the DSRT system. In Rialto, processor usage estimation is done at execution time (after the CPU reservation is done) based on processor usage value from the previous iteration. In the DSRT system, the probing service runs during the probing phase (prior to the reservation) and is based on a history of usage values.

Rialto has a few add-ons, such as Vassal [9]. Vassal allows RT applications to design their own schedulers and to dynamically load them into the Windows NT kernel. The benefit is that different RT applications can use their own schedulers for their specific needs simultaneously. However, interference can occur when two different applications are loading two conflicting schedulers. Perhaps a hierarchical scheduler approach described in section 6.1.5 can be used to create protection among different schedulers.

The Rialto group also performs a careful study on the dispatch latency, the true causes for variance on dispatch latency, and the Timer call-back functions in Windows NT kernel [36, 34].

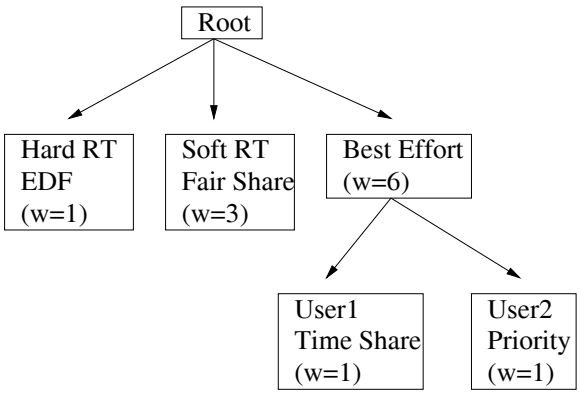


Figure 6.1: Application class tree for hierarchical CPU scheduler.

6.1.5 Hierarchical CPU Scheduler

Hierarchical CPU Scheduler [31] partitions processor resource into hierarchical application classes in a tree-like structure as shown in Figure 6.1. Different scheduler can be employed for applications in each different leaf class. For example, the hard RT class can use an EDF scheduling algorithm whereas the soft RT class can use a fair share scheduling algorithm. All leaf application classes are scheduled by a Start-time Fair Queuing (STFQ) algorithm, which allocates processor bandwidth according to the relative weights in the application classes. For example, the hard RT application class will get 10% of the processor bandwidth and the $user_1$ application class will get 60%*50%=30% of the processor bandwidth. The STFQ algorithm is a modified version of the weighted fair sharing algorithm. It provides protection among various application classes. Hierarchical CPU Scheduler is implemented in the Solaris kernel.

The DSRT system differs from Hierarchical CPU Scheduler on the following aspects. (1) In the DSRT system, the CPU service classes are used by RT applications with different processor usage patterns (e.g., constant/variable processor usage). In Hierarchical CPU Scheduler, the application classes are used by RT applications based on the most suitable scheduling algorithms at their classes. (2) In the DSRT system, partitions divide the processor bandwidth according to reserved runs, overruns, and TS loads. In Hierarchical CPU Scheduler, application classes divide the processor bandwidth according to different scheduling algorithms. (3) Since Hierarchical CPU Scheduler runs at the level of leaf class schedulers, it does not address issues at the level of individual application such as reservation, overrun, and adaptation.

There is also fundamental differences on how to deal with different processor usage patterns among RT applications that share the same CPU. The DSRT system allows the application developers to specify their different processor usage patterns through CPU service classes, and to let the system satisfy their specifications. On the other hand, the Hierarchical CPU Scheduler allows application developers to specify their scheduling algorithms that can best handle their different processor usage patterns through hierarchical application classes. For application developers, we believe that specification of processor usage patterns in the DSRT system is a simpler model to use than specification of scheduling algorithms in Hierarchical CPU Scheduler.

6.1.6 Open System

Open System [20, 19, 18] is based on the concept that each RT task has its own virtual processor, equivalent to a fraction of real processor utilization that the RT task specifies. Within its virtual processor, a RT task designs its own scheduling algorithm and validates schedulability for its applications independently of other RT tasks. The innovation of Open System is that it can provide timing guarantees for hard RT periodic or sporadic tasks. Because of the hard RT performance guarantees, Open System assumes that RT applications provide precise *a-priori* information about their execution time, sizes of their critical sections, and any factors that may affect the hard RT performance guarantees. Then these information goes through a carefully designed schedulability test to ensure hard RT performance guarantees.

Open System has a two level hierarchical scheduling design. The lower level scheduler (*OS scheduler*) schedules the higher level schedulers (*servers*) which contain scheduling algorithms provided by RT tasks on their own virtual processors. Open System is implemented in Windows NT kernel.

Open System combines features from Vassal which applications can load their own scheduling algorithms into the kernel, and Hierarchical CPU Scheduler which the system allows many levels of schedulers. However, Open System improves upon them with its schedulability analysis for hard RT applications.

The DSRT system differs from Open System in the following aspects. (1) The DSRT system is a soft RT system, whereas the Open System is a hard RT system. (2) Because it is a hard RT system, Open System assumes no overruns and no changes in processor usage patterns. Because

the execution time are known *a-priori*, Open System does not need to make any provisions for overrun protection and adaptation.

6.1.7 Nemesis

In most general purpose operating systems, the kernel is a multiplexing point that performs services requested by applications. However, multiplexing in the kernel can create interference, (QoS crosstalk), among RT applications in such a way that the kernel may not be able to distinguish which requests are more urgent than the others. The Nemesis [4, 43] eliminates such interference by moving as much processing from the kernel (e.g., protocol stacks) into shared library code linked by applications. It tries to shift as much kernel processing as possible into the user-space.

Nemesis takes a different approach from RT Mach client thread which passes its reserve to a kernel thread that performs the processing using the client's capacity reserve.

Nemesis uses an EDF algorithm to schedule RT applications. The processor usage specification is in the form of (s, p, x, l): s is amount of processor time required every period of length p, x is a boolean value used to indicate whether the application is prepared to receive slack time or not, and l is a latency hint. l is usually the relative deadline, or p. However, if the application requires faster response time, l can be less than p. The usage specification is similar to RT Mach. Overruns are sorted in an EDF queue. When there are no reserved runs, an overrun with the earlier deadline is scheduled.

Nemesis uses a *feedback QoS model* to implement adaptation in the application. Application performance is fed into a QoS Controller that decides if the application quality needs to be adjusted. Then the QoS Controller informs a QoS Manager which then re-allocates resources to satisfy the new quality.

The DSRT system differs from Nemesis in the following aspects. (1) The DSRT system provides a more dynamic set of usage specification and a probing service. (2) The adaptation service in the DSRT system is system-initiated such that the system implements the adaptation algorithm, instead of user-initiated adaption in Nemesis. (3) The DSRT system uses a multi-level round robin algorithm to schedule overruns. Nemesis uses an EDF queue to schedule overrun, which tends to favors applications with shorter deadlines. Nemesis also requires sig-

nificant modifications in the general purpose operating system, e.g. protocol stacks and device drivers.

6.1.8 Feedback-Driven Proportion Allocation

The Feedback-driven Proportion Allocation [61] is a system that can accommodate RT applications that require processor reservation (RT threads), and RT applications that do not know how to derive their reservations but still require constant throughput (real-rate threads). The solution is a feedback system that observes the *progress* of the RT applications and adaptively changes the allocation based on the observed progress. The observation is based on a *buffer* in consumer-producer model, which a producer thread produces output data to a buffer and a consumer thread consumes input data from the buffer. If the buffer is nearly full, the progress on the consumer thread is considered too fast, and its processor allocation will be adjusted downward. On the other hand, if the buffer is near empty, the progress on the producer thread is considered too slow and its processor allocation will be adjusted upward. One important assumption in this feedback-driven approach is that applications need to be designed in this consumer-producer model. For some applications that cannot tolerate potential long latency introduced by the various buffers in the system, this feedback-driven approach may not be applicable.

The Feedback-driven Proportion Allocation contains an admission control test for reservation-based applications. The usage specification for reservation-based threads is the same as that of RT Mach. When an overloading situation occurs, it sheds processor allocation among the RT applications that exceed their weighted fair-share. The Feedback-driven Proportion Allocation is implemented on top of Linux and it uses SWiFT[29] for its adaptive controller.

The DSRT system differs from Feedback-Driven Proportion Allocation in the following aspects. The DSRT system solves the problem on how to extract the reservation parameters from applications through the use of its probing service. The DSRT system also supports a dynamic set of usage specification (CPU service classes) which enables it to capture different processor usage patterns in its reservation.

6.1.9 Dynamic QoS Resource Manager

Dynamic QoS Resource Manager (DQM) [6, 7] is a policy-independent mechanism that uses resource allocation information from the operating system and execution level information from applications to balance the load. One major assumption is that applications can support a range of qualities depending on the load in the system. The usage specification is represented by several execution levels, each level consists of three values (quality level, resource usage, and benefit). An application developer designs several execution levels (one for each quality level), and submits them to the DQM. During runtime, the DQM selects the execution level based on the available resource in the operating system. Similar to the DSRT system, this is a system-initiated adaptation. DQM has several different *selection algorithms* with different goals in selecting the different execution levels of running applications. Among them, the *fair algorithm* allocates CPU fairly among all applications. The *optimal algorithm* maximizes the total benefits of applications in the system.

Like the DSRT system, DQM is implemented as middleware. Unlike the DSRT system, DQM is only a best-effort system. DQM makes no guarantees on resource allocations to SRT applications, except that DQM may be fair or optimal in achieving application benefits based on the selection algorithm. It has no admission control policy to reject new admissions of SRT applications even when the existing SRT applications are running at their minimum quality levels.

6.1.10 TAO

TAO [28, 60] is a real-time object request broker (ORB) end-system that supports QoS requirements for real-time CORBA applications. TAO modifies the I/O subsystem in the general purpose operating system so that they can execute I/O protocol stacks in real time. The general approach in TAO is to dedicate a separate queue and a kernel thread to run the protocol stack for RT or TS connection(s) of equal priority. Then TAO schedules the kernel threads to satisfy the QoS requirements of real time connections.

TAO solves the following two RT issues found in most general purpose operating systems. (1) Packets from multiple connections are multiplexed into the same queue served by kernel threads regardless of their priority or urgency. The kernel threads simply service the queue in

FIFO fashion, and they do not distinguish between high priority packets from QoS connections or low priority packets from TS connections. (2) Kernel threads do not inherit priority from user-level threads which send or receive packets to or from them. This is a potential for priority inversion. These two problems are also discussed in Nemesis (called QoS crosstalk). However, TAO employs kernel threads to run the protocol stack, which is different from Nemesis which shifts the protocol stack processing into a shared library linked by applications.

TAO requires CORBA applications to specify resource usage on their real time I/O operation in the form of period, worst-case execution-time, criticality, and dependency. It employs a static fixed priority implementation of the rate monotonic (RM) algorithm. It assigns the highest fixed RT priority to a kernel thread that serves the connection with the highest rate (or the smallest period). The priority assignments are calculated off-line and they are *static*. TAO also performs an admission control test to accept or reject new resource requests based on resource availability.

TAO focuses on a different area than the DSRT system. TAO is specializing on building a SRT I/O subsystem and protocol stack in combination with CPU scheduling. On the other hand, the DSRT system is an independent CPU resource scheduler which can be used to build SRT I/O subsystems by providing SRT support to I/O resource brokers, e.g., communication broker or disk broker, as in the case of QualMan architecture described in chapter 7.

If we only compare the CPU scheduling capabilities between TAO and the DSRT system, the DSRT system has the following advantages over TAO. (1) TAO does not provide overrun protection, where a high priority kernel thread serving a misbehaving connection with the highest rate can monopolize CPU and can block access to other kernel threads serving connections with lower rate. (2) TAO also realizes that the worst-case execution time may lead to under-utilization of the resources. However, given their specification, using the worst-case execution time was necessary to provide guarantees to the applications. The DSRT system contains service class specification for bursty processor usage behavior. (3) TAO suggests that the worst-case execution time may be determined by simulation, instruction counting, or benchmarking, which may be difficult. The DSRT system uses the probing service to extract the reservation parameters.

TAO shares one similarity with the DSRT system in that it is implemented as middleware on top of Solaris.

6.1.11 Proportional Share Resource Allocation (PSRA)

Stoica, *ed. al.* [62] proposes a framework that unifies both reservation-based and proportional share resource allocation. The system is divided into two classes: *reservation class* and *proportional share class*. The reservation class allows a client application, which requires a precise amount of the CPU, to specify a *share* of the CPU resource, e.g. a percentage of the total CPU resource. Reservation-based clients are guaranteed their shares.

The proportional share class allows a client application, which requires a proportional share of the CPU resource, to specify a *weight*. The proportional share client receives a share relative to the aggregate weight in the proportional share class. The share, which a proportional share client receives, is not constant at all time. It depends on relative weight and shares of other clients in the system.

A variant of the virtual clock scheduling algorithm, called the earliest eligible virtual deadline first (EEVDF) algorithm [63], is developed to allocate the CPU resource according to the weight and shares of the clients. Similar to the DSRT system, clients are scheduled in units of time quantums. PSRA is implemented in the FreeBSD.

The DSRT system differs from the PSRA in the following aspects. (1) In PSRA, the CPU resource specification contains two classes, one class for proportional sharing clients and one class for reservation based clients. These two classes do not support clients that have dynamic processor usage patterns, and at the same time they require reservations. However, it is possible to augment the DSRT system with a proportional sharing partition to accommodate proportional sharing clients. (2) In PSRA, reservation-based clients are allocated CPU according to its share, and overruns (not caused by imprecision from the time quantum allocation) are not accommodated. In the DSRT system, a separate overrun partition is setup for overruns.

6.1.12 Soft Migration at Intel Architecture Labs

Soft Migration project at Intel Architecture Labs is aimed to help transition of SRT processing from dedicated special purpose processors at peripheral devices, (e.g, digital signal processors on Modem and sound cards, 3D chips on 3D graphics cards) to Intel's general purpose CPU(s). The soft modem guideline [41, 40, 15] shows modem developers how to build a kernel mode driver that performs the DSP processing on a system's general purpose CPU(s) rather than

on a dedicated DSP chip on a modem card. The soft migration can help to reduce the cost of peripheral devices if processing can be shared on the general purpose processor. For example, a soft modem without a dedicated DSP chip is considerably cheaper than a modem with a DSP chip. However, proper RT scheduling on the CPU system is required to service multiple peripherals.

There are no published works on the scheduling framework in a soft migration project. However, it is included here to demonstrate the values and applicability of RT scheduling on general purpose operating systems. It can reduce the amount of dedicated hardware on desktop computers, which translates into lower cost.

6.1.13 Others

CPU inheritance scheduling [21] is a framework in which arbitrary threads can act as schedulers for other threads. Threads form a tree-like hierarchical structure. Intermediate/root threads are schedulers which contains their own scheduling algorithms to schedule their own child threads. Leaf threads are application threads. Parent threads schedule their child threads by donating CPU time to them. This design is similar to Hierarchical CPU scheduler described in section 6.1.5

Lottery Scheduling [66] is another variant of proportional sharing resource allocation with a hierarchical allocation design. The intermediate or root nodes contain *currency value* which are the relative weight among their immediate sibling nodes. The leaf node contains a client thread which specifies a weighted share of the resource within its immediate parent node. The client receives lottery tickets based on its relative weight. Scheduling among clients is probabilistically proportional to their weight or the number of lottery tickets. A client with more lottery tickets are more likely to be scheduled, and that probability is its relative weight over the sum of all weight in the system. When the number of allocation increases, the actual allocation will match that of its expected probabilistic allocation. This is similar to PSRA except that it introduces probability.

Gopalakkrishnan [30] implemented the *Real Time Upcall (RTU)* to support periodic tasks. Each RTU contains an event handler that registered with the kernel its execution time and period. The kernel dispatcher is modified to schedule (or call) the RTUs using the rate monotonic

algorithm. In order to increase the predictability and efficiency, the kernel dispatcher disallows preemptions during the middle of any RTU execution.

URsched [38] is a simple user-level realtime scheduler based on the POSIX 4.0 real time extension. It introduces the basic concept of user-level priority dispatch which is refined and expanded in the DSRT system.

6.2 Probabilistic Performance Guarantees

Tia, *ed. al.* [64] defines a semi-periodic task as a task with a regular released time; however, its processor usage time may vary widely across iterations. A system with total maximum utilization of periodic/semi-periodic tasks greater than 1 has some probability that some tasks will miss their deadlines. Analytical methods can be used to compute the upper bound on the probability of deadline misses. Ideally, the admission control policy can incorporate such methods to derive a precise probabilistic guarantee to processes with variable execution time (e.g., the bursts in PVPT class). However, these analytical methods are applicable to systems under the following assumptions.

1. Total average utilization is small and the system is only rarely overloaded.

2. Tasks are independent and in phase.

3. The system uses a fixed-priority scheduling algorithm, such that tasks are assigned static priorities.

These three assumptions may be valid in hard RT systems. However, assumptions 1 and 2 may not be desirable in most soft RT systems. A soft RT system can tolerate occasional violations, but it may need to achieve high average resource utilization. A fixed-priority system (e.g., using a rate monotonic algorithm) may not be desirable because it does not provide as optimal utilization as a dynamic-priority system (e.g., using an EDF algorithm).

Gardner, *ed. al.* [24] improves upon Tia's work by relaxing assumption 1. His stochastic analysis to bound the probability of deadline misses is still restricted to only fixed-priority systems. For dynamic-priority systems, stochastic analysis becomes too complex and simulations [25] are used instead.

Given such analytical methods are not applicable for most soft RT systems, the DSRT system uses a different approach to provide probabilistic guarantees for variable execution time (e.g., bursts) in SRT processes. The DSRT system takes a monitor-and-adjust approach. It allows system administrator to specify SGB (statistical guarantee on bursts), which is the maximum probability that SRT processes can tolerate their bursts that miss their deadlines. The DSRT system monitors $\overline{SGB}$, the actual probability of any bursts that miss their deadline. When $\overline{SGB} > SGB$ occurs, the size of the overrun partition is increased to accommodate more bursts.

Chapter 7

QoS-Aware Resource Management (QualMan)

For distributed multimedia applications, it is insufficient to have only the CPU resource management provided by the DSRT system at the endpoints. Distributed multimedia applications require the *end-to-end quality of service (QoS)*. For example, users of video/audio conferencing applications may demand a bounded end-to-end delay and jitter so that they can have a meaningful conversation without the annoying out-of-sync effects. Figure 7.1 shows a distributed multimedia environment where we consider the end-to-end QoS issues.

The environment consists of general purpose workstations or PCs equipped with multimedia devices such as video cameras, microphones, and speakers. The multimedia end-points are connected via local area networks such as ATM (Asynchronous Transfer Mode), which are currently widely available in academia and industry. In order to provide end-to-end quality to the video conferencing application in this environment, it requires the following steps to be done in real time:

- The sending process at the sender's host is scheduled in time and periodically to receive video/audio data from a video camera or a microphone. Then it feeds the data into a real time transport protocol stack.

- The real time transport protocol needs to deliver the video/audio data to the receiving host in time.

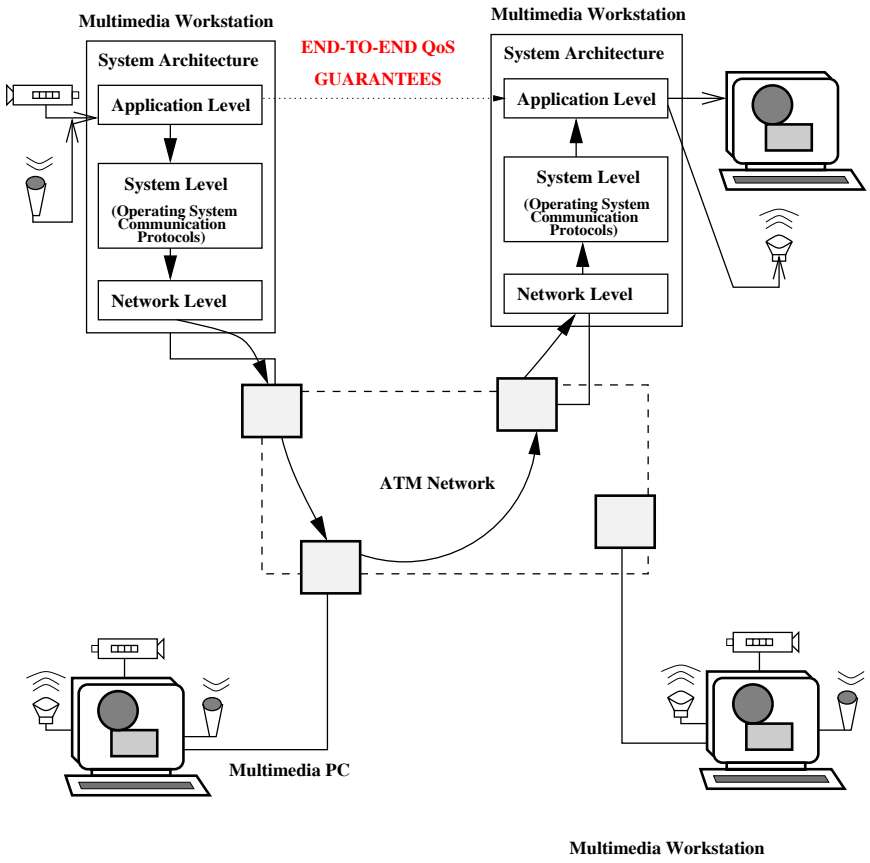


Figure 7.1: The end-to-end scenario of distributed multimedia applications.

- The receiving process at the receiver's host is scheduled in time and periodically to receive the video/audio data from its transport protocol. Then it feeds the data on display or to a speaker.

We need to extend the resource management protocol to include both the endpoint and communication resources that are on the execution path of distributed multimedia applications. This leads to the design and implementation of the distributed QoS-Aware multi-resource management called QualMan. Various types of shared resources including CPU, memory, and network are managed by QualMan in which the DSRT system is used as its CPU resource manager. In addition, QualMan aims to achieve the following goals:

- Provide a uniform view and access to various types of resources (e.g., CPU, memory, and network). Applications can reserve resources through a uniform QoS specification.

- Provide a general resource model for resource admission control, scheduling, enforcement, adaptation, and negotiation. The model must be able to provide guarantees based on reservations specified by the applications through resource scheduling or enforcement.

- Provide a middleware solution that is implemented in user-space without modifications in the kernel. The advantages of a middleware solution are (1) *portability* to different UNIX operating system platforms, (2) *loadablity* without reconfiguration of the kernel, and (3) *flexibility* to different applications, and (4)*scalability* toward multiprocessor architecture. However, the disadvantage is a lack of very fine resource control which can be achieved only at the kernel level.

The main goal is to show the integration of the individual servers and to present the end-to-end QoS measurements with the QualMan architecture. We organize this chapter as follows. Section 7.1 presents the design of QualMan. Section 7.2 describes the CPU server. Section 7.3 describes the memory server. Section 7.4 describes the communication server. Section 7.5 explains the implementation of QualMan. Section 7.6 evaluates the performance of QualMan. Section 7.7 states the contributions of QualMan.

7.1 Design

The design of QualMan contains two parts: the *QoS specification model* and the *resource model*. The specification model allows the applications to quantify the amount of resources they need during their reservation phase. The resource model is a general multi-resource management architecture that can provide resource guarantees to the resource specification model.

7.1.1 QoS Specification Model

We consider parameterization of the QoS because it allows us to provide quality-controllable services. There are many possible QoS parameters such as visual tracking precision, image distortion, packet loss rate, jitter of arriving frames, synchronization skew, and others. They can be classified from different aspects. One aspect is according to the layered multimedia communication architecture which consists of four main layers: users, application, system, and network layers[51]. If we assume this type of end-point layering, then we can separate QoS into

QoS type	Specification	QoS parameter	Symbol
Application QoS	Processing requirements	Sample size	M_A
		Sample size (I, P, B)	M_A^I, M_A^P, M_A^B
		Sample rate	R_A
		Number of frames per GOP	G
		Compression pattern	G_I, G_P, G_B
		Original size of GOP	M_G
		Processing size of GOP	M_G'
		Degradation factor	D
	Communication	End-to-end delay	E_A
		Synchronization Skew	$Sync_A$
		Jitter	J_A
System QoS	CPU	See Table 2.1	
	Memory	Pinned Memory(KB)	Mem_{req}
	Communication	Packet size	M_N
		Packet rate	R_N
		Bandwidth(Kbps)	B_N
		End-to-end delay(ms)	E_N

Table 7.1: Application and system QoS specification and parameters.

application QoS (e.g., 20 frames per second video), *system QoS* (e.g., 20ms processing time every 50ms period) and *network QoS* (e.g., 16 Mbps bandwidth) classes. This classification allows each layer to specify its own quality parameters. However, this classification also requires translations at the boundaries between individual layers[50]. Some examples of application and system QoS parameters for MPEG-compressed video streams are shown in Table 7.1.

In this chapter, we consider the *system QoS parameters* such as the CPU QoS, memory QoS, and communication QoS parameters for QualMan.

7.1.2 Resource Model

To provide QoS, each of the shared resources at the end-points must be modeled autonomously enough to provide its own QoS control. We extend the shared resource model with the brokerage functionality as shown in Figure 7.2. This general model allows us to provide a *uniform view* at any shared resource in a distributed multimedia system with QoS requirements. The uniform resource view then allows for development of feasible higher level heuristics algorithms to solve the distributed resource allocation problem which is otherwise NP-complete problem [58]. We

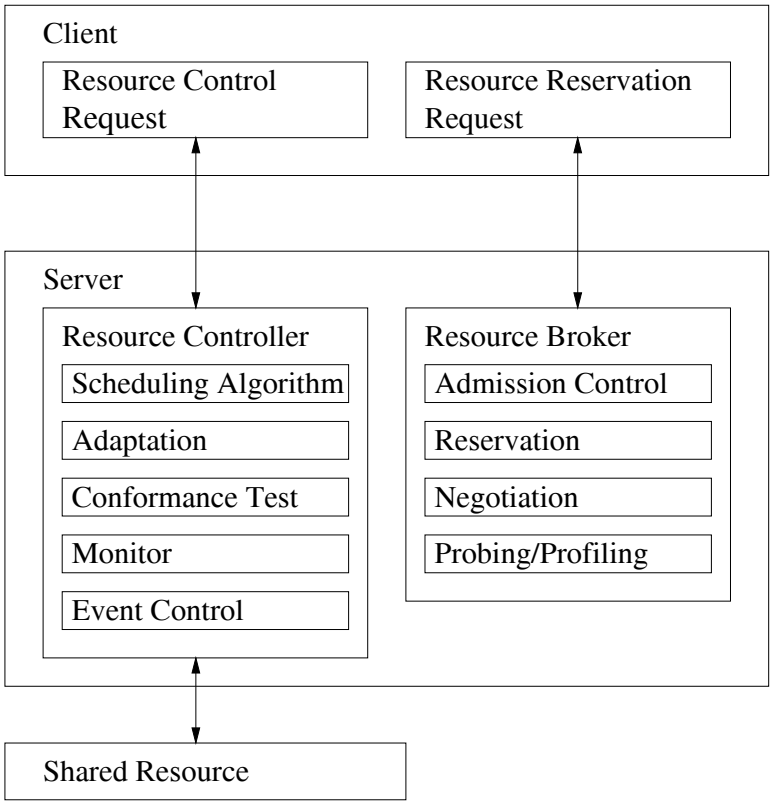


Figure 7.2: Resource model with corresponding services.

provide *piecewise solutions* at individual resource servers such as algorithms for resource reservation, admission control, enforcement, and adaptation. We also provide reservation protocols and coordination within communication protocols to integrate the distributed resource servers in an end-to-end computing and communication environment.

The access to a shared resource is based on a client/server model:

- The client process sends a resource request to the resource broker during the reservation/connection setup phase. If the request is accepted by the resource broker, it becomes a contract. Then the client utilizes the specified resources during its execution/transmission phase.

- The *server* contains two parts: the *resource broker* and the *resource controller*. The resource broker accepts requests from clients and performs admission control tests to check for resource availability. Note that in order for the resource broker to perform admission control, the client must provide the amount of resource needed using a common QoS

specification. If the client does not know the needed amount, then it can acquire this information through the *probing and profiling service* provided by the server[1]. The resource controller receives the resource contract which includes the resource usage parameters of the contract and also a feasible schedule from the resource broker. Then the resource controller performs the appropriate scheduling/allocation algorithm to satisfy the resource contract. The resource controller may also receive timing and event control messages from the client processes during their execution time. Furthermore, the resource controller is also responsible for QoS monitor and adaptation when QoS variations are observed.

We can apply this resource model to each resource type where individual brokers and resource controllers communicate with each other and create an integrated end-to-end solution as shown in Figure 7.3. The application layer contains a distributed resource management which coordinates the resource reservation and allocation among the various individual resource brokers at the system layer. This distributed resource management requires a distributed reservation protocol which is described in detail in section 8.2. The system layer is divided into the QualMan middleware and the general purpose operating system. The middleware interacts with the application layer through a set of middleware application program interfaces (API). The middleware consists of the various resource servers–CPU, memory, and communication. In the reminder of this chapter, we will discuss the design, implementation, and experimental results of our CPU, memory, and communication servers.

7.2 CPU Server

The CPU server is the DSRT system. Hence, we will not elaborate on this further.

7.3 Memory Server

The execution time of client's RT process also depends on the state of memory contention and the resulting number of page faults. We designed a *memory broker* where the RT process can reserve memory prior to their RT execution.

[1]The CPU probing/profiling service is discussed in section 3.1

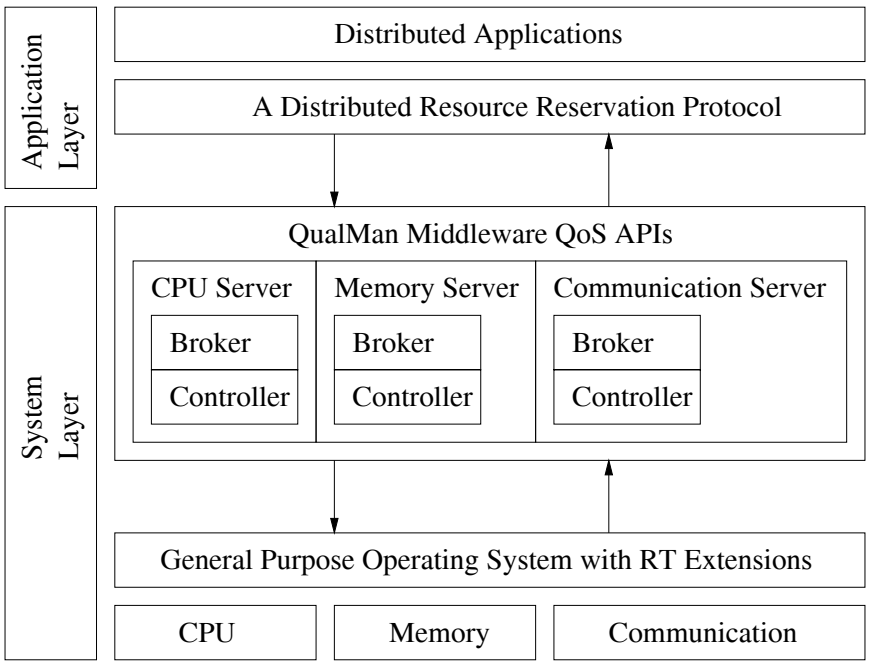


Figure 7.3: The QualMan middleware broker/controller architecture.

The memory server consists of the *broker* and the *memory scheduler* according to the resource model in Figure 7.2. The memory server is a root process that can be started at the system boot up time. It is initialized with a parameter called `global_reserve`, which is the maximum amount of pinned memory (in bytes) that the server can allocate to RT processes. The `global_reserve` should be chosen carefully so that it does not starve the TS processes and the kernel. The memory server waits for requests from RT processes.

The RT process begins with the reservation phase. It contacts the memory broker to try to establish a *memory reserve* with a specified amount of memory request in bytes. The reserve should be an estimated amount of the pinned memory that the process needs in order to satisfy its timing requirement. It should include all its text, data, and shared segments. Once the memory broker receives the request, it performs the following admission test:

$$Mem_{req} \leq Mem_{avail}, i.e., Mem_{req} + \sum_{j=1}^{k} Mem_{acc_j} \leq Mem_{glob-resv}$$

It checks that the incoming request for memory reserve, Mem_{req}, added to the already accepted memory reserves, $\sum_{j=1}^{k} Mem_{acc_j}$, does not exceed the global reserve, $Mem_{glob-resv}$. If the admission test succeeds, the memory broker returns a reserve id (`rsv_id`) to the process,

and it creates an entry in its table (`rsv_id`, Mem_{acc}). The process should then lock its text segment using the reserve `rsv_id`.

During (or prior to) the execution phase, the process can send the memory controller a request `(rsv_id, size)` to acquire the pinned memory data allocation (e.g., `malloc()`). Once the request is received, the server checks whether there is enough reserve to satisfy this request. If so, it decreases `size` bytes of memory from the reserve `rsv_id`. The server then allocates the pinned memory in the form of *shared memory* to the process. The server creates a shared memory segment of `size` using `shmid = shmget(key, size)` and locks it using `shmctl(shmid, SHM_LOCK)`. The shared memory key is then passed to the process which attaches the shared memory segment into its address space.

When the process wants to free its pinned memory, it detaches the shared memory segment and sends a request containing the shared memory key to the memory server. Then the server destroys the share memory segment and increases the corresponding memory reserve.

We choose not to apply probing and adaptation in our memory server because the application programmer can usually determine the actual amount of memory the process needs throughout its runtime. However, we do allow the process to re-negotiate the memory reservation with the memory server when the process finds that the amount reserved is insufficient.

7.3.1 Relation between Processes and Memory Reserves

The relationship between the memory reserves and processes can be many to many. A process can establish multiple reserves to protect memory usage among various parts of the same program. For example, a distributed video playback application can assign separate reserves for its display, decoded, and network buffers. It will restrict the growth of some buffers that use pinned memory. Multiple processes can also share the same reserve. For example, a distributed video playback application may require services from the network, decoder, or display processes (or modules/drivers) which can charge their memory usage to the application's reserve.

The underlying shared memory implementation also helps to eliminate the copying overhead when various processes need to pass data around. Consider a network module that assembles packets into frames and passes the frames to the decoder process. The network module and the decoder process can establish a joint memory reservation and create a common shared memory

region. The network module charges the reserve for every new frame it uses; the decoder process gets the frames through the shared memory region without copying.

7.3.2 Limitations

There are several limitations in our shared memory implementation of the memory reserve. The first is that the memory reserve covers only the text and data segments, but not the stack segment. We have found it difficult to monitor and manage the stack segment without modifications inside the kernel. In a typical program, its stack segment is usually much smaller than its text or data segments. Therefore, it is unlikely that the stack segment will get swapped out.

The second limitation is with the data allocation in the linked/shared library. Users can not modify the data allocations in the linked libraries (e.g., X library) to call our memory reserve routines. These data segments in these libraries are not pinned nor accounted for in the reservation.

We have chosen the shared memory implementation because it can be done at the *user-level* without modifications in the kernel. These limitations can be overcome with another choice of implementation which involves modifications to the virtual memory system. However, this would mean a defeat of the desired loadable capability which our current middleware has.

7.4 Communication Server

The next component in the QualMan architecture is the communication server developed by another Monet group member Narayan [52]. We will describe only the essential features of the communication server and the integration between the communication and the CPU servers. This shows how QualMan provides the end-to-end QoS guarantee for distributed multimedia applications. Similar to the CPU and memory servers, the communication server consists of two components according to the resource model in Figure 7.2: the *communication broker*, which admits and negotiates the network QoS, and the *multimedia-efficient transport protocol (METP)*, which enforces the communication QoS at the end-points and propagates the ATM QoS parameters/guarantees to the higher communication layers.

7.4.1 Communication Broker

The communication broker is a management daemon, which in conjunction with the transport protocol, provides QoS guarantees required by the distributed multimedia application. The broker performs service registration, admission control, negotiation, connection setup, monitoring and adaptation as follows:

- Service Registration:

  The multimedia application (RT client) sends a service request to the broker with the following quality specification: *peak, mean, and burst bandwidth* ($B_{peak}, B_{mean}, B_{burst}$), *size* of the application protocol data unit (APDU) M_A, *end-to-end delay* E_A, *specification of data flow* (either simplex or duplex), *reliability* enforcement (either total or partial), and *timeout* duration t_{out}, which specifies how long to wait for a PDU or for an acknowledgment in our reliability mechanism.

- Admission Control:

  Upon receiving the service request, the broker performs checks to verify that the service request can be guaranteed. It performs admission on bandwidth availability and end-to-end delays. For *bandwidth availability*, the admission condition is:

$$\sum_{i=1}^{k} B_{acc_i} + B_{req} \leq B_{HI}$$

  B_{acc_i} is the accepted bandwidth for the i-th connection and B_{req} is the requested bandwidth of the new connection. The *end-to-end delay* depends on a number of factors such as the application PDU size, load on the network, loads on the end hosts, and the bandwidth reserved for the connection. Admission control for end-to-end delay is performed using a *profiling scheme*. A profile of the end-to-end delays for various APDU sizes is measured off-line and used as the seed. For the CPU bandwidth and memory availability for METP processing, the communication broker contacts the CPU and memory servers. The communication broker needs to have information about the processing time C and size M_A corresponding to the processing of APDUs in the transport tasks (e.g., segmentation of APDUs to TPDUs, header creation, movement of PDUs) in METP. The period T of the transport tasks is derived from the frame rate R_A. The broker gets the size M_A

from the user who knows the size of the APDU to be sent out. The processing time PT of APDUs within transport tasks is acquired by the probing service. During the CPU probing time, the CPU broker monitors the processing times of the transport tasks and stores them in a corresponding QoS profile. The processing time includes the time of METP tasks after receiving APDU by METP to send the segmented TPDUs in a burst every $T_A = \frac{1}{R_A}$. The communication broker reads the profile of the processing time and uses the information to get reservations from the CPU broker for the transport tasks.

- Connection Setup:
  After the service requests are accepted at both the sender and the receiver, the brokers at both sides setup the ATM connection.

7.4.2 Multimedia-Efficient Transport Protocol (METP)

The communication server includes a thin layer of transport service support. For real time support, the transport layer requires an appropriate amount of CPU bandwidth and memory from the CPU and memory servers so that its transport tasks can move and process TPDUs in a predictable fashion. In addition, it expands the native ATM mode (AAL API) to provide the following features which are not provided by the AAL layer:

- Efficient zero copy send for the send operation.

- Reliability Mode.
  The transport layer can operate in two modes - a *totally reliable* mode and a *partially reliable* mode. In the totally reliable mode, there are no timing guarantees. The user is allowed to specify an expected QoS. An effort is made to fulfill the timing requirements, but priority is given to the reliability of delivery. In the partially reliable mode (also termed the real-time mode), the timing guarantees take precedence over reliability of delivery. The maximum time for *send* and *receive* operations can be specified. The transport layer sends as much data as possible within the time specified.

- Real time timed-out-data discard policy for the *send* operation.
  If a *send* operation takes longer than its allotted time, the sending side discards future

data until it catches up with the timer. This is done in anticipation of a discard on the receive side.

- Dynamic time-out value adjustment for the send and receive operations.
  Both the *send* and the *receive* operations use timers in order to provide real-time guarantees to the application. These timers are used for retransmission in the *send* operation, and acceptance or rejection of data in the *receive* operation. The time-out value is dynamically tuned the timeout value in order to achieve better throughput.

7.5 Implementation

7.5.1 Specific Issues about the Memory Server

In modern computer architecture, the memory hierarchy consists of 3 levels in decreasing order of access time: cache (1st and 2nd level), physical Memory, and disk. The penalty for a cache miss (2nd level) is in the range of 30-200 clock cycles (100s ns) [56]. As long as the cache miss ratio falls into a consistent range throughout a process execution, it has little impact on the on-time performance of the soft RT processes. Therefore, we do not provide any cache management or guarantee. However, the penalty for a virtual memory (physical memory) miss is in the range of 700,000-6,000,000 clock cycles (10s of ms) [56]. For a software video decoder/encoder running at 30 frames per second (or $33ms$ per frame), a few virtual memory misses might lead to the loss of several frames.

In UNIX, each process has its own virtual address space. Within its virtual address space, a process memory is divided into several segments: text, stack, data, shared libraries, shared memory, and memory map. The text segment contains the program binaries. The stack segment contains the execution stack. The data segment contains the process data (e.g., `malloc()`).

Note that in C++, memory allocation for a new class object is done implicitly through the constructor call (e.g., new CLASSNAME). In such cases, the memory allocation does not go through our `Mem::alloc()` API call and hence it is not pinned.

7.5.2 Specific Issues about Communication Server

We have implemented our communication server in an integrated fashion with the underlying ATM network, CPU, and memory servers. The communication server runs on SPARC 10 machines. The SPARC 10 machines have FORE SBA-200E ATM adaptor cards, which are connected to a FORE ASX-200 switch. The switch is configured with 16 ports and with 155 Mbps capacity per port.

The bandwidth overhead of our METP is measured to be around 20%, which includes the ATM cell header overhead (8/53 Bytes), AAL MTU header overhead, and our transport layer PDU header. This means that if the application requests a connection with 10 Mbps user-level bandwidth[2], our communication broker will reserve a connection with 10 Mbps * 120% = 12 Mbps of mean bandwidth B_{mean} allocation. Furthermore, our implementation integrates the peak and burst bandwidth into one parameter, the peak bandwidth, because our ATM adaptor card does not support the B_{burst} parameter, whereas the ATM standard has specification for it. This means that in the above example, the peak bandwidth B_{peak} is set to be an additional 5 Mbps on top of the mean bandwidth (12 Mbps + 5 Mbps = 17 Mbps)[3]. The acknowledgment connection (reverse connection) is also established for sending acknowledgment information from the receiver back to the sender; its bandwidth is set to be one fifth of the forward connection's bandwidth.

7.5.3 Application Program Interface

The CPU and memory servers provide a C++ programming interface to the user processes. The methods are encapsulated into C++ classes called `CpuApi` and `MemApi`. The interprocess communication (IPC) between the resource broker/scheduler and the user processes is hidden inside the API. This API includes the following methods:

- Reservation and allocation of CPU and memory resources through the CPU and memory QoS parameters.

[2]This is an average bandwidth which considers an average size of the APDUs among the various frame sizes in the stream.

[3]The 5 Mbps corresponds to the overhead under the following assumptions: (1) all video frames transmitted over the connections are I frames ($B^I = M_A^I * R_A$), and (2) our METP protocol segments APDU into a set of TPDUs which may create a burst bandwidth over a short period of time, and this burst bandwidth may be larger than the mean bandwidth B_{mean}, or B^I.

- Management of CPU and memory resources.
- Modification and de-allocation of the allocated resources.

The communication server also provides a C++ programming interface to the user processes. Programs developed using the framework will need to communicate with the communication broker for service registration, connection establishment, and connection tear-down. The server API is encapsulated into the class `CommSvcApi`, and the client API is encapsulated into the class `CommCliApi`.

To illustrate the API described above in an integrated fashion, we provide a simple sample program below that shows how to use our CPU, memory, and communication servers. This sample program presents a piece of Video on Demand client code. For the purpose of illustration, we remove all parts dealing with error checking and resource rejection due to insufficient available resources.

```
/** Initiator side (client which displays video frames) **/
  ...
  MemApi mem;
  int rsvId = mem.reserve(500000);   // Memory reservation in number of bytes.
  mem.lockTxt(rsvId);
  char* displayBuf = mem.alloc(rsvId, 76800);  // buffer allocation for display
  char* decoderBuf = mem.alloc(rsvId, 20000);  // buffer allocation for decoder

  CommCliApi comm;
  Quality qos(APDU size=20000, endToEndDelay=300$ms$, sample rate=10 APDUs/sec);
  comm.open("server.cs.uiuc.edu", svcId, qos); // Communication reservation

  CpuApi cpu;
  CpuReserve cpuReservation;
  int period = 1000/framerate;

  // Probing Phase
  cpu.probe(period);
```

```
cpu.start();
for (int i=0; i < numProbeIterations; i++) {
  comm.recvData(decoderBuf, vsize);  // receive data
  decode(decoderBuf, displayBuf, vsize);  // decode data
  display(displayBuf); // display data
  cpu.yield(); // mark the end of one iteration
}
cpu.stop()
cpu.probeEnd();
cpu.probeMatch(&cpuReservation);

cpu.reserve(cpuReservation); // CPU Reservation Phase

// Execution Phase
cpu.start();
while () {
  comm.recvData(decoderBuf, vsize);
  decode(decoderBuf, displayBuf, vsize);
  display(displayBuf);
  cpu.yield();
}
cpu.stop();

// Cleanup Phase
cpu.free();  // free CPU reservation
mem.free(rsvId); // free Memory reservation
comm.close();  // free communication reservation
...
```

The testbed where our implementation and experiments are running consists of two Sparc 10 workstations under Solaris 2.5 operating system. They are connected via ATM Fore networks

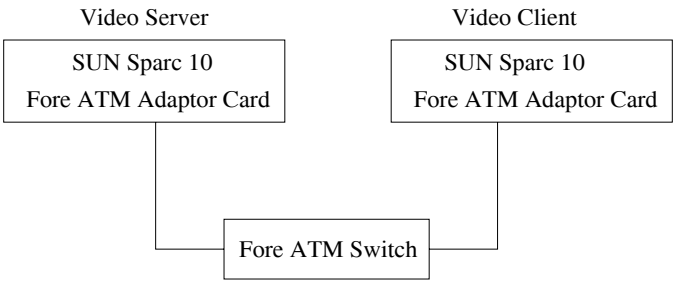


Figure 7.4: QualMan experimental setup.

as shown in Figure 7.4. The experiments are designed to show that with QualMan framework, end-to-end QoS requirements for bounded jitter, synchronization skew, and end-to-end delay for distributed multiple applications can be provided under additional load sharing of the resources such as CPU, memory, and network bandwidth.

7.6 Performance Evaluation

We have tested our CPU server with the communication and the memory servers. The network experiment uses two machines, one acting as a sender and the other as a receiver. The ATM network configuration is described in section 7.5.2. Except for the additional network support, the machines are of the same configuration as in section 7.5.3.

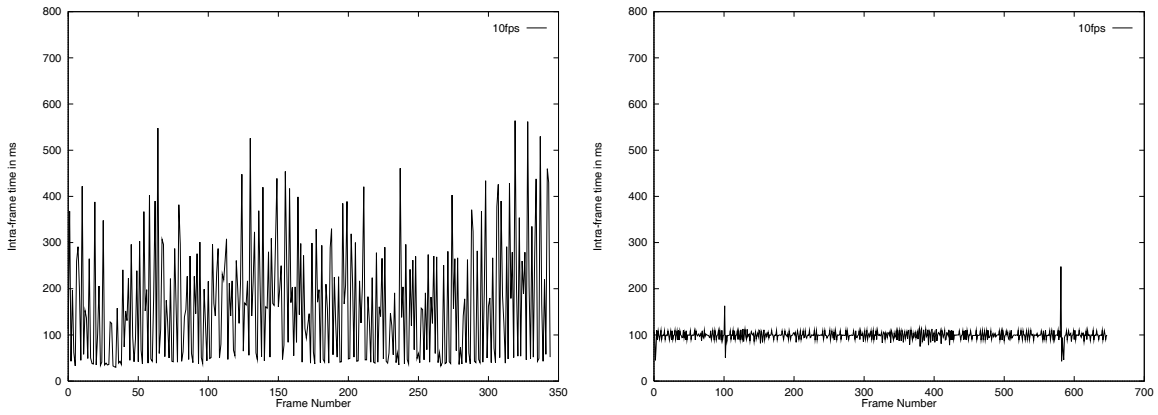
The communication server experiment runs a video server program on one machine and potentially several client video programs running on other machines. The video server program forks a child server process to service each client, and the server's child process retrieves a requested MPEG stream and sends the compressed video frames via METP protocol. The video client mpeg_play program is built on top of the Berkeley mpeg_play program, with modifications to read data from our RT transport protocol instead of a file. The client program mpeg_play performs the same decoding and displaying as in the original Berkeley mpeg_play program.

7.6.1 First Experiment

The RT mpeg_play server and client mpeg_play programs are played at 10 fps. They run concurrently with the following load mixture of background TS programs on both the server and the client machines:

- The gcc compiler compiles the Berkeley mpeg_play code.
- A compute program calculates *sin* and *cos* tables using the infinite series formula.
- A memory intensive program that copies mpeg frames in a ring of buffers.

Figure 7.5(a) has the client and server programs without any resource reservation. Figure 7.5(b) has the client program with reservation (CPU:PCPT class, Period=100ms, Peak Processing Time=80ms, memory=3MB, net=1Mbps) and the server program with reservation (CPU:PCPT class, Period=100ms, Peak Processing Time=40ms, memory=3MB, net=1Mbps). Without any resource reservation, noticeable jitter over 200ms occurs frequently (49 times). The largest jitter is about 450ms. With resource reservation, noticeable jitter over 200ms does not occur.



(a) (b)

Figure 7.5: Intra-frame time measurement for the client and server mpeg_play programs with and without CPU, memory, and network servers in QualMan.

7.6.2 Second Experiment

The second experiment consists of two RT concurrent mpeg_play clients and servers at 6fps and 3fps. Figure 7.6(a) has the client and server programs without any resource reservation. Figure 7.6(b) has the 6fps client with reservation (CPU:PCPT class, Period=166ms, Peak Processing Time=100ms, memory=3MB, net=0.6Mbps) and server with reservation (CPU:PCPT class, Period=166ms, Peak Processing Time=40ms, memory=3MB, net=0.6Mbps), and the

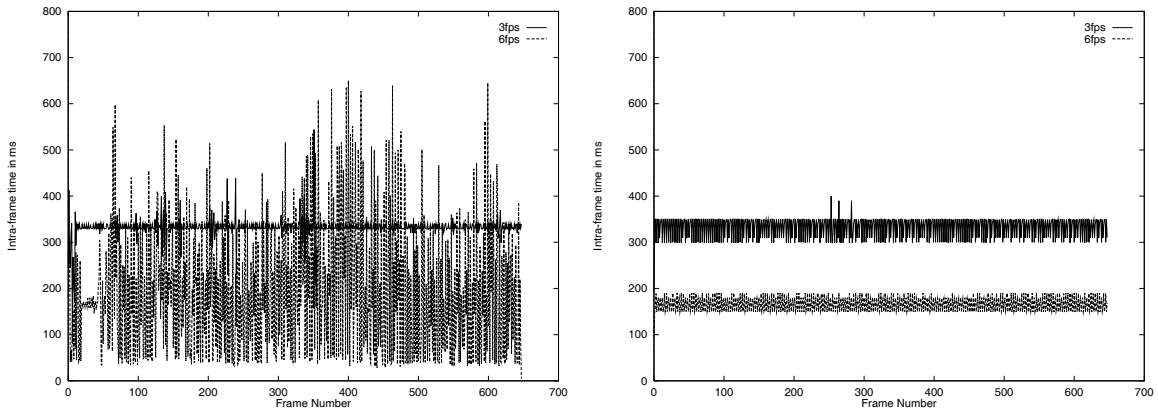


Figure 7.6: Intra-frame time measurement for the client and server mpeg_play programs with and without CPU, memory, and network servers in QualMan.

3fps client with reservation (CPU:PCPT class, Period=333ms, Peak Processing Time=100ms, memory=3MB, net=0.3Mbps) and server reservation at (CPU:PCPT class, Period=333ms, Peak Processing Time=40ms, memory=3MB, net=0.3Mbps). Without any resource reservation, noticeable jitter over 333ms for the 6fps client mpeg_play program occurs frequently (30 times), however jitter for the 3fps client mpeg_play program occurs less frequently because it consumes little resources at this low rate. With resource reservation, jitter stays within 20ms range for the 6fps client mpeg_play program and within 30ms range for the 3fps client mpeg_play program.

7.6.3 Third Experiment

Since the MPEG stream is compressed into low bandwidth, which is not a good stress test on our transport subsystem, we have performed additional experiments with the video server sending uncompressed video frames to potentially multiple clients at a much higher bandwidth using METP. Each uncompressed video frame is of fixed size 200KB. It consists of a single client program requesting video frames at 10fps (16Mbps) from a server program. The same mixture of TS background programs as described in the first experiment run concurrently with the video server and client programs on both the server and client machines. We measure the intra-frame time of the uncompressed video frame at the client side. Figure 7.7(a) has the client and

server programs without any resource reservation. Figure 7.7(b) has the client program with reservation (CPU:PCPT class, Period=100ms, Peak Processing Time=40ms, net=16Mbps) and the server program with reservation (CPU:PCPT class, Period=100ms, Peak Processing Time=30ms, net=16Mbps). Jitter over 100ms (one frame time) under no resource reservation occurs frequently (64 times), whereas it does not occur under the resource reservation.

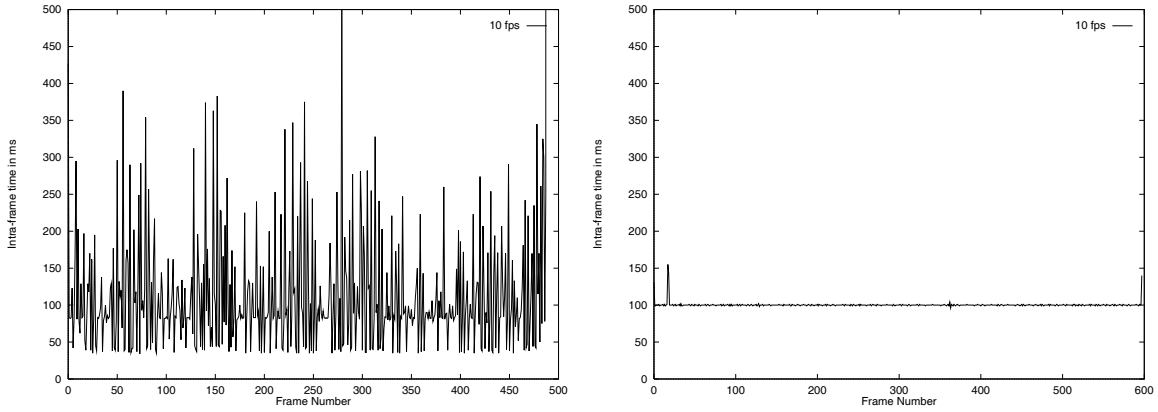


Figure 7.7: Intra-frame time measurement for the client and server with uncompressed video programs with and without resource reservation in QualMan.

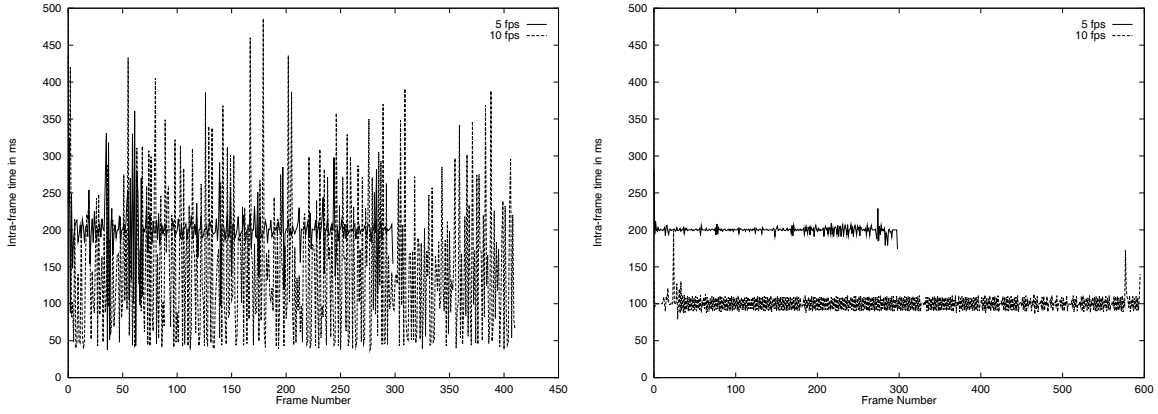


Figure 7.8: Intra-frame time measurement for the client and server uncompressed video programs with and without resource reservation in QualMan.

7.6.4 Fourth Experiment

The fourth experiment consists of two concurrent clients that request video frames at 10fps (16Mbps) and 5fps (8Mbps). Again the same mixture of TS background programs run concurrently with the video server and client programs on both the server and client machines. Figure 7.8(a) has the client and server programs without any resource reservation. Figure 7.8(b) has the 10fps client program with reservation (CPU:PCPT class, Period=100ms, Peak Processing Time=40ms, net=16Mbps), the 10fps server program with reservation (CPU:PCPT class, Period=100ms, Peak Processing Time=30ms, net=16Mbps), and the 5fps client program with reservation (CPU:PCPT class, Period=200ms, Peak Processing Time=40, net=8Mbps) and the 5fps server program with reservation (CPU:PCPT class, Period=200ms, Peak Processing Time=30ms, net=8Mbps). Noticeable jitter over 200ms (two frames times) for the 10fps client occurs frequently (35 times) under no resource reservation; whereas it does not occur under resource reservation.

Due to the limit of the processing power (CPU bandwidth) on the Sparc 10 machine, we cannot run as many concurrent MPEG streams as we would like. The bottleneck is in the software MPEG decoding which takes a significant amount of processing time. However, our solution is perfectly scalable to support multiple streams with a faster processor or with a hardware MPEG decoder.

7.7 Contributions

QualMan is a resource management platform which allows the RT distributed applications to specify QoS parameters in terms of CPU, memory, and communication QoS parameters. Through the specification, they can control the resource allocation according to their desired application level quality. In order to support this application level control, the resource management needs to be extended with brokerage and reservation capabilities. Our resource model for shared resources contains the resource broker, which provides negotiation, admission, and reservation capabilities over the shared resource, and the resource controller which allocates and schedule the resources to satisfy the quality specifications provided by applications.

We have showed through numerous experimental results that this integrated system layer architecture is feasible. The CPU, memory, and communication servers are implemented as

loadable middleware on a general purpose operating system and network platforms. Our results have shown that QualMan provides acceptable and desirable end-to-end QoS guarantees for various multimedia applications such as the MPEG player and video-on-demand application. Perceptually, it makes a huge difference in user acceptance if one watches the display of jitter-full video streams vs. smoothed streams.

Chapter 8

DSRT System Enhancement Services

In this chapter, we describe several enhancement services that are built on top of the DSRT and QualMan middleware for client applications as shown in Figure 8.1. These enhancements provide the DSRT system and its clients with additional services and capabilities such as *advance reservation*, *distributed reservation*, *security and access control*, and *remote monitor*. Section 8.1 describes advance reservation service, which allows clients to make advance CPU reservation for a specific time interval in the future. Section 8.2 describes distributed reservation, which allows distributed and collaborating clients to synchronize their CPU (or other resource types) reservations on multiple hosts (machines) across a network. Section 8.3 describes access

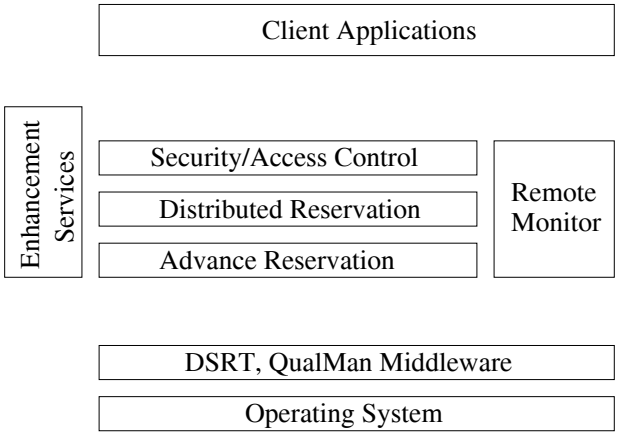


Figure 8.1: Enhancement services on top of the DSRT(QualMan) middleware.

control, which enhances the DSRT system with an access control and accounting mechanism on how much resources each user/client can acquire through reservations. Section 8.4 describes remote monitor, which provides a centralized database on resource availability information on remote/local hosts and usage pattern information of clients using the DSRT system over a distributed environment.

8.1 Advance Reservation

Advance reservation, designed in collaboration with Garimella and Nahrstedt [26], is an enhancement service that is built on top of the DSRT system. It allows a client to make an advance CPU reservation for a specific time interval in the future. There are many circumstances in which clients would want to make an advance reservations, reserving a level of QoS now that will take effect in the future. For example, a microscopist needs to use an expensive microscope, and he/she also needs guaranteed CPU allocation on a machine to run an application that processes data in real time from the microscope and provides control message back to the microscope. The microscopist is assigned a future time slot to use the microscope next week. In conjunction, he/she makes an advance reservation for CPU allocation on a machine. Some other examples include applications that use expensive instruments/parallel computers that cannot afford to stay idle, video conferencing, remote lectures, multi-player games, or multi-person visualization.

Advance reservation is implemented by the *advance reservation (AR) server*. Running concurrently with the DSRT server, the AR server wraps an additional layer of abstraction on top of DSRT server as shown in Figure 8.1. It also exports enhanced APIs which build on top of the programming interface provided by the DSRT system. The AR server performs the following tasks:

- It provides advance reservation APIs which extend the CPU service class specification in Table 2.1 with *start time* and *duration* in each service class. Advance reservations are distinguished from immediate reservations whose start time is immediate and duration (if not specified) is infinite.
- It performs admission control test on advance reservation requests, taking into account all other clients that would be active at the specified start time of the advance reservation

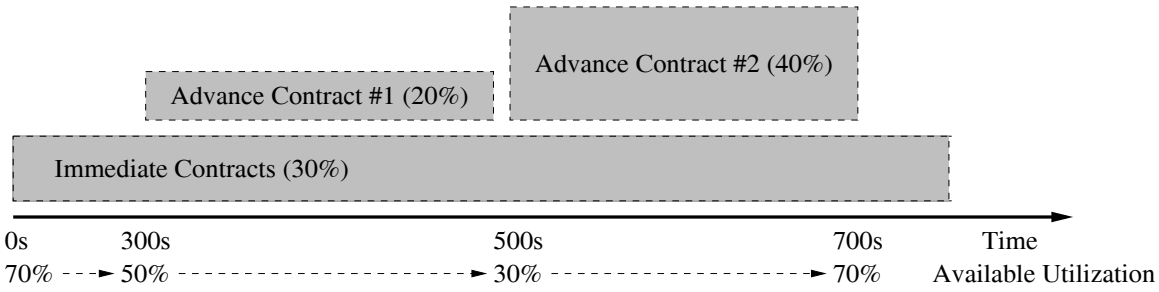


Figure 8.2: Reservation time graph.

requests. It also intercepts all immediate reservation requests and performs admission control test on them. It must ensure that resources guaranteed to advance reservations in the future will not be in conflict with resources guaranteed to immediate reservations.

- It maintains all information of accepted advance reservations, and it informs DSRT system when start time of an advance reservation becomes current or when its duration expires.

8.1.1 Admission Control Test

The core of the AR server is its admission control test. The AR server traps all immediate and advance reservation requests to the DSRT server, and it performs admission control test on behalf of the DSRT server. The admission control test accepts a reservation request only when it does not violate the guarantees on the existing immediate and advance contracts. To check for resource availability, it maintains a *reservation time graph* which shows the amount of available CPU capacity at various time intervals in the future. Utilization for each advance or immediate client is first computed using the same formulas as in the DSRT system described in section 3.2.

A sample reservation time graph is shown in Figure 8.2. It contains two advance contracts (*advance contract #1* starts from 300s to 500s with 20% utilization, *advance contract #2* starts from 500s to 700s with 40% utilization), and one or more immediate contract(s) whose sum of utilization equals 40%. Available capacity is calculated at time points on each contract's starting and ending time. Given a new advance reservation request ($startTime$, $endTime$, $utilization$), a linear search is performed to locate two existing time points in the reservation time graph: t_s is the time point immediately preceding $startTime$, and t_e is the time point

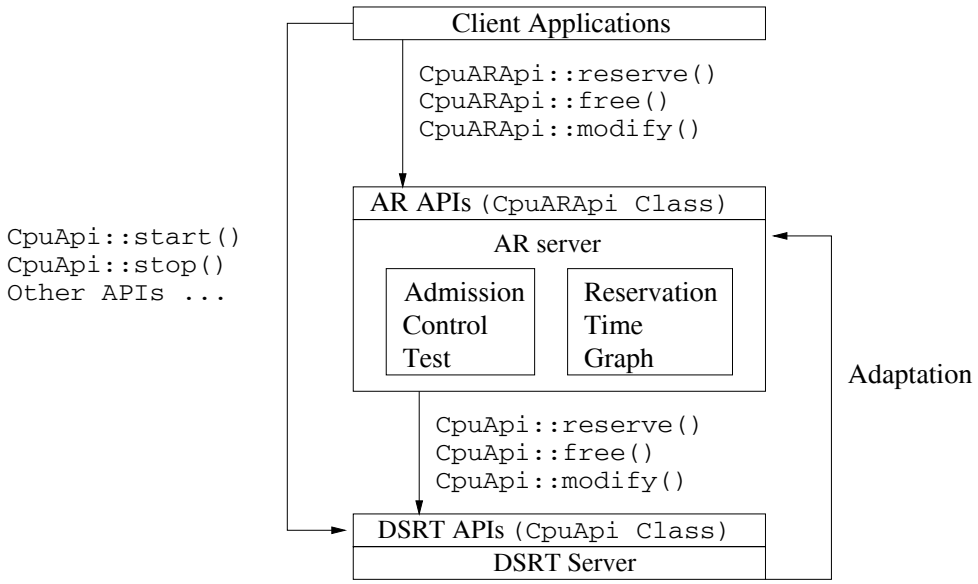


Figure 8.3: The AR server design.

immediately after $endTime$. The admission test condition is that for all the time points between t_s and t_e, the utilization of the new request must be less than or equal to the available capacity.

The reservation time graph needs to be updated when a new contract is admitted, an existing contract is freed, or an existing contract is modified because of a client-initiated negotiation or a server-initiated adaptation. A new admission of contract involves insertion of new time points in the reservation time graph at $startTime$ and $endTime$ of the contract, as well as deduction for the amount of allocated utilization for time points between $startTime$ and $endTime$. A release of contract involves deletion of time points on the reservation time graph at $startTime$ and $endTime$ of the contract, as well as replenishment for the amount of freed utilization on time points between $startTime$ and $endTime$. A modification of contract involves either deduction or replenishment on the corresponding time points.

When the AR server is running, resource allocation decision in the RT Partitions (e.g., admission control, re-negotiation of contract) is moved from DSRT server to the AR server. The DSRT server does not perform any duplicate admission control test; it simply trusts the resource allocation decision made by the AR server.

8.1.2 Design

The AR server design is shown in Figure 8.3. The AR server is a separate process. It can run as a RT process in order to provide a fast response time to clients, or it can run as a high priority TS process because the reservation phase does not need to be completed in real time. If it is run as a RT process, it can reserve a fixed amount of CPU time periodically from the DSRT server.

The AR server provides an API library encapsulated in a class called `CpuARApi` where clients can access AR services. `CpuARApi` is implemented as a derived class of `CpuApi` in the DSRT API library. The AR server traps all the reservation requests from the clients by overloading methods in CpuApi class. The `CpuApi::reserve()` method is overloaded by `CpuARApi::reserve()` method. When clients call `CpuARApi::reserve()` to make an advance reservation, it generates a message to be sent to the AR server instead of the DSRT server. Upon receiving the message, the AR server performs an admission control test, and updates the reservation time graph accordingly. If the AR server decides to accept a reservation request, it calls the DSRT server to enforce a contract establishment on behalf of the client using the DSRT API call `CpuApi::reserve()`. If the AR server decides to reject a reservation request, it replies to the client directly without going through DSRT server.

Methods in `CpuApi` class that affect the amount of available capacity in the system, namely `free()` (for releasing a contract) and `modify()` (for modifying a contract), are also overloaded in the CpuARApi class. These methods generate messages that go to the AR server so that the reservation time graph can be updated. Then AR server calls the DSRT server to enforce the release or modification of a contract on behalf of the clients through the DSRT API call `CpuApi::free()` or `CpuApi::modify()`.

Methods in `CpuApi` class that do not affect the amount of available capacity in the system, e.g., `start()`, `stop()`, .., are not overloaded in the `CpuARApi` class. These methods generate messages that go to the DSRT server directly as if when the AR server is not running.

For an advance reservation/contract, the AR server will wait until the start time of the contract, and calls the DSRT server to enforce a contract establishment on behalf of the client. When the duration of the contract expires, the AR server calls the DSRT server to enforce the release of contract on behalf of the client if the contract has not been released by the client.

The DSRT server may modify a contract through a specified system-initiated adaptation algorithm when the usage pattern of a client changes. When it occurs, the DSRT server sends an asynchronous message to the AR server with the amount of utilization that needs to be modified in a contract. The AR server would periodically poll the message queue for any new messages from the DSRT server. If a message is found, the AR server uses the information in the message to perform an admission control test based on the reservation time graph for the contract in question. If the modification can be accepted, the AR server calls `CpuApi:modify()` to make the DSRT server enforce the modification. If the modification can not be accepted, the AR server simply ignores the message.

8.1.3 Implementation

The advance reservation service has been implemented on top of Solaris and Irix operating systems. It has also been integrated and runs in conjunction with the DSRT system. Interested reader can find the implementation detail and experimental results in [26].

8.1.4 Related Work

There are several related works on advance reservations in the network domain. However, the concept of advance reservations is relatively new in the realm of CPU scheduling.

Berson, *ed.al.* [3], proposes advance reservations that augment RSVP [5]. It describes a server-based architecture where no reservation state or scheduling state needs to be set up in the routers until the reservation is established. It also allows applications to request advance reservations without the application running during the length of the advance reservation. The AR server is similar to Berson's server where it maintains advance reservations, and activates them at their start time.

Schelen, *ed.al.* [59], describes an architecture in which clients in a networked environment can make end-to-end resource reservations through agents. For each domain in the network, there is an agent responsible for immediate and advance admission control. The admission control algorithm involves a parameter called the *look ahead time* which is the point at which resources are actually made available to approaching advance reservations by rejecting immediate requests. When resources cannot be made available through rejection of immediate requests, selective preemption of immediate reservations is performed. The AR server differs

from Schelen's agent such that there is no lookahead time. Our admission control test provides guarantees to both advance and immediate contracts.

GARA [22], Globus Architecture for Reservation and Allocation, also supports advance reservation. Globus is also an application of the DSRT system, and we describe it in detail in section 9.1.

8.2 Distributed Reservation

Distributed reservation, designed in collaboration with Gupta [33] and Nahrstedt, is an enhancement service that is built on top of DSRT system. This service is target toward distributed multimedia applications that involve resources spread across a network or even multiple networks. In order to achieve an acceptable quality, these applications require their reservations on multiple hosts(machines) to be interdependent and collaborative. Some examples are ScoreGraph [2], CAVE [16], and other virtual reality applications. A typical scenario involves a few hosts generating visual and auditory data, another set of hosts working with the computational models, and yet another set tracking user input. Performance degradation on one host can adversely affect the complete distributed application.

Distributed reservation achieves the following goals:

- It allows a distributed application to reserve resources for its constituent clients on their respective end systems.
- It allows the distributed application to fine tune the reservations depending upon its changing requirements.

8.2.1 Design

The design for the distributed resource management is shown in Figure 8.4. We give a brief description on these four components as follows.

- A *Resource Agent* is installed on each host machine. It provides a reservation manager and a simple unified interface to the underlying resource servers (not limited to the DSRT CPU server). Resource agents on different hosts can communicate with each other. They can reserve different types of resources on their local hosts through its local resource

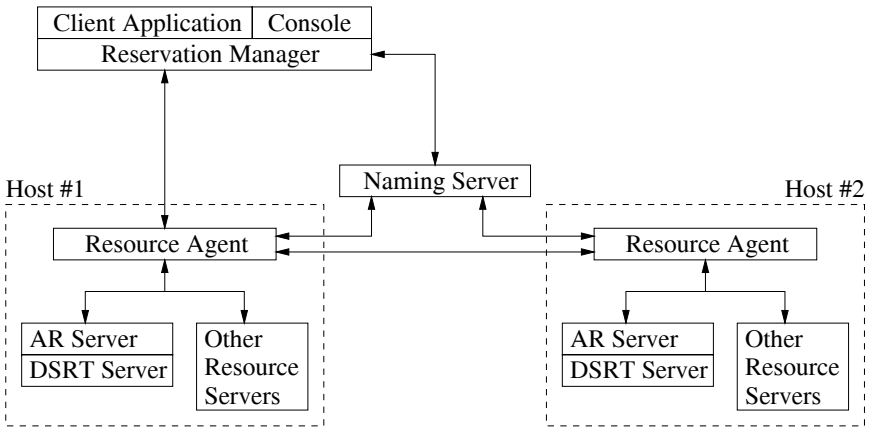


Figure 8.4: Distributed reservation resource management design.

servers(brokers), or on a remote host through a remote resource agent. This enables the reservations for local/remote processes of a distributed application to be managed from one single host. The DSRT server is used as the CPU resource server. A resource agent can also make an advance reservation through the AR server described in section 8.1.

- A *Reservation Manager* helps an application manage and control reservations of all its constituent processes. It hides from the client application programmer the detail of communication with resource agents and provides a further simplified API.

- A *Naming Server* is included which resource agents are able to locate each other on the network. The address of the naming server is known *a-priori* to each resource agent. Each resource agent registers with the naming server its name and its associated address in a known format. Then it can obtain addresses of other resource agents by querying the naming server. The applications or the reservation manager can also query the naming server to locate resource agents on a particular host.

- A *Console* is a graphical user interface representing a reservation manager. It allows the users to view and to control a distributed reservation in progress.

The intelligence in this distributed resource management is in the resource agent and reservation manager. We describe them in further detail in the following sections.

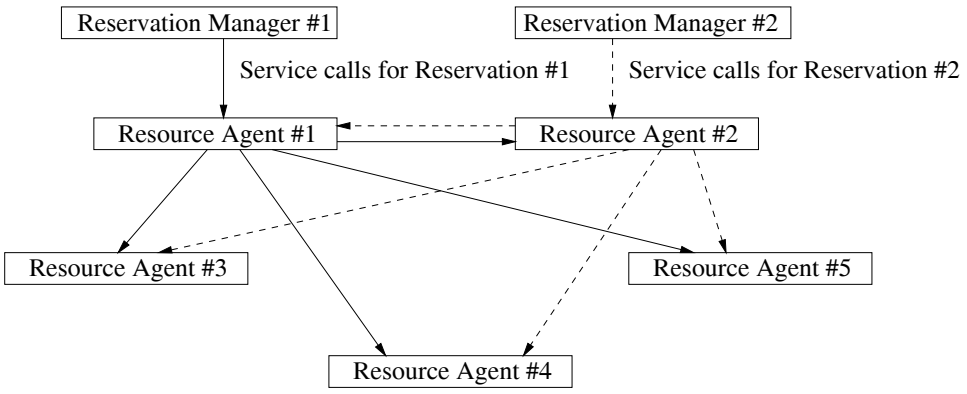


Figure 8.5: Configuration of resource agent network.

8.2.2 Resource Agent

The network of resource agents can be setup in two possible ways:

- Master-slave: One resource agent is designated as the master, and all other resource agents on the network are its slaves. All reservation requests originate only from the master. Only the master resource agent has complete control and reservation information in the network. The slave resource agents merely assist the master in providing local reservations to applications. This is an absolutely centralized approach.

- Peers: All resource agents are peers. They have equal privileges, similar information, and control. All can accept reservation requests, and process them with help from other resource agents. A reservation made with one resource agent can be modified through another. A single application may communicate with multiple resource agents directly. A typical implementation of this totally distributed setup requires maintaining a lot of redundant information among resource agents.

Our resource agents setup is a combination of the two approaches. A reservation made through a particular resource agent can be modified and released only through that resource agent. At the same time, all resource agents have equal privilege to originate requests for resources anywhere on the network. Figure 8.5 shows the reservation calls made for two distributed applications on different hosts. reservation manager #1 on host #1 manages reservation #1 for an application, and resource agent #1 acts as #1's master who makes distributed reservations involving #1's remote hosts. Reservation manager #2 on host #2 manages reser-

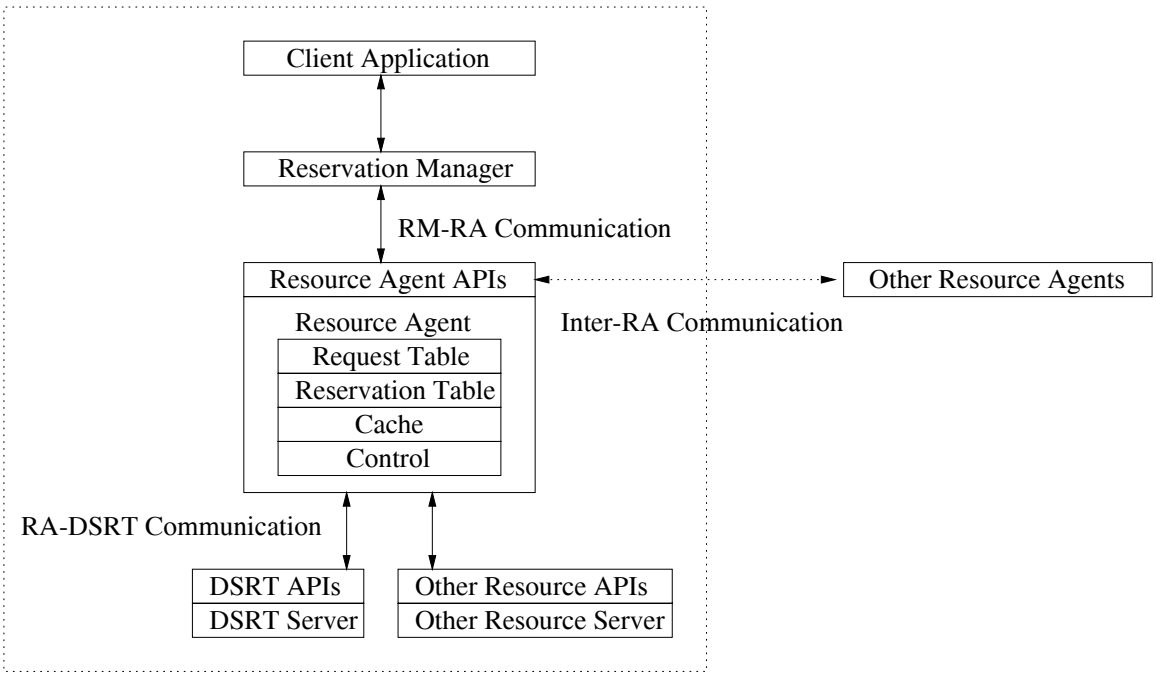


Figure 8.6: Resource agent.

vation #2 for another application, and resource agent #2 acts as #2's master who makes distributed reservations involving #2's remote hosts.

A resource agent contains several components internally as shown in Figure 8.6.

- *RA-DSRT Communication:*

  A resource agent (RA) makes a local reservation on the DSRT server through the RA-DSRT communication channel, which is simply the APIs provided by the DSRT CPU server. This design is extendible to include other types of resource servers (e.g., memory server, disk server, ...).

- *RM-RA Communication:*

  A reservation manager (RM) sends a reservation request to its local resource agent (RA) through the RM-RA communication channel. This is implemented as APIs provided by the resource agent.

- *Inter-RA Communication:*

  A resource agent can send a reservation request to a remote resource agent through the

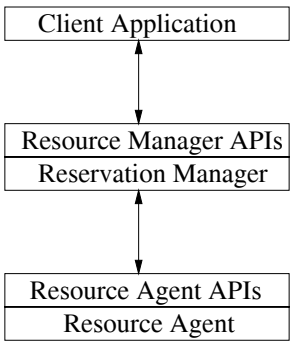


Figure 8.7: Reservation manager.

inter-RA communication channel. This is implemented as the same APIs provided by the resource agent.

- *Reservation Table:*

  A resource agent generates a unique reservation identifier associated with every process it has made a reservation at its local server. The master resource agent can later use this identifier to request modifications on the reservation. A resource agent maintains a list of all valid identifiers along with the corresponding process ids in the reservation table.

- *Request Table:*

  A resource agent also generates a unique request identifier associated with a multi-resource reservation. A multi-resource reservation can contain several reservations for CPU, memory, or disk resources on a single host.

- *Cache:*

  Addresses to other resource agents, whom has been contacted recently, are cached locally. This helps reduce the number of queries to the naming server.

- *Control:*

  A resource agent has logic for enforcing resource reservation constraints among multiple resources and multiple hosts. The various distributed reservation policies are described in section 8.2.5

8.2.3 Reservation Manager

A reservation manager contains the following components shown in Figure 8.7. It provides a set of APIs that form another level of abstraction around the resource agent. One reservation manager can be used to manage all resource requests from one client application.

Upon initialization, a reservation manager first locates the resource agent on its local host by querying the naming server whose address is known a-prior. It can accept local or remote reservation requests from client applications. A client can add new reservation requests, can modify an existing contract, can release an existing contract, and so on. A reservation manager maintains a list of all local and remote processes and their resource requests that were submitted by the client.

8.2.4 Distributed Reservation Protocol

A distributed reservation protocol is designed to coordinate all the components to successfully complete a distributed reservation. It contains four basic steps. A bootstrap process is used to start all local and remote processes of a distributed application, called a *process set*. The bootstrap process does not need any reservation for itself. It manages the synchronization of local/remote processes in its process set, and runs the distributed reservation protocol. It uses the APIs provided by the reservation manager. Note that the Console is a generic bootstrap process.

- *Step 1.* The bootstrap process first initializes a reservation manager, and submits a set of resource requests. Each request corresponds to a local/remote process in the process set. It contains the following information: resource requirements of this process, host name (local or remote) to execute this process, and the executable file for the process. The bootstrap process also submits a unique request identifier, e.g. hostname:bootstrapPid, which can later be used to identify all the reservations associated with this application. One of the reservation policies described in 8.2.5 can be specified for this application.

- *Step 2.* The reservation manager forwards the resource requests to its local resource agent, which becomes a *master* for these requests. The master resource agent calls the naming server to find references to all remote resource agents on hosts that are involved in this application. These remote resource agents are slaves to the master resource agent.

The master resource agent sends execution commands to slave resource agents to create application processes. If all processes in the process set are created successfully, the master resource agent starts to make reservations in step 3; otherwise, it returns failure to reservation manager and terminates.

- *Step 3.* The master resource agent sends slave (remote) resource agents with resource requirements of the created process, which in turn calls its local resource server (the DSRT server) to make a reservation at host machines. If the DSRT server rejects the reservation request due to insufficient resource availability, appropriate action is taken depending on the chosen reservation policy described in section 8.2.5.

- *Step 4.* The reservation manager collects reservation identifiers of local/remote processes from all resource agents. The bootstrap process or the reservation manager can send control messages to add a new process with new reservation to the process set, or to modify/free the reservations in the process set.

8.2.5 Reservation Policies

We propose the following reservation policies when our distributed resource management protocol cannot satisfy all the resource requests by a distributed application.

- *All-or-none Policy:*

  Some critical applications, like remote surgery, need to run with absolute guarantees on all resources they request. Partial reservations are of no use to them. The all-or-none policy acquires either all requested resources or nothing. The distributed reservation protocol returns failure to its client as soon as it finds that one request cannot be satisfied, and frees all the acquired reservations.

- *Best-effort Policy:*

  An alternative set of not-so-critical applications can do with just partial satisfaction of its resource requests. Video conferencing is such an example. It is not critical for all participants to get guaranteed picture quality. Hence, the distributed reservation protocol will try to satisfy as many requests as possible.

- *Persistent Best-effort Policy:*

  Sometimes it is desirable that an application begins with whatever minimum amount of resources that can be acquired at the start time. When more resources become available later, the application can acquire them during runtime. The emphasis for this policy is that the application be started at a specified time at the expense of lower quality.

- *Timeout Policy:*

  The previous three policies make just one reservation attempt per request before returning success or failure. This can be changed by specifying a timeout duration when the distributed reservation protocol can repeat its attempt to acquire resources.

8.2.6 Implementation

The distributed reservation service has been implemented and tested on Irix operating system. It has also been integrated and runs in conjunction with the DSRT system. APIs for the resource agent and reservation agent are too numerous to be described in this thesis. Interested reader can find them and experimental results in [33].

8.2.7 Related Work

Livny, *ed.al.* [46] presents a resource management system (RMS) in a high throughput computing (HTC) environment. RMS manages a pool of distributively owned resources. Client applications and resources are brought together by a matchmaking service. Applications make requests for resources to the matchmaking service which matches these with the advertisements of available resources. Resource owners can specify constraints on the kinds of jobs that may use the resource, and client applications can place restrictions on their resource requests. The Condor Project [8] is an implementation of this paradigm. Our distributed reservation service differs from the Condor Project in that we provide resource reservation and guarantees to distributed applications, whereas Condor is focused on making full use of the available distributed resources in the HTC environment.

Legion [10] is a model for a worldwide virtual computer. This computer system will support a large number of hosts and objects, geographically separated but connected with high-speed network links. Its resources will be owned and maintained by various voluntary organizations.

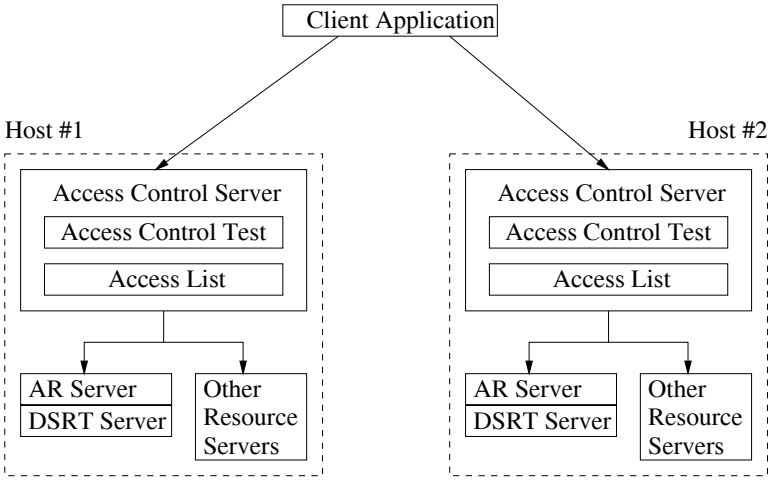


Figure 8.8: Access control design.

It also contains a reservation based scheduling scheme. The reservations are made through a negotiation process between resource providers and resource consumers. The final authority over the use of a resource lies with the resource and its owner. Legion's goal is to provide a virtual computer using distributed resources while our focus is to provide QoS guarantees to distributed applications. Despite the differing goals, we share a common philosophy of distributed resource reservation.

8.3 Access Control

Access control, designed in collaboration with Kim [39] and Nahrstedt, is an enhancement service that is built on top of the DSRT system. This service is target toward resource owners who want to dictate how much resources each authorized user can acquire through immediate and advance reservations. For example, a resource owner may want to place an upper limit on the aggregate amount of resources that applications started by each authorized user. A *pricing model* can also be built on top of the access control to provide revenues for resource owners who can charge computing cycles to the clients.

8.3.1 Design

The design for the access control is shown in Figure 8.8. Each host contains an access control (AC) server that performs an *access control test*, which decides whether to accept an incoming

reservation request. If the reservation request is accepted, the AC server forwards it to its resource servers for reservation. Otherwise, the AC server replies access denial to client application. Note that the access control test in the AC server is performed before the admission control test in the resource server (e.g., the DSRT server).

The access control test is based on an *access list*, which is specified by the resource owner on different hosts. The access list contains limits on resources (e.g., in CPU, memory, or network) that each authorized user can acquire through immediate or advance reservations. A sample access list is shown in Table 8.1. It is possible that access control test succeeds but admission control test fails.

userid	CPU	Memory	Network
owner	100%	100%	100%
John	50%	50%	50%
Alex	30%	20%	10%
Allison	40%	50%	60%
Naoko	25%	25%	25%

Table 8.1: A sample access list.

The access control test checks if a reservation request violates the limit of the requesting user. It maintains an *utilization counter* for each authorized user, which counts the aggregate amount resource utilization from all his/her applications on that host. The AC server needs to be notified from the underlying resource servers (or the AR servers) whenever a reservation request is accepted, a contract is freed, or modified. For advance reservations, we have two possible accounting policies.

- *All-Time Policy:*

  It treats an advance reservation as an immediate reservation. It sums up the utilization of immediate and advance reservations, regardless of their start time and end time, and ensures that it is below the resource limit.

- *Any-Time Policy:*

  It sums up the utilization of concurrent immediate and advance reservations at each point of time, and ensures that it is below the resource limit. It can make use of a reservation time graph shown in Figure 8.2.

8.3.2 Implementation

The access control service has been partially implemented on Solaris operating system. It has not been integrated with the DSRT system.

8.4 Remote Monitor

Remote monitor, designed in collaboration with Geisler [27] and Nahrstedt, is an enhancement service that is built on top of DSRT system. This service is target toward distributed applications or their users who need to find out the amount of available CPU resources on local/remote host machines. With the resource availability information, they can direct their reservation requests to hosts that have enough available capacity. A *resource discovery protocol* can be implemented on top of a remote monitor that matches reservation requests to hosts.

Remote monitoring service achieves the following goals:

- It provides resource availability information on local/remote hosts.
- It provides a distributed application with information about its distributed reservations and processor usage pattern of its processor set[1].

8.4.1 Design

The design for the remote monitor is shown in Figure 8.9. We describe various components as follows:

- *CPU Reporter:*

  It is created by a DSRT server on each host. It periodically acquires the resource availability information and the process information from the DSRT server. Then it feeds the information to a remote LDAP Server whose location is known *a-priori*. The periodicity (or frequency) is an adjustable and tunable parameter. Higher(lower) frequency means more(less) accurate information but higher(lower) network and CPU bandwidth overhead.

- A *LDAP (Lightweight Directory directory) Server:*

  It [69] stores the availability and process information from all CPU reporters on remote

[1] A processor set of a distributed application contains all its local/remote processes.

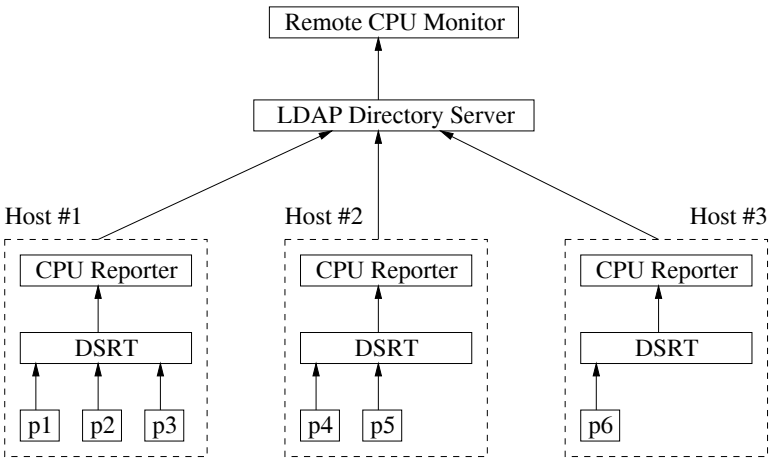


Figure 8.9: Remote monitor design.

hosts. CPU reporters are configured to talk to a customized LDAP Server for Globus, called Metacomputing Directory (MDS) server.

- A *Remote CPU Monitor:*

  It can query the LDAP server to find out the availability and process information, and then presents them in a graphical user interface or a visualization tool.

8.4.2 Implementation

The remote monitor has been implemented and tested on Solaris operating system. It has also been integrated and runs in conjunction with the DSRT system and the Globus project. Interested reader can find further detail on implementation and APIs in [27].

Chapter 9

Applications

We describe two important applications in which the DSRT system is used: the Globus Project at Argonne National Lab, and the Catalase Project at Beckman Institute Image Technology Group.

9.1 Globus Project

The Globus project [23], at Argonne National Lab, is an ongoing research project that utilizes large number of connected, heterogeneous, and high performance computers available over the network to form virtual supercomputers for distributed high performance computing applications. The Globus architecture is shown in Figure 9.1. We describe each of the Globus components as follows:

- *Resource Broker:*
  A client application first describes its resources request to a Globus resource broker using a language called *resource specification language (RSL)*. The RSL request can be in the form of the number of computers, the number of processors on each computer, the amount of memory, etc. The resource broker queries the Globus *Metacomputing Directory Server (MDS)* to locate the hosts that have the matching resource availability. This is a *resource discovery protocol*. The resource broker transforms a RSL request into a more concrete language called *Ground RSL*, which contains locations of the hosts that will service this request.

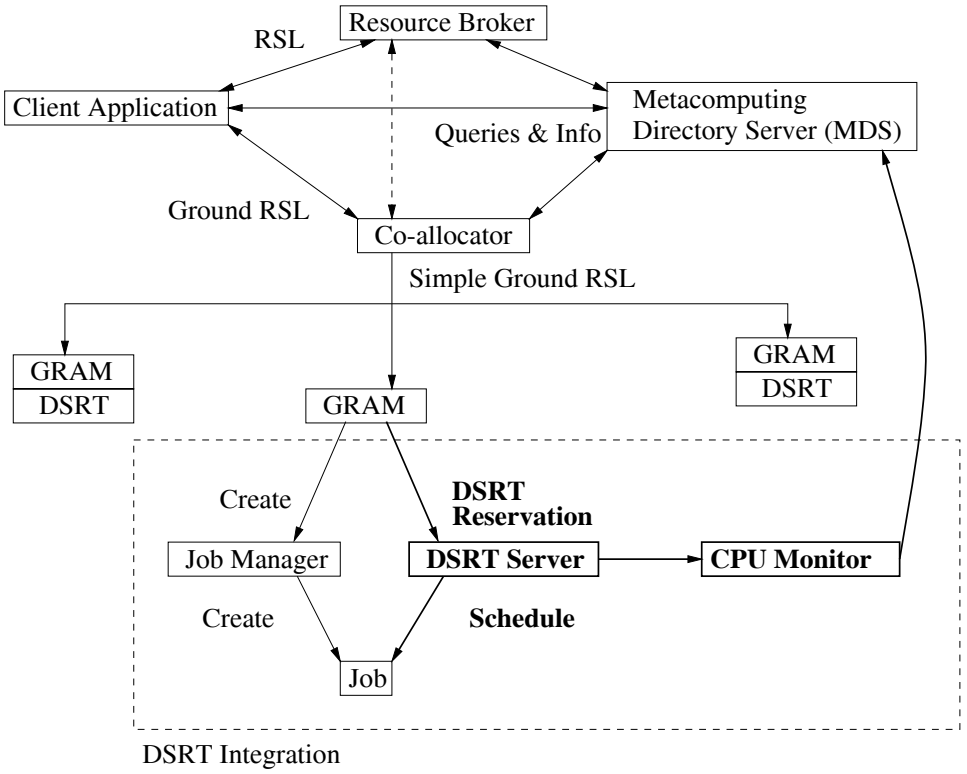


Figure 9.1: Globus architecture and the DSRT system integration.

- *Metacomputing Directory Server (MDS):*

  It stores and makes availability information accessible about computing resources in this virtual supercomputer. The information includes the architecture type, operating system version, amount of memory, network bandwidth, availability, and system loads.

- *Co-allocator:*

  The co-allocator receives the resource request in the ground RSL language. It further transforms it into pieces of *Simple Ground RSL*, and distributes one piece for each *Globus Resource Allocation Manager (GRAM)*, which manages and controls the resource of one host machine or a set of machines.

- *GRAM:*

  Upon receiving the request, GRAM creates a job manager that executes the client's job on its local host(s). Given that many jobs may run concurrently, GRAM employs different

Local Resource Allocation Manager (LRAM) to make sure the each request is satisfied at runtime. One possible LRAM is the DSRT system; its integration into Globus is described later.

- *Heartbeat Monitor:*

  It periodically monitors the health and status of a distributed set of processes in a client application. The client application can monitor the progress of its remote processes, or it can receive notifications on any failures and then attempts a recovery. It is similar to the reservation manager in the distributed reservation enhancement of the DSRT system.

9.1.1 DSRT Integration into Globus

The DSRT system and its enhancement services compliment the Globus project in many aspects. The DSRT server can act as a local resource allocation manager (LRAM) inside the GRAM, providing both immediate and advance reservations, real time scheduling, processor time guarantees, adaptation, and other services. The DSRT system's middleware implementation also works well with the heterogeneous environment in the Globus project where it is not possible to modify the operating systems on distributed owned host servers. The DSRT system integration into Globus is shown in Figure 9.1.

- *Extended RSL:*

  The RSL language is extended to include the CPU service class specification. The extended RSL allows a client application to precisely state the fraction of processor resource that each of its processes needs.

- *DSRT server:*

  It is added as a LRAM component inside the GRAM and on each machine that supports the extended RSL described above. Advance or immediate reservation requests are first forwarded to the DSRT server, which performs an admission control test. After the admission control test succeeds, the job manager executes the client's job using the accepted reservation.

- *CPU Monitor:*

  This DSRT remote monitor enhancement is added as another component inside the

GRAM wherever the DSRT server is running. It periodically monitors the amount of available processor on the local host and updates the Globus MDS.

Readers can read more about the DSRT integration with Globus project in Globus Architecture for Reservation and Allocation (GARA) [22].

9.2 Catalase Project

The Catalase Project [57], at Beckman Institute Image Technology Group, is aimed to automatically acquire high quality images from an electron microscope. The automated image acquisition is both computationally and data intensive. As a result, it requires distributed computing resources. However, not only does the system require distributed computing resources, but it also needs them in a dynamic and guaranteed fashion. Dynamic requirement for the computing and communication comes because various processing events depend on the captured knowledge about images. The requirement for quality of service guarantees for computing and communication comes because, when the application needs the distributed resources, they must be allocated in a timely fashion, so that proper control of media and devices can occur.

The control flow in the Catalase Project is shown in Figure 9.2. The scope/camera server is a dedicated machine that acquires images from the scope and camera. There are several phases. First, the scope/camera server acquires a low resolution image and sends it to an image processing server(s), which is a shared powerful server machine. Second, the image processing server analyzes the acquire images, finds the potential high resolution spot for further zooming, and returns the new location back to the scope/camera server. Third, the scope/camera server moves the scope/camera to the high resolution location, acquires a high resolution image, and sends it to the image processing server. Forth, the image processing server measures the image quality, and executes a learning algorithm for finding the high resolution spot. Last, it sends the high resolution images to the client station for display.

Both the camera and scope time are expensive. In order to keep them busy, the image processing on the shared server must be scheduled as soon as images arrives. We employ the DSRT system to reserve processing time and guarantee the image processing processes. However, one complexity in this application is that the amount of processing time depends on its phase. For example, phase 2 requires less processing time than phase 4. As a result, we

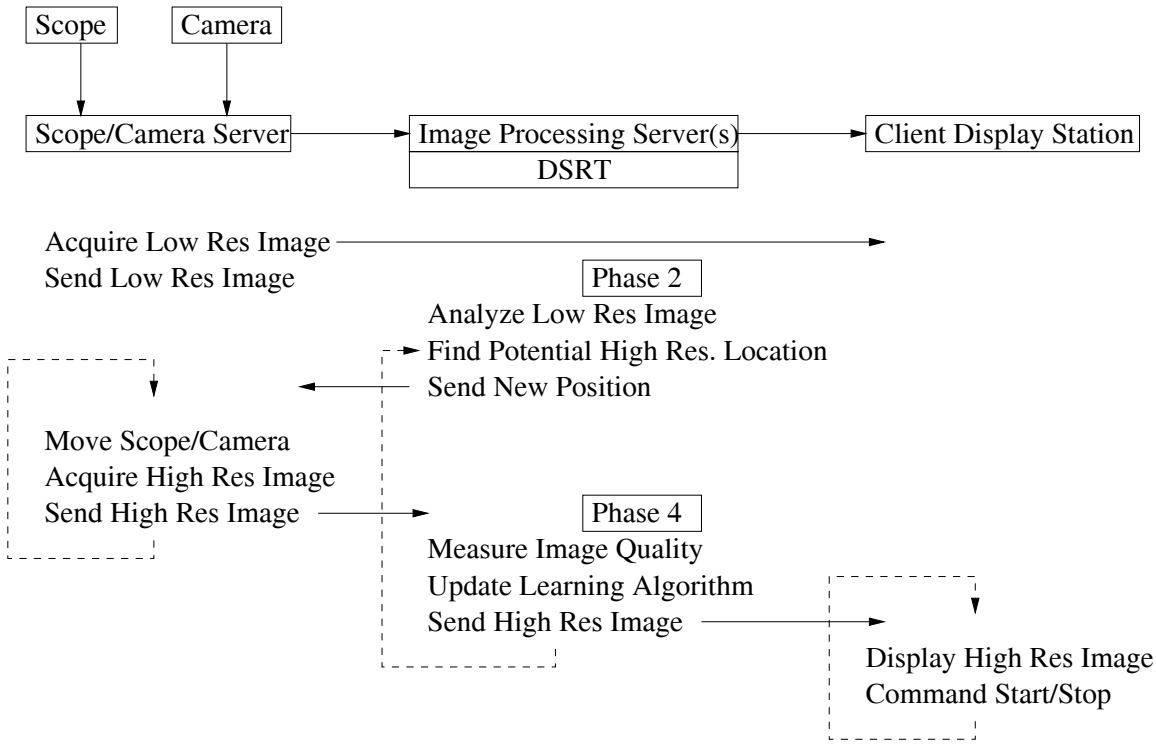


Figure 9.2: Control flow in the Catalase project.

expand the CPU Service Class specification to accommodate different processor usage patterns in different phases of the same client process.

	Phase 1	Phase 2
CPU Service Specification	PCPT Class $Period = 800ms$ $PPT = 400ms$	PCPT Class $Period = 1200ms$ $PPT = 900ms$

Table 9.1: A two phase reservation.

Table 9.1 shows a two-phase reservation. The first phase is a PCPT class reservation with length of $800ms$ and utilization of 50%. The second phase is a PCPT class reservation with length $700ms$ and higher utilization of 75%. This means that this process executes repeatedly phase 1, phase 2, phase 1, phase 2, ... Note that it is possible for phase 1 and phase 2 to be in different CPU service classes.

Chapter 10

Conclusion

10.1 Summary

This thesis contains three major parts: the DSRT system, QualMan, and the enhancements to the DSRT system.

The DSRT system has been designed and implemented to support both soft real time multimedia applications (appliances) and traditional time sharing applications simultaneously in a general purpose shared server environment. The design of the DSRT system is centered around three innovative concepts. The *CPU Service Classes* specification allows different SRT clients to describe their CPU usage patterns to the DSRT server. The DSRT server maps the CPU Service Classes into *Multiprocessor Partitions* in order to provide processor time guarantees to clients' contracts. The SRT client is programmed according to *Three Phases Execution Flow*, which includes a *probing phase* when reservation parameters are extracted, a *reservation phase* when a CPU reservation(contract) is negotiated, and an *execution phase* when real time execution occurs and the contract can be automatically adjusted according to a *system-initiated adaptation strategy*.

The DSRT system has been enhanced with additional capabilities and services to the clients. *Advance reservation* allows SRT clients to make advance CPU reservations for a specific time interval in the future. *Distributed reservation* allows distributed and collaborating clients to synchronize their CPU reservations (or other resource types) on multiple hosts across a network. *Access control* enhances the DSRT system with an access control and accounting mechanism

on how much resources each authorized user can acquire through reservations. *Remote monitor* provides a centralized information database about resource availability on remote/local hosts and processor usage pattern of clients using the DSRT system over a distributed environment.

QualMan is a multi-resource QoS-aware management system which considers CPU, memory and network resources. QualMan is designed for the same general purpose shared server/network environment as the DSRT system. QualMan applies a general and unified *resource specification model* and *resource control model* for different resource types it manages.

10.2 Contributions

This thesis makes the following contributions:

- The DSRT system stands out among the related systems by introducing the *CPU service classes* in the processor domain. The *three phase execution flow* is innovative, and it allows developers to easily write new or convert existing general purpose applications to run in the reservation-based DSRT system. To accommodate dynamic processor usage behavior in client applications, the DSRT system provides a selection of *system-initiated adaptation strategies*, which requires only a very simple API call and no work on the part of the client applications. We have shown that the DSRT system achieves acceptable overhead while providing good guarantees to SRT processes using a user-space middleware implementation without any kernel modifications. Our experimental results support this claim.

- The DSRT system is further enhanced by additional capabilities and services to the clients. The enhancements include *advance reservation*, *distributed reservation*, *access control*, and *remote monitor*.

- We have done a comprehensive literature survey and found 16 systems related to the DSRT system. Through comparisons, we have concluded that the DSRT system is innovative in its concepts and design.

- We have made a widely-used software release for the DSRT system software, and it is used by numerous research groups all over the world. The DSRT system has been integrated into the Globus Project and applied to the Catalase Project. We also list a few of the

registered groups or users: GRASP lab at University of Pennsylvania, Electrotechnical Laboratory in Tsukuba Science City, Japan, Computer Systems Lab in Korea University, BITS in Pilani, India, Indian Institute of Technology at Madras, Iowa University, University of Izmir, Korea Institute of Science and Technology, University of Texas at Arlington, ...

The significance of our work will be especially visible in the future when the computing paradigm shifts to shared servers and thin clients environment. We envision that the networks and general purpose processor speed will improve to a point where intensive computational tasks within applications will be performed at remote shared general purpose multiprocessor servers delivering results to thin clients in real time. These applications and appliances (e.g., multimedia, high performance, distributed computing) demand guaranteed processor time allocation in order to deliver real-time quality of services to the clients. This will require deployment of QoS-aware dynamic soft real-time scheduling paradigm such as the DSRT system and QualMan.

10.3 Future Work

We discuss two future research areas that can further expand the capabilities of the DSRT system.

10.3.1 Pricing Model

The pricing model is a complement to any reservation-based systems. Unlike a fair-sharing best-effort system, a client in a reservation-based system can monopolize the resources forever with advance and immediate reservations. When multiple clients compete for limited resources, a good metric to decide which clients' reservations to accept is *price*. Pricing is one possible solution to the *fairness issue* in a reservation-based system.

A *pricing model* also allows the resource provider to generate the necessary revenues to offset the cost of buying and maintaining the resources. At the same time, it also allows clients to get *guaranteed* service through reservations. There are many open research issues on how the resource providers should charge for immediate and advance reservations, and at *what price* clients should pay for *what quality*. There has been much research work that focuses on them [65]. These issues become more interesting and complex if we consider that there are multiple

competing resource providers available over the network, and how the competition affects the pricing model.

10.3.2 Resource Discovery Server

A resource discovery server takes a reservation request from a client and tries to locate a shared server over a network with a matching resource availability. A resource discovery server usually contains a *database* (e.g., MDS described in section 8.4) which contains the location of the servers (IP address), the hardware/software configuration on the servers (e.g., the amount of memory, processor type, number of processor, operating system), and resource availability information (e.g., 30% available capacity). The resource availability information is updated periodically through a remote monitor (described in section 8.4). When it receives a reservation request, it performs a search on the database to find a matching server.

A resource discovery server is useful in a large network (WAN) environment where there exists a large number of shared servers. It would simply be too slow and inefficient for clients to contact each shared server to find out its resource availability. A resource discovery server can provide much faster response to the clients' requests. However, the design of the resource discovery server needs to be scalable to the number of shared servers in the network. One possible solution is a two-level hierarchical design where we partition the network into domains. Each domain has its own resource discovery server, and reservations across domains are handled by an inter-domain resource discovery server.

When a pricing model is enforced over a network, a client may also specify a *pricing constraint* in a request in addition to its resource requirements. A resource discovery server will attempt to find a shared server that gives the best pricing below the pricing constraint, while satisfying the resource requirements.

Bibliography

[1] ATM FORUM Technical Committee. Traffic Management Specification Version 4.0, April 1996. Available from http://www.atmforum.com/atmforum/specs/approved.html.

[2] Robin Bargar, Insook Choi, and Alex Betts. ScoreGraph: a Software Architecture for Rapid Configuration of Multi-Modal Interaction in Distributed Virtual Environments. In *ARL Federated Laboratory Advanced Displays and Interactive Displays Consortium, Advanced Displays and Interactive Displays Third Annual Symposium*, February 1999.

[3] Steven Berson, Robert Lindell, and Robert Braden. An Architecture for Advance Reservations in the Internet. Technical report, USC Information Sciences Institute, July 1998.

[4] Richard Black, Paul Barham, Austin Donnelly, and Neil Stratford. Protocol Implementation in a Vertically Structured Operating System. In *IEEE Conference on Computer Networks*, November 1997.

[5] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource ReSerVation Protocol (RSVP) - Version 1 Functional Specification, September 1997. RFC 2205.

[6] Scott Brandt, Gary Nutt, Toby Berk, and Marty Humphrey. Soft Real-Time Application Execution with Dynamic Quality of Service Assurance. In *6th IEEE/IFIP International Workshop on Quality of Service (IWQoS '98)*, May 1998.

[7] Scott Brandt, Gary Nutt, Toby Berk, and James Mankovich. A Dynamic Quality of Service Middleware Agent for Mediating Application Resource Usage. In *19th IEEE Real-Time Systems Symposium*, December 1998.

[8] Allan Bricker, Michael Litzkow, and Miron Livny. Condor Technical Report. Technical report, University of Wisconsin at Madison, January 1992.

[9] George M. Candea and Michael B.Jones. Vassal: Loadable Scheduler Support for Multi-Policy Scheduling. In *2nd USENIX Windows NT Symposium*, pages 157–166, August 1998.

[10] Steve Chapin, Dimitrios Katramatos, John Karpovich, and Andres Grimshaw. Resource Management in Legion. Technical Report CS-98-11, Computer Science Department, University of Virginia, 1998.

[11] Hao-hua Chu and Klara Nahrstedt. A Soft Real Time Scheduling Server in UNIX Operating System. In *European Workshop on Interactive Distributed Multimedia Systems and Telecommunication Services*, September 1997.

[12] Hao-hua Chu and Klara Nahrstedt. CPU Service Classes for Multimedia Applications. In *IEEE International Conference on Multimedia Computing and Systems (IEEE Multimedia Systems '99)*, June 1999.

[13] D. D. Clark, S. Shenker, and L. Zhang. Supporting Real-Time Applications in an Integrated Services Packet Network: Architecture and Mechanism. In *SIGCOMM '92*, August 1992.

[14] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[15] Erik Cota-Robles and James P. Held. A Comparison of Windows Driver Model Latency Performance on Windows NT and Windows 98. In *3rd USENIX Symposium on Operating Systems Design and Implementation*, 1999.

[16] Carolina Cruz-Neira, Daniel J. Sandin, Thomas A. DeFanti, Robert V. Kenyon, and John C. Hart. The CAVE: Audio Visual Experience Automatic Virtual Environment. In *Communications of ACM*, pages 62–72, June 1999.

[17] A. Demers, S. Keshav, and S. Shenker. Analysis and Simulation of a Fair Queueing Algorithm. *Journal of Internetworking: Research and Experience*, pages 3–12, 1990.

[18] Zong Deng. *An Open System Environment for Real-Time Applications*. PhD thesis, Department of Electrical and Computer Engineering, University of Illinois at Urbana Champaign, 1998.

[19] Zong Deng and J.W.-S. Liu. Scheduling Real-Time Applications in an Open Environment. In *18th Real-Time Systems Symposium*, pages 308–319, December 1997.

[20] Zong Deng, J.W.-S. Liu, and Jun Sun. A Scheme for Scheduling Hard Real-Time Applications in Open System Environment. In *9th Euromicro Workshop on Real-Time Systems*, pages 191–199, June 1997.

[21] Bryan Ford and Sai Susarla. CPU Inheritance Scheduling. In *2nd Usenix Symposium on Operating System Design and Implementation*, October 1996.

[22] I. Foster, C. Kesselman, C. Lee, R. Lindell, K. Nahrstedt, and A. Roy. A Distributed Resource Management Architecture that Supports Advance Reservations and Co-Allocation. In *International Workshop on Quality-of-Service*, June 1999.

[23] Ian Foster and Carl Kesselman. The Globus Project: A Status Report. In *IPPS/SPDP '98 Heterogeneous Computing Workshop*, 1998.

[24] Mark K. Gardner and Jane W. S. Liu. Analyzing Stochastic Fixed-Priority Real-Time Systems. In *5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, March 1999.

[25] Mark K. Gardner and Jane W. S. Liu. Performance of Algorithms for Scheduling Real-Time Systems with Overrun and Overload. In *11th Euromicro Conference on Real-Time Systems*, June 1999.

[26] Goutham Garimella. Advance CPU Reservations with the Dynamic Soft Real-Time Scheduler. Master's thesis, Department of Computer Science, University of Illinois at Urbana Champaign, 1999.

[27] Roland Geisler. A Remote Monitor System for Distributed Applications Using the Soft Real-Time Scheduler. Master's thesis, Department of Computer Science, University of Illinois at Urbana Champaign, 1999.

[28] Chris Gill, David Levine, Douglas C. Schmidt, and Fred Kuhns. The Design and Performance of a Real-Time CORBA Scheduling Service. *International Journal of Time-Critical Computing Systems special issue on Real-Time Middleware*, to appear in 1999.

[29] Ashvin Goel, David Steere, Calton Pu, and Jonathan Walpole. SWiFT: A Feedback Control and Dynamic Reconfiguration Toolkit. Technical Report 98-009, Oregon Graduate Institute, September 1998.

[30] R. Gopalakrishnan. *Efficient Quality of Service Support within Endsystems for High Speed Multimedia Networking*. PhD thesis, Department of Computer Science, Washington University, December 1996.

[31] Pawan Goyal, Xingang Guo, and Harrick Vin. A Hierarchical CPU Scheduler for Multimedia Operating System. In *2nd Usenix Symposium on Operating System Design and Implementation*, October 1996.

[32] MONET Group. Dynamic Soft Real Time System Software Release, 1999. Available from http://cairo.cs.uiuc.edu/software.html.

[33] Manish P. Gupta. Reservation Based Distributed Resource Management. Master's thesis, Department of Computer Science, University of Illinois at Urbana Champaign, 1999.

[34] M. Jones and J. Regehr. Issues in Using Commodity Operating Systems for Time-Dependent Tasks: Experiences from a Study of Windows NT. In *8th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV '98)*, July 1998.

[35] Michael B. Jones, Joseph S. Barrera III, Alessandro Forin, Paul J. Leach, Daniela Rosu, and Marcel-Catalin Rosu. An Overview of the Rialto Real-Time Architecture. In *7th ACM SIGOPS European Workshop*, pages 249–256, September 1996.

[36] Michael B. Jones and John Regehr. The Problems You're Having May Not Be the Problems You Think You're Having: Results from a Latency Study of Windows NT. In *7th Workshop on Hot Topics in Operating Systems (HotOS-VII)*, March 1999.

[37] Michael B. Jones, Daniela Rosu, and Marcel-Catalin Rosu. CPU Reservation and Time Constraints: Efficient, Predictable Scheduling of Independent Activities. In *16th ACM Symposium on Operating Systems Principles*, October 1997.

[38] Jun Kamada, Masanobu Yuhara, and Etsuo Ono. User-level Realtime Scheduler Exploiting Kernel-level Fixed Priority Scheduler. In *Multimedia Japan*, March 1996.

[39] Kihun Kim. Monet Internal Design Document, May 1999.

[40] Intel Architecture Labs. Tools and Techniques for Softmodem IHV/ISV Self-Validation of Compliance with PC 99 Guidelines for Driver-Based Modems, July 1998. Available from http://developer.intel.com/ial/sm/doc.htm.

[41] Intel Architecture Labs. Windows 98 Latency Characterization for WDM Kernel Drivers, September 1998. Available from http://developer.intel.com/ial/sm/doc.htm.

[42] Chen Lee, Ragunathan Rajkumar, and Clifford Mercer. Experience with Processor Reservation and Dynamic QOS in Real-Time Mach. In *Multimedia Japan*, 1996.

[43] Ian Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul Barham, David Evers, Robin Fairbairns, and Eoin Hyden. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *IEEE Journal on Selected Areas In Communications*, pages 1280–1297, September 1996.

[44] Chih-han Lin, Hao-hua Chu, and Klara Nahrstedt. A Soft Real-time Scheduling Server on the Windows NT. In *2nd USENIX Windows NT Symposium*, August 1998.

[45] C.L. Liu and J.W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment. *JACM*, pages 46–61, 1973.

[46] Miron Livny and Rajesh Raman. *In Chapter High Throughput Resource Management, The Grid: Blueprint for a New Computing Infrastructure.* Morgan Kaufman Publishers Inc., Junly 1998.

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

[48] Clifford W. Mercer and Ragunathan Rajkumar. An Interactive Interface and RT-Mach Support for Monitoring and Controlling Resource Management. In *Real-Time Technology and Applications Symposium*, May 1995.

[49] Klara Nahrstedt, Hao-hua Chu, and Srinivas Narayan. QoS-Aware Resource Management for Distributed Multimedia Applications. *Journal on High-Speed Networking, Special Issue on Multimedia Networking*, pages 229–258, 1998.

[50] Klara Nahrstedt and Jonathan M. Smith. Design, Implementation and Experiences of the OMEGA End-Point Architecture. *IEEE JSAC, Special Issue on Distributed Multimedia Systems and Technology*, pages 1263–1279, September 1996.

[51] Klara Nahrstedt and Ralph Steinmetz. Resource Management in Networked Multimedia Systems. *IEEE COMPUTER*, pages 52–63, May 1995.

[52] Srinivas Narayan. Multimedia Efficient and QoS-Aware Transport Subsystem. Master's thesis, Department of Computer Science, University of Illinois at Urbana Champaign, 1997.

[53] Jason Nieh and M.S. Lam. SMART UNIX SVR4 Support for Multimedia Applications. In *IEEE International Conference on Multimedia Computing and Systems*, June 1997.

[54] Jason Nieh and M.S. Lam. The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications. In *16th ACM Symposium on Operating Systems Principles*, October 1997.

[55] Jason Nieh and M.S. Lam. Multimedia on Multiprocessors: Where's the OS When You Really Need It? In *8th International Workshop on Network and Operating System Support for Digital Audio and Video*, July 1998.

[56] David A. Patterson and John L. Hennessy. *Computer Architecture: a Qualitative Approach.* Morgan Kaufmann Publishers, Inc., 1996.

[57] C.S. Potter, B. Carragher, H. Chu, B.J. Frey, R. Josephs, C. Lin, N. Kisseberth, K.L. Miller, and K. Nahrstedt. A Testbed for Automated Acquisition from a TEM. In *Microscopy and Microanalysis '98*, July 1998.

[58] Raj Rajkumar, Chen Lee, John Lehoczky, and Dan Siewiorek. Practical Solutions for QoS-based Resource Allocation Problems. In *IEEE Real-Time Systems Symposium*, December 1998.

[59] Olov Schelen and Stephen Pink. Resource Sharing in Advance Reservation Agents. *Journal of High Speed Networks, Special issue on Multimedia Networking*, 1998.

[60] Douglas C. Schmidt, David Levine, and Sumedh Mungee. The Design of the TAO Real-Time Object Request Broker. *Computer Communications Special Issue on Building Quality of Service into Distributed Systems, Elsevier Science*, April 1998.

[61] David C. Steere, Ashvin Goel, Joshua Gruenberg, Dylan McNamee, Calton Pu, and Jonathan Walpole. A Feedback-driven Proportion Allocator for Real-Rate Scheduling. In *Operating Systems Design and Implementation (OSDI)*, Febuary 1999.

[62] Ion Stoica, Hussein Abdel-Wahab, and Kevin Jeffay. On the Duality between Resource Reservation and Proportional Share Resource Allocation. In *Multimedia Computing and Networking*, February 1997.

[63] Ion Stoica, Hussein Abdel-Wahab, Kevin Jeffay, Sanjoy K. Baruah, Johannes E. Gehrke, and C. Greg Plaxton. A Proportional Share Resource Allocation Algorithm for Real-Time, Time Shared Systems. In *17th Real-Time Systems Symposium*, December 1996.

[64] Too-Seng Tia, Zong Deng, Mallikarjun Shankar, Matthew Storch, Jun Sun, L. Wu, and J. Liu. Probabilistic performance Guarantee for Real-time Tasks with Varying Computation Times. In *Real-Time Technology and Applications Symposium*, May 1995.

[65] Nalini Venkatasubriamanian and Klara Nahrstedt. An Integrated Metric for Video QoS. In *ACM Multimedia*, November 1997.

[66] Carl A. Waldspurger and William E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *1st Symposium on Operating System Design and Implementation*, November 1994.

[67] David K.Y. Yau and Simon S. Lam. Operating System Techniques for Distributed Multimedia. Technical Report TR-95-36, University of Texas at Austin, July 1995.

[68] David K.Y. Yau and Simon S. Lam. Adaptive Rate-Controlled Scheduling for Multimedia Applications. In *ACM Multimedia*, November 1996.

[69] W. Yeong, T. Howes, and S. Kille. Lightweight Directory Access Protocol, March 1995. RFC 1777.

[70] Lixia Zhang. Virtual Clock: A New Traffic Control Algorithm for Packet Switching Networks. *ACM Transactions on Computer Systems*, pages 101–124, May 1991.

Vita

Hao-hua Chu was born on July 18, 1971 in Taipei, Taiwan, R.O.C. At an age of 14, he moved from Taipei to Massachusetts on July 10, 1985. From 1985 to 1986, he attended Cushing Academy, Ashburnham, Massachusetts. From 1986 to 1989, he transfered to Northfield Mount Hermon school, Northfield, Massachusetts, where he received his high school degree in May 1989. From 1989 to 1994, he enrolled as a student at Cornell University, Ithaca, New York. He received both his B.S. and M.E. degrees in Computer Science in January 1994. From January 1994 to August 1994, he worked as a software programmer at the Design Research Institute, Xerox Corporation, Ithaca, New York.

Since August 1994, he has been a Ph.D. student in the Department of Computer Science at University of Illinois at Urbana Champaign, Urbana, Illinois. From January 1995 to August 1995, he worked as a research assistant in Professor Andrew Chien's Concurrent Systems Architecture (CONCERT) group. From January 1996 to May 1996, he worked as a teaching assistant in Professor Klara Nahrstedt's Multimedia Computing Systems class. From May 1996 to August 1999, he worked as a research assistant in Professor Klara Nahrstedt's Multimedia Operating Systems and Networking (MONET) Group. He took a summer off research to work as an intern at the Validation Technology group at Intel Corporation, Folsom, California in 1997. After completing his Ph.D. in August 1999, he will join the Soft Migration project at Intel Architecture Lab, Hillsboro, Oregon.