# Dynamic QOS Control Based on the QOS-Ticket Model*

Kiyokuni KAWACHIYA
IBM Research, Tokyo Research Laboratory
1623-14, Shimotsuruma, Yamato,
Kanagawa 242, Japan
<kawatiya@trl.ibm.co.jp>

Hideyuki TOKUDA
Faculty of Environmental Information,
Keio University
5322 Endo, Fujisawa, Kanagawa 252, Japan
<hxt@sfc.keio.ac.jp>

## Abstract

*The most notable characteristic of continuous-media data is the existence of timing constraints. To handle such data appropriately, some system support for resource management and QOS control is indispensable. For this reason, we are proposing a "QOS-Ticket" model, which combines resource reservation and adaptation. The QOS-Ticket model has been implemented on RT-Mach. In this prototype, a new thread mechanism, named "Q-Thread," is provided. A Q-Thread allows a user to specify the tolerable range of the period and computation time for periodic invocations, and makes it easy to write a continuous-media session that can control the QOS dynamically. We showed the effectiveness of the QOS-Ticket model based on a dynamic QOS-control experiment with our prototype.*

**Keywords:** Dynamic QOS control, Resource management, Execution model, Real-time operating system, Multimedia

## 1. Introduction

The most notable characteristic of data for continuous media such as audio and video is the existence of constraints on the timing of the data processing. To handle such data appropriately, the system must observe the timing constraints, and must guarantee sufficient system resources (CPU, memory, disk, network, etc.) for the processing. Of course, since the real system resources are finite, sufficient resources may not be provided for a particular processing session. In such cases, the execution time is usually extended. But in continuous-media processing with timing constraints, the quality of service (QOS) rather than the execution time should be changed [1]. For example, in video processing, the frame rate and/or the quality of each frame can be changed to meet the resource restriction.

This QOS-control mechanism is closely related to the system's resource management, and it is difficult to achieve such control without any system support. However, most existing resource-management mechanisms focus on the fairness of the resource assignment, and provide insufficient support for the resource guarantee, enforcement, and observation. Therefore, a new resource-management mechanism should be developed in order to provide appropriate QOS control. To meet this need, we are proposing a "QOS-Ticket" model, which combines resource reservation and adaptation.

The QOS-Ticket model has been implemented on Real-Time Mach 3.0 (RT-Mach), using the system's processor capacity reservation mechanism. In this prototype, a new thread mechanism for continuous-media processing, named "Q-Thread," is provided. By using a Q-Thread, a user can specify the tolerable range of the period and computation time for periodic invocations. A dynamic QOS-control experiment was carried out with this prototype to show the effectiveness of the QOS-Ticket model.

In the rest of this paper, Section 2 first examines features required for QOS control of multimedia, and then introduces the QOS-Ticket model. Next, Section 3 describes an implementation of the QOS-Ticket model consisting of a QOS-Control Server and a Q-Thread library, and Section 4 reports an experiment using the prototype. After a discussion of related work on QOS control of multimedia in Section 5, Section 6 offers some conclusions and topics for future work.

## 2. A Model for Dynamic QOS Control

As explained in Section 1, some new resource-management mechanism should be developed for handling the timing constraints of continuous media. There may be a "static" resource-management policy according to which a continuous-media session is allowed to start only if there are sufficient resources for the processing (admission control). However, in an interactive multimedia environment that

*This research is conducted under the Open Fundamental Software Technology Project of Information-technology Promotion Agency, Japan (IPA).

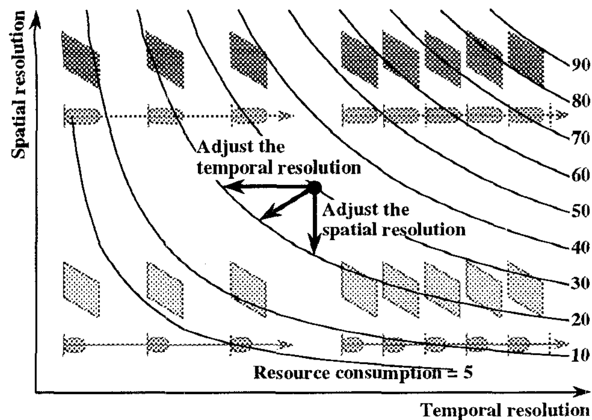


**Figure 1. Various ways of changing the QOS**

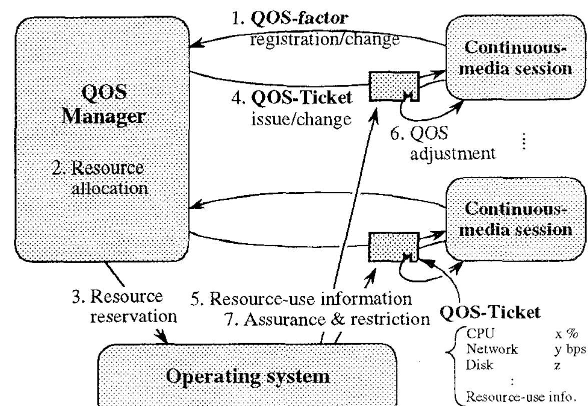


**Figure 2. QOS-Ticket model**

handles multiple sessions, some mechanism for controlling QOS dynamically among sessions is preferred.

For such a mechanism, we adopted the QOS Manager-based scheme [2]. In the following subsections, we will examine the required features of such a model, and introduce a "QOS-Ticket" model that meets the requirements.

## 2.1. Features Required for a QOS-Control Model

Before introducing the QOS-Ticket model, let us consider required features of a QOS-control architecture. The following features are considered to be required for such a framework:

1. Resource reservation and restriction
   Since the usual "fair-share" resource management is insufficient to allow each session to meet its timing constraints, a mechanism for guaranteeing (reserving) the resource assignment is necessary. Moreover, this resource management should be applicable to each session rather than to each process or thread, because a session is usually made up of multiple processes or threads.
2. Resource-use adaptation
   The QOS-control framework should not depend on some specific system configuration or set of media data. It is thus desirable that some feedback and adaptation mechanism based on the actual resource consumption should be provided in the framework.
3. Coordination of multiple sessions
   When a system handles multiple sessions, the QOS-control mechanism should take account of the characteristics of the sessions. For example, if there is a resource shortage, the QOS of low-priority sessions should be reduced more than that of high-priority sessions.
4. Separation of policy and mechanism
   There are various ways of changing a session's QOS to meet a specific resource restriction. The graph in Fig. 1 illustrates the relations between the temporal and spatial resolutions for specific levels of resource consumption. Arrows in the figure show various ways of reducing the QOS when the available resources are decreased from 30 to 20. The policy for choosing one of these ways should be kept separate from the mechanism for QOS control.

## 2.2. QOS-Ticket Model

We are proposing a new QOS-control model that satisfies all the above requirements, named the QOS-Ticket model [3, 4]. Figure 2 shows an outline of this model, which takes account of both resource reservation and adaptation, whereas most existing mechanisms include only one of them. Its core part is a mechanism named "QOS-Ticket," which is used for resource reservation and restriction, and for monitoring. In this QOS-Ticket model, resource management and QOS control are achieved through the cooperation of an operating system, a QOS Manager, and individual continuous-media sessions. The QOS-Ticket for each session, which represents that session's resource reservation, mediates these activities.

In the QOS-Ticket model, as a rule, all the system resources are used with reservation. The model's basic behavior is as follows. Each continuous-media session registers its "QOS factor," which describes the characteristics of the session, with the QOS Manager at the time it is started. When a session is registered, the QOS Manager calculates the resource allocation among sessions on the basis of the QOS factors, reserves resources, and issues a QOS-Ticket that contains the reservation information for the session. The session adjusts its own QOS to meet the resource restriction specified in the QOS-Ticket. The resource management needed for the QOS-Ticket is handled by the operating system.

The roles of the five components in the QOS-Ticket model are as follows:

- The **QOS factor**, registered by each session with the QOS Manager, describes the characteristics of the session. It specifies a priority and a tolerable allocation range (maximum and minimum values) for each resource request. The QOS Manager will allocate resources to meet this request.
- The **QOS-Ticket** is a "ticket," issued by the QOS Manager to each session, for preferential use of resources. It represents a reservation of resources for a session. Through the QOS-Ticket, a session can use resources preferentially, within the reserved limits. The resources consumed in accordance with a QOS-Ticket are recorded in it, and each session can determine the quantity of resources consumed, as well as its own resource-reservation status. The QOS-Ticket should be a resource-management mechanism for each session. Even if a session consists of multiple processes or threads, the QOS-Ticket should be shared among them.
- The **operating system** provides a resource-management mechanism for QOS-Ticket. This consists of the resource-reservation mechanism and the resource-use information offering. The resource reservation is indispensable for realizing QOS-Ticket. The resource-use information provides each session with important hints on how to control the QOS.
- The **QOS Manager** is a kind of scheduler that allocates resources to sessions. Basically, the allocation is calculated according to the QOS factors, but the policy is decided by the QOS Manager. On the basis of the allocation, the QOS Manager reserves resources and issues a QOS-Ticket to each session. When the number of sessions or some QOS factor is changed, the QOS Manager recalculates the resource allocation, modifies the resource reservation of the QOS-Ticket, and notifies each session of the change.
- Each **continuous-media session** estimates the amount of resources needed for processing, and declares it to the QOS Manager as part of the QOS factor. The requested resources are allocated by the QOS Manager, and a QOS-Ticket is issued. The session adjusts its own QOS to meet the resource restriction specified in the QOS-Ticket. The resource-use information in the QOS-Ticket can be used as a hint on the adjustment.[1]

In the QOS-Ticket model, the requirements listed in Section 2.1 are satisfied as follows. The resource reservation and restriction are done by the operating system. The QOS-Ticket provides the resource-use information, which enables the feedback and adaptation mechanism to realize the independency of the system or media data. The QOS Manager takes charge of the coordination of multiple sessions.

The separation of policy and mechanism is achieved on two levels. One level is that of resource allocation. The resource reservation mechanism resides in the operating system, while the resource allocation policy is up to the QOS Manager. The other level is that of QOS control. The resource allocation mechanism for QOS control is implemented in the QOS Manager (and operating system), but the QOS control based on the allocated resources is handled by each session according to its own policy. The QOS-Ticket (and QOS factor) mediate these three activities. In general, the QOS Manager and the operating system have more knowledge on the resource usage, while each session knows more about the characteristics of the media. Therefore, we consider this separation to be very reasonable.

## 3. Implementation of the QOS-Ticket Model

To realize resource management and QOS control in accordance with the QOS-Ticket model, we have been developing a prototype on RT-Mach, an operating system based on the Mach microkernel with several real-time extensions [5]. The first prototype is implemented for the CPU-resource management, and consists of the "QOS-Control Server" and the "Q-Thread library." The QOS-Control Server manages the CPU allocation among multiple sessions, and the Q-Thread library helps a user program in each session to adjust its QOS. As the QOS-Ticket for the CPU resource, the prototype uses the "processor capacity reservation" mechanism of RT-Mach. The CPU resource was chosen as the first target for the following reasons:

- The CPU reservation mechanism of RT-Mach can be used as a prototype of QOS-Ticket.
- The CPU is a typical shared resource in a system, and should be carefully controlled.
- The CPU is a comparatively easy resource to control dynamically.

Figure 3 shows the structure of the prototype. The following subsections give details of the implementation of this prototype, and the next section describes a dynamic QOS-control experiment using the prototype.

### 3.1. The QOS-Control Server

In the prototype, the CPU reservation mechanism of RT-Mach [6] is used. This mechanism makes available in the system a new abstraction named "CPU reserve." A reserve represents a CPU-resource reservation in the form of "requires $C$ seconds every $T$ seconds (i.e. $\frac{C}{T}$ of the CPU)." Every thread in the system is associated with a CPU reserve, and can gain preferential use of the CPU through it.

[1] In fact, it is desirable that some support library for a QOS adjustment should be provided. The Q-Thread library, which will be described in Section 3.2, is an example of such a support library.

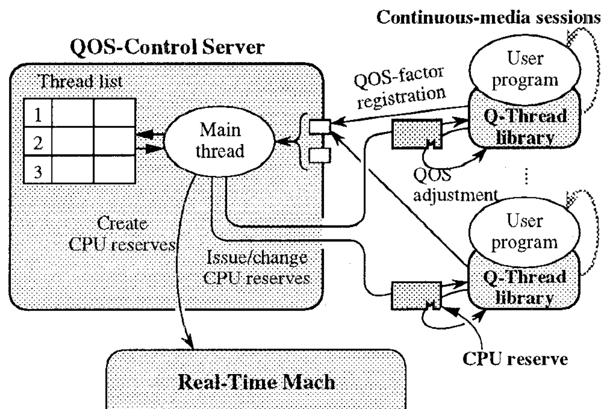


**Figure 3. The QOS-Control Server and the Q-Thread library**

A CPU reserve accumulates information on the CPU time used by its associated threads. On the basis of this information, in assigning the CPU, the scheduler gives top priority to threads whose CPU reserves are not used up. Each thread can also access the CPU-time information in its reserve, and thus find out how much CPU time has been spent on its processing. It is also possible to associate multiple threads with one reserve or to pass a reserve to a server via RT-Mach IPC.

In a word, a CPU reserve has all the required features of a QOS-Ticket, which were described in Section 2.2.

The QOS-Control Server is an experimental implementation of the QOS Manager for CPU-resource allocation [3, 4]. As mentioned above, a CPU reserve is used as a QOS-Ticket. As QOS factors, the prototype uses the range of CPU-resource allocation (minimum $a\%$ to maximum $b\%$) that a session can accept and the priority of the session.

Each session registers its QOS factor with the server via Mach IPC when it is started. In accordance with the QOS factors, the QOS-Control Server calculates the CPU-resource assignment for each session, and "issues" a CPU reserve to each session.[2] In the current implementation, 90% of the CPU resource is divided among the sessions in proportion to their priorities, with some possible adjustment to match the requested range.

When a CPU reserve is issued (or modified), each session must adjust its own QOS to meet the restriction imposed by the reserve, with the assistance of the Q-Thread library. The CPU-time information in the reserve can be used as a hint in making this QOS adjustment. If a session has just been registered or completed, or the QOS factor of some session has been changed, the server recalculates the CPU-resource assignment and sets it to the CPU reserves. Consequently, each session re-adjusts its QOS.

[2]Strictly speaking, the server creates a reserve for each session and associates the session's thread with the reserve.

**Table 1. Comparison of the periodic RT-Thread and the Q-Thread**

| | Periodic RT-Thread | Q-Thread |
|---|---|---|
| Function | User-specified entry point is called periodically. | |
| Thread's attribute | (Fixed) invocation period, etc. | *Range* of the period and computation time, QOS-control policy, etc. |
| Invocation period | Fixed (can be re-specified) | Dynamically changed within the range based on the QOS-control policy |
| Arguments passed to the entry point | User-specified argument | Same, plus a QOS-control hint (available computation time, etc.) |
| Guarantee of execution | None (possible with CPU reservation) | Guaranteed, within the available computation time |
| Purpose | (Soft/hard) real-time processing | Continuous-media processing (with dynamic QOS control) |

## 3.2. The Q-Thread Library

Of course, a user can write a continuous-media session that directly negotiates with the QOS-Control Server, as we reported previously [3, 4]. However, the QOS factor and the QOS-Ticket (CPU reserve) are rather system-oriented (i.e. low-level) information, and it is appreciated that this information is "translated" into more abstract indices that can be easily handled by a user program. Moreover, the sequence of negotiations with the QOS-Control Server may not differ much from one session to another. Therefore, we have developed a user-level support library that negotiates with the QOS-Control Server and helps a user program in each session to adjust its QOS. This library provides a new thread mechanism for continuous-media processing, named "Q-Thread" [7].

### 3.2.1. Overview of the Q-Thread

Continuous-media processing programs can be written in RT-Mach by using the system's periodic real-time thread (RT-Thread) [8]. A media data unit (MDU), such as a frame of video data, is processed on every invocation. For this processing, the QOS can be adjusted by changing the period (invocation interval) of the processing (temporal resolution) and/or the amount of computation for each processing unit (spatial resolution). The Q-Thread library helps with this adjustment, and is implemented as a user-level library by means of the periodic RT-Thread.

The Q-Thread can be considered as an extension of the periodic RT-Thread for continuous-media processing with

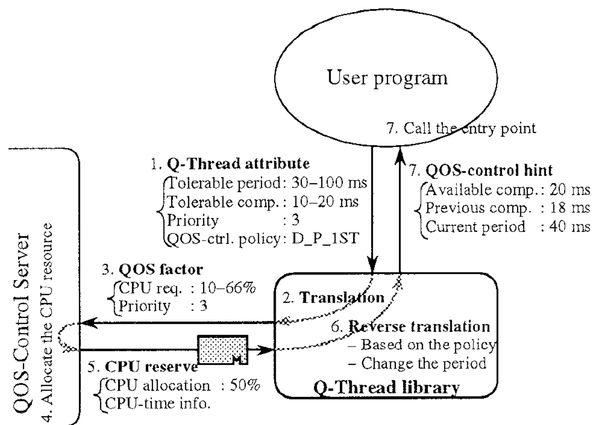


**Figure 4. Example of QOS translations by the Q-Thread library**

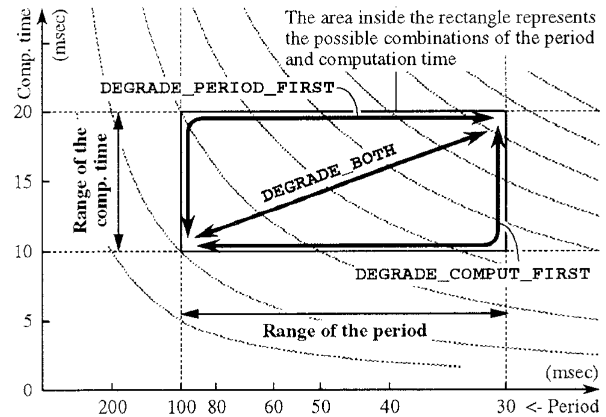


**Figure 5. The Q-Thread attribute and the QOS-control policies**

dynamic QOS control. In the periodic RT-Thread, a fixed invocation period is specified as a thread's attribute,[3] but in the Q-Thread, the tolerable "range" of the period and computation time (CPU time) for invocations can be specified. The user-specified entry point is called periodically, but the period (invocation interval) is changed automatically and dynamically within the specified range in accordance with the available CPU resource allocated by the QOS-Control Server. In addition to this adjustment of the period, the information of available computation time is passed to a user program at each invocation as a hint for workload adaptation. Table 1 summarizes the comparison of the periodic RT-Thread and the Q-Thread.

### 3.2.2. QOS Translations by the Q-Thread Library

The major roles of the Q-Thread library are translations of the QOS specifications as follows:

- "Translation" of the Q-Thread attribute into the QOS factor, and negotiating with the QOS-Control Server
- "Reverse translation" of the CPU reserve into a QOS-control hint, and providing the hint to the user program

Figure 4 illustrates this sequence of translation. When a user program indicates the creation of a Q-Thread, the library first "translates" the range of the period and computation time in the Q-Thread attribute into the QOS factor, and then creates a periodic RT-Thread and registers the QOS factor with the QOS-Control Server. In response to the registration, a CPU reserve (QOS-Ticket) is issued by the QOS-Control Server. The available CPU-resource amount indicated in the reserve is "reverse-translated" into some possible combination of period and computation time, and the period of the RT-Thread is changed in accordance with the result. The result is also passed to the user program as a hint on QOS control.

### 3.2.3. QOS-Control Policies of the Q-Thread

Several policies can be considered for reverse-translating the CPU reserve into a possible period and computation time. In the current implementation, the following four policies are provided. The `DEGRADE_PERIOD_FIRST` policy tries to preserve the maximum computation time, and degrades the period first in a CPU-resource shortage. The `DEGRADE_COMPUT_FIRST` policy tries to preserve the minimum invocation period. The `DEGRADE_BOTH` policy degrades the period and computation time equally. In the `HINT_ONLY` policy, the QOS adjustment is totally left up to the user program. The library does nothing other than provide a QOS-control hint. One of these policies is specified in the Q-Thread attribute.

Figure 5 shows the relation of the Q-Thread attribute and the QOS-control policies. In this example, the range from 30 to 100 msec is specified as the tolerable invocation period, and the range from 10 to 20 msec is specified as the tolerable computation time for invocations. Consequently, the inside of the rectangle in the figure contains all the possible combinations of the period and computation time. How a specific combination should be chosen from the available CPU resource is based on the QOS-control policy, as shown by the three paths in the figure.

### 3.2.4. QOS-Control Hint

The period of a Q-Thread is automatically changed by the library (unless the policy is `HINT_ONLY`). However, the amount of computation performed at each invocation must be adjusted inside the user program. To facilitate

[3] It is possible to change the period of RT-Thread by re-specifying the thread's attribute. The Q-Thread is implemented by using this function.

**Table 2. Functions provided by the Q-Thread library**

| Name | Description |
|---|---|
| `qoslib_init()` | Initializes the Q-Thread library |
| `qthread_attribute_init()` | Initializes a Q-Thread attribute |
| `qthread_create()` | Creates a Q-Thread |
| `qthread_get_comput()` | Gets the consumed CPU time |
| `qthread_get_attribute()` | Retrieves a Q-Thread attribute |
| `qthread_set_attribute()` | Re-specifies a Q-Thread attribute |
| `qthread_terminate()` | Terminates a Q-Thread |

this adjustment, a "QOS-control hint" is passed to the user program when the user-specified entry point is called. So long as the user program observes the computation-time indication in the hint, its execution is guaranteed by the CPU reserve.

The QOS-control hint also contains information on the CPU time that was actually consumed at the last invocation. This information is calculated by the Q-Thread library from the accumulated CPU-time information in the CPU reserve, and can be used by the user program to adjust its workload to match the available computation time.

In addition, the user program can re-specify (modify) the Q-Thread attribute in accordance with the QOS-control hint. In this case, the Q-Thread library re-registers the QOS factor with the QOS-Control Server, and the CPU allocation is recalculated.

### 3.3. A User Program Using the Q-Thread

By using the Q-Thread, a user can easily write a continuous-media session that controls the QOS dynamically in accordance with the QOS-Ticket model. All exported functions of the Q-Thread library are listed in Table 2.

Figure 6 shows a sample user program based on the Q-Thread library. In this example, the user-specified entry point `qt_entry()` is called periodically with periods from 30 to 100 msec. The available computation time (CPU time) for each invocation is indicated inside the second argument, `qt_hint`, and the user program must change the amount of its work on the basis of this information. Although not shown in the example, the user program can also modify the range of tolerable computation time using the `qthread_set_attribute()` function, in accordance with the previous actual computation time given in the `qt_hint` structure.

## 4. An Experiment in Dynamic QOS Control

The QOS-Control Server and Q-Thread library have already been implemented on RT-Mach. Both programs

```
#include "qoslib.h"

void qt_entry( /* Periodic entry point                    */
  int           *arg,   /* User-specified argument        */
  qthread_hint_t qt_hint/* QOS-control hint               */
)
{
  DO_SOME_WORK(qt_hint);
        /* qt_hint->comput indicates the available
           computation time (msec)                        */
        /* qt_hint->comput_prev gives the actual
           computation time at the last invocation        */
}

int main(int ac, char **av)
{
  qthread_t             qthread;
  qthread_attr_data_t   qt_attr;
  kern_return_t         kr;

  kr = qoslib_init();   /* Initialize the Q-Thread lib. */
  kr = qthread_attribute_init( /* Set up a Q-Thread attr*/
         qt_entry,NULL, /* Periodic entry point and arg.*/
         30,100,        /* Range of the period (msec)   */
         10,20,         /* Range of the comp.time (msec)*/
         3,             /* Priority                     */
         DEGRADE_PERIOD_FIRST, /* QOS-control policy    */
         NULL,NULL,     /* Deadline handler and its arg.*/
        &qt_attr);
  kr = qthread_create(mach_task_self(),&qthread,&qt_attr);
                        /* Create a Q-Thread            */
  thread_terminate(mach_thread_self());
}
```

**Figure 6. Sample program using the Q-Thread**

are written in C; the QOS-Control Server contains about 1500 lines and the Q-Thread library about 1000 lines. An experiment in dynamic QOS control was carried out to show the effectiveness of the QOS-Ticket model.

In this experiment, three dummy threads written by using the Q-Thread library were used. When the entry point is called, these dummy threads use up the CPU resource for the amount of the available computation time indicated in the QOS-control hint.[4] The Q-Thread attributes of the dummy threads are shown in Table 3. All threads specify the same ranges of the period (30 to 100 msec) and of the computation time (10 to 20 msec), but the priority (a high value means high priority) and the QOS-control policy are different. In addition to these dummy threads, a "disturbing" thread runs from 50 to 60 sec. This thread tries to exhaust the CPU by looping infinitely.

We observed the behavior of dynamic QOS control by executing these threads with the QOS-Control Server. The experiment was carried out on an IBM ThinkPad 755C (9545-L, IntelDX4-75MHz). The version of RT-Mach was MK94[5] and the resolution of the system clock was changed

[4] In fact, this workload adjustment should be based on information about the previous computation time and so on, but, this mechanism is omitted in our experiment. For this point, we are currently evaluating "real" applications such as a QuickTime player written with the Q-Thread.

[5] RT-Mach MK94 is a version released in 1994 by our Keio-MMP project [17], based on CMU's MK83i with some extensions.

**Table 3. Q-Thread attributes of the dummy threads in the experiment**

| No. | Exec. term (sec) | Period (msec) | Comp. (msec) | Priority | QOS-ctrl. policy |
|---|---|---|---|---|---|
| 1 | 00 → 30<br>30 → 70 | 30 – 100 | 10 – 20 | 2 | D_P_1ST<br>D_C_1ST |
| 2 | 10 → 80 | 30 – 100 | 10 – 20 | 3 | D_P_1ST |
| 3 | 20 → 40<br>40 → 80 | 30 – 100 | 10 – 20 | 3<br>6 | D_BOTH |
| D | 50 → 60 | (Disturbing thread that exhausts the CPU by looping infinitely) | | | |

to 1 msec. Figure 7 shows the results. The upper part of this figure shows the actual period (i.e. temporal QOS) of each dummy thread, and the lower part shows the CPU utilization and deadline misses during the experiment. The available computation time (i.e. spatial QOS) indicated by the Q-Thread library is also shown in the upper graph by expressions of the form "c = ... ."

In the first 10 seconds, only thread 1 was running and the highest QOS (30 msec period and 20 msec computation time) was achieved. But after higher-priority threads 2 and 3 were started at 10 and 20 sec, the period of thread 1 was extended to about 90 msec. Threads 2 and 3 had the same priority, but their QOS-control policies were different. Therefore, thread 3 was able to run with a shorter period than thread 2, but its available computation time was smaller than that of thread 2.

After 30 sec, the QOS-control policy of thread 1 was changed to DEGRADE_COMPUT_FIRST. Consequently, the period of thread 1 was reduced and the computation time was decreased. In this case, the QOS factor was not re-registered, so this change had no influence on other threads. After 40 sec, the priority of thread 3 was changed and its QOS factor was re-registered. By this re-registration, the QOS-Control Server recalculated the CPU-resource assignment and set it to the CPU reserves. As a result of the recalculation, the period and computation time of the three dummy threads were modified.

Even during the 50 to 60 sec that the disturbing thread was running, the three dummy threads were able to run without any influence, regardless of the 100% CPU utilization shown in the lower graph. The behavior of the dummy threads was very stable, and very few deadline misses occurred, as the lower graph shows.[6]

The experiment shows that the QOS of the CPU resource can be effectively controlled by the QOS-Control Server and the Q-Thread library in accordance with the QOS-Ticket model.

[6] The deadline misses observed in the lower graph were caused by the consumption of more CPU resources than the available computation time allowed, because of some mis-evaluation of the workload.

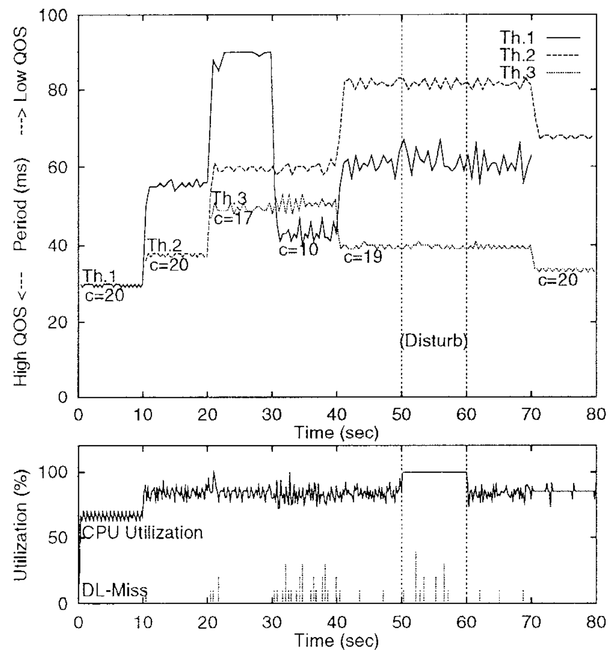


**Figure 7. Result of dynamic QOS control with the QOS-Control Server and Q-Threads**

## 5. Related Work

There have been several studies of QOS-control architecture and execution models for continuous-media processing. This section introduces some of these activities, and compares them with ours.

Researchers at Lancaster University have tried to provide an integrated and coherent framework for QOS control [9]. A group at the University of Pennsylvania has proposed a "QOS Broker" model that organizes multimedia resource management over network [10]. This model uses database files named "QOS profile" for the QOS translation, while our model uses the feedback and adaptation mechanism. The Rialto OS by Microsoft Research proposes hierarchical resource management in the operating system [11]. The highest-level "Resource Planner" controls the resource allocation of the whole system. The RT-Mach group at Carnegie Mellon University is also developing a server for QOS control and a network-phone system, using the processor capacity reservation [12]. Researchers at Fujitsu Laboratories are proposing an interesting QOS-control scheme, which uses a "market mechanism" for the acquisition of resources by multiple sessions [13].

For scheduling of processors in continuous-media processing, the Rate-Based Execution model has been proposed by a group at the University of North Carolina [14]. This model uses $(x, y, d)$ parameters to represent the execution

"rate" of a process. There have also been several studies of processor scheduling for continuous-media processing [15, 16].

In relation to these studies, we think that the unique features of our proposed mechanism are as follows:

- We focus on QOS control of multiple sessions.
- Our model incorporates both reservation and adaptation.
- Actual implementation is being done on RT-Mach.

Whereas some of the above research projects focus on the QOS control of individual sessions over a network, ours focuses on dynamic QOS control of multiple sessions in a system. However, it may be possible to use the QOS-control policies of the above projects to adjust the QOS of each session in our QOS-Ticket model. Some of the previous QOS-control research considers only the resource reservation or the resource adaptation, while our QOS-Ticket model takes account of both reservation and adaptation. In addition to designing a model, we have also started to implement the QOS Manager and its support library in accordance with the QOS-Ticket model.

## 6. Conclusions and Future Work

In this paper, we first introduced the QOS-Ticket model as a new resource-management and QOS-control architecture for continuous-media applications. In this model, dynamic QOS control is achieved through cooperation of an operating system, a QOS Manager, and individual continuous-media sessions. The QOS-Ticket, which represents the resource reservation for each session, mediates these activities. Next, we described the QOS-Control Server and Q-Thread library as an implementation of the QOS-Ticket model. The Q-Thread library provides a new thread mechanism for continuous-media processing, named "Q-Thread." A Q-Thread allows a user to specify the tolerable range of the period and computation time for invocations, and makes it easy to write a continuous-media session that can control the QOS dynamically. We also described a dynamic QOS-control experiment, using our prototype, that shows the effectiveness of the QOS-Ticket model.

As we have shown, the QOS-Ticket model works very well for managing the CPU resource. However, our ultimate target is to manage all system resources — not only the CPU, but also the memory, disk, network, and so on — in an integrated fashion, within the framework of the QOS-Ticket model. Currently, the resource-control mechanisms for QOS-Ticket are being added to RT-Mach. In parallel with this OS-level extension, we are developing a new QOS Manager by extending the QOS-Control Server. Real applications such as a QuickTime player are also being developed by using the Q-Thread.


## Acknowledgements

The authors would like to thank members of the Keio-MMP project [17] and the Keio-IBM partnership program [18] for their help and advice.


## References


[1] J. F. K. Buford et al.: *Multimedia systems*, Chapter 8, ACM Press SIGGRAPH Series, Addison-Wesley (1994).

[2] H. Tokuda and T. Kitayama: "Dynamic QOS Control based on Real-Time Threads," *Proc. NOSSDAV '93*, pp. 113–122 (1993).

[3] K. Kawachiya et al.: "Evaluation of QOS-Control Servers on Real-Time Mach," *Proc. NOSSDAV '95*, pp. 123–126 (1995).

[4] K. Kawachiya and H. Tokuda: "QOS-Ticket: A New Resource-Management Mechanism for Dynamic QOS Control of Multimedia," *Proc. Multimedia Japan '96*, pp. 14–21 (1996).

[5] H. Tokuda et al.: "Real-Time Mach: Towards a Predictable Real-Time System," *Proc. USENIX Mach Workshop*, pp. 73–82 (1990).

[6] C. W. Mercer et al.: "Processor Capacity Reserves: Operating System Support for Multimedia Applications," *Proc. IEEE ICMCS '94*, pp. 90–99 (1994).

[7] K. Kawachiya and H. Tokuda: "Q-Thread: A New Execution Model for Dynamic QOS Control of Continuous-Media Processing," *Proc. NOSSDAV '96* (1996).

[8] H. Tokuda et al.: "A Real-Time Thread Model for Continuous Media Applications," *ART Group Tech. Report*, Carnegie Mellon University, May 1993 (1993).

[9] A. Campbell et al.: "Dynamic QoS Management for Scalable Video Flows," *Proc. NOSSDAV '95*, pp. 107–118 (1995).

[10] K. Nahrstedt and J. M. Smith: "The QOS Broker," *IEEE Multimedia*, Vol. 2, No. 1, pp. 53–67 (1995).

[11] M. B. Jones et al.: "Support for User-Centric Modular Real-Time Resource Management in the Rialto Operating System," *Proc. NOSSDAV '95*, pp. 55–65 (1995).

[12] C. Lee et al.: "Experiences with Processor Reservation and Dynamic QOS in Real-Time Mach," *Proc. Multimedia Japan '96*, pp. 22–31 (1996).

[13] T. Aoki et al.: "Dynamic QOS control using a market mechanism," *Proc. 51st Annual Convention IPS Japan*, pp. 4-5–4-6, in Japanese (1995).

[14] K. Jeffay and D. Bennett: "A Rate-Based Execution Abstraction for Multimedia Computing," *Proc. NOSSDAV '95*, pp. 67–78 (1995).

[15] J. Nieh and M. S. Lam: "Integrated Processor Scheduling for Multimedia," *Proc. NOSSDAV '95*, pp. 215–218 (1995).

[16] R. Yavatkar and K. Lakshman: "A CPU Scheduling Algorithm for Continuous Media Applications," *Proc. NOSSDAV '95*, pp. 223–226 (1995).

[17] Keio-MMP Project: WWW Home Page, URL: <http://www.mmp.sfc.keio.ac.jp/>.

[18] Keio-IBM Partnership Program: WWW Home Page, URL: <http://ibmpp.sfc.wide.ad.jp/>.