

# Real-time Multiple Video Player Systems

Chris C.H. Ngan and Kam-Yiu Lam  
Department of Computer Science  
City University of Hong Kong  
83 Tat Chee Avenue, Kowloon  
HONG KONG

## Abstract

*In previous years, various real-time scheduling techniques have been proposed to improve the performance of a distributed multimedia system. However, not much work has been done on how a client player can play multiple videos concurrently. In this paper, we have designed a distributed multimedia system where a client player can play different videos at the same time. We introduce two mechanisms: priority assignment mechanism and feedback capture video mechanism for the system. The issues on how to implement the mechanisms in Windows NT are discussed. Experiments are conducted to investigate the performance of the proposed mechanism in Windows NT to study the most optimum configuration factors. We also define the monitoring factor and priority group mapping difference, and study how they affect the overall system performance. From the results, we find that the mechanisms can effectively improve the system performance in terms of percentage of displayed frames.*

## 1 Introduction

In recent years, a lot of work has been done in the study of distributed video player systems [1,2,6]. Real-time multiple video play system is focusing on how to retain a good quality of service when multiple videos are playing in a client player. Examples of the applications can be the security guard monitoring system, which monitors different corridors and lifts.

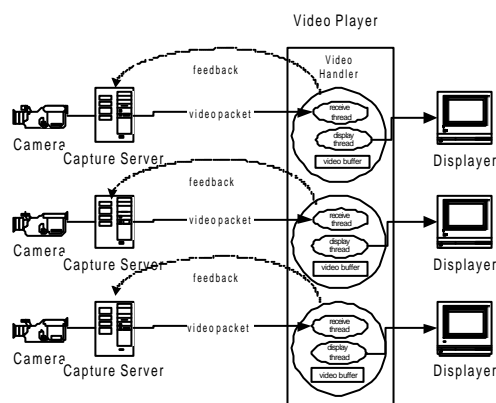
The main objective of the design of the systems is to ensure the quality of service (QoS) in the player. To ensure a high QoS, the system must be predictable. A predictable system can be attained with the use of sophisticated scheduling policies to manage the system resources such as the CPU in the video server, the network, the buffers and the CPU in the client [1,3,7]. Although the simplest way is to use some static mechanisms for resource allocation [7], they may not be practical for many distributed multimedia applications, as the mechanisms require priori knowledge on the system environment and the requirements of each client. The required information may not be available. Furthermore, the requirements of the clients are not static. They may request videos with different sizes and resolutions, and to playback the videos at different speeds. At the same time,

the transmission delay of the video frames in the network is obviously unpredictable. It is difficult to predict network jitter and latency due to the differences in bandwidth among different links in the network and the dynamic workload of the network. In order to improve the predictability of the system performance and to provide a better QoS in the video playback, the system must be adaptive to the changing environment [1,3].

As return to our system, if we only simply connect the client player to different capture cameras and leave the player to play video in a traditional round-robin manner, it is difficult to control the QoS since there exist many factors (e.g. network, CPU resource) which can affect the QoS of the playback. We need to adjust the priority of the client player in response to the displayed quality of the video [1,5]. However, if the video player still comes to an unendurable status, only adjusting the priority scheduling or buffer control is not enough. We need to send feedback [1,3] to the server side to perform some actions in order to improve the system performance such as to reduce the capture video quality.

In this paper, priority scheduling mechanism, and feedback video capture management are introduced for the real-time multiple video player system. The implementation will be constructed in Windows NT environment. We will also study the real-time features provided by Windows NT and discuss on the implementation considerations.

## 2 Systems Architecture



**Figure 1: System Architecture**

Figure 1 depicts the basic architecture of a real-time multiple video system which consists of a video player server, several video capture servers and several displayers (clients). Upon the receipt of a request from a displayer, the video player server selects the video capture servers that the displayer wants to connect with. The video capture servers capture videos from their cameras and then send the video frames to video player continuously. For each video request, the video player server spawns a video handler to handle the request. Each video handler consists of two threads: receive thread and display thread. The receive thread receives video frames from video capture server and put them into video buffer of the handler. The display thread fetches video frames from the video buffer and then passes them to displayer. Finally, the displayer displays the video frames. During playback, all the threads are scheduled in different priorities through the adopted priority scheduling mechanism. When a video player comes to an unendurable status, it will send a feedback signal to the video capture server which will determine whether to reduce the video quality in order to save the bandwidth of the whole system according to the adopted video capturing management mechanisms.

### 3 Priority Feedback Scheduling Mechanism

In order to improve the adaptiveness of the system, we introduce a priority scheduling mechanism in which the priorities of the video handlers will be dynamically adjusted according to the observed status in the playing video. If a video handler comes to a poor status, its priority will be adjusted to a higher level. However, if the video handler returns to a better situation, the priority will be dropped to a lower level.

In the design of dynamic priority scheduling algorithm, an important consideration is that the additional overhead in priority scheduling must be low [1]. Otherwise, the normal processing of the system will be seriously affected. Although different priority scheduling algorithms have been proposed, most of them are complex and require the priori knowledge of the processes. Thus, they are not suitable for our player system which is in a dynamic environment. Instead, a simple priority-scheduling algorithm is more preferable and therefore is adopted in our proposed priority feedback mechanism.

In the design of priority scheduling algorithm, we need to determine an indicator that reflects the status of the playback. In our implementation, we choose the percentage of dropped frames within a fixed period of time as the indicator. If value is high, this means that situation

of the video playback is not good and need to raise its priority. The value of percentage of dropped frames within a fixed period of time is input into a priority mapping function to calculate the priority of the server process.

#### 3.1 Priority Mapping Function

We have designed three priority mapping functions to cater for different service requirements of the clients based on the requesting playback speed of the clients: *linear*, *increasing*, and *decreasing*. In the *Linear* function, the priority mapping of a process is directly proportional to the number of frames being dropped, i.e.,  $\text{Priority}(P) = x + \% \text{ of frames to be dropped in the period}$ , where  $x$  is the default priority of the process. (E.g., the priority of a process when it is at the normal status.)

The problem of the *Linear* function is that it treats all the clients the same way according to their number of frames being dropped. If the play speed of the clients is not the same, linear priority mapping will favour the higher play speed clients as they usually drop more frames as compared with the low play speed clients. As a result, we define two more mapping functions, the increasing function and the decreasing function. In the *Increasing* function, the rate of increase in priority increases with the number of frames being dropped, i.e.,  $\text{Priority}(P) = (\% \text{ of frames to be dropped in the period})^2 / k$ , where  $k$  is a constant. The *Decreasing* function is the reverse of the increasing function in which the rate of increase in priority decreases with the number of frames being dropped, i.e.,  $\text{Priority}(P) = \sqrt{k \times (\% \text{ of frames to be dropped in the period})}$

#### 3.2 Priority thread and Priority Group

As discussed before, each handler has two threads: the receive threads and the display threads. Since the two threads have different functionality and properties, they should have different priority mapping functions. We define two priority groups to categorize these two kinds of threads: the receive priority group and the display priority group as shown in figure 2. All threads in the same priority group have the same priority mapping function. Intuitively, the receive priority group should have higher priorities than the display priority group since receiving video packet is more important.

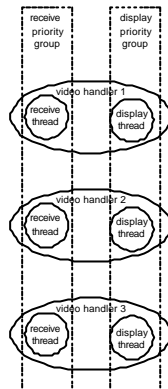


Figure 2: Priority Thread and Priority Group

#### 4 Feedback Video Capture management

If the video player comes to an unendurable status, simply adjusting the priority scheduling may not be enough. We may need to send feedback signals [3] to the server so that appropriate actions will be performed. Like priority scheduling mechanism, percentage of displayed frame video (PD) is designed as our feedback signal argument.

Another question is what action should be taken after a feedback is received. In distributed video transmission and playback, the content of the video (i.e. MPEG video in our case) dominates the bandwidth and resources taken of the whole system. As recalled from MPEG structure, high quality of video playback requires more CPU resource to decode and display it. Thus, if there are not enough resources (e.g. CPU) for the video playback, an alternative to resolve the problem is to down grade the captured video quality (e.g. resolutions) to a lower level. However, down grading of the video quality may affect other video players which also connect to the same capture server. As a result, we need to figure how this action affects the performance of the whole system. We define a performance index which indicates the performance of the whole system.

- Performance index = function(# connected player, # unendurable player, PD of each unendurable players).

Then, we define the captured video quality as:

- Captured video quality = function( Performance index )

The mechanism is designed as:

- When a video player's PD > an Unendurable Factor, it sends its PD as a feedback to capture server.
- Capture server sends requests to all the connected video players for their current PDs.
- Each video player sends its current PD to the capture server.

- Capture server bases on the information to calculate the performance index.
- Capture server bases on the performance index to reduce the capture video quality.
- After the actions are taken, if the video player situation becomes better and the video player's PD < restoring normal factor, it sends its feedback to capture server to restore the original quality.

#### 5 Windows NT's as a Real-Time OS

After the introduction of the priority feedback mechanism, we discuss how it can be implemented in a commercial general-purpose operating system, e.g., Windows NT.

##### 5.1 Priority and Scheduling

The priority model within Windows NT includes 32 priority levels, of which 16 are reserved for the operating system and real-time processes. Each process maintains a private address space to ensure that it will not be interfered by other processes, and each process has a base priority class. Real-time applications can run with a base priority class of 31 (highest priority), 24, and 16. Real-time processes will run at priority 24. Other applications (dynamic classes) have a base priority class of 15, 13, 9 (normal foreground process), 7, 4, 1, and 0.

Each process also has one or more threads associated with it, within the same address space, where each thread represents an independent portion of that process. Each thread has a current priority derived from the process' priority class. By using an application programming interface (API) call that varies up or down from the process' base priority, you can vary the current priority up or down within defined limits. For example, a process running at real-time class 24 can have threads that run anywhere between classes 26-22, depending on their own priorities. These threads will always stay within the real-time priority class.

Threads are scheduled independently by the executive. Associated with the process is a *quantum* (the maximum amount of time a thread can execute before the system checks to see if other threads with the same priority want to execute). In general, real-time processes will have priority over almost all other activities or system events. However, for processes in the spectrum of dynamic classes that are running at lower priority levels, a number of events within the system, such as I/O completion, could cause a temporary priority boost for a thread, giving it priority within a process.

Finally, there is a single system process, within which there can be multiple system threads running. This system process runs all device drivers, the kernel, the executive,

and device drivers. All these components share a single address space, known as the *system space*.

## 5.2 NT's Properties and Limitations

In Windows NT, the priority level of interrupts is always higher than that of a user-level thread, including threads in the real-time class. When an interrupt occurs, the trap handler saves the machine's state and calls the interrupt dispatcher. The interrupt dispatcher among other things, makes the system execute an Interrupt Service Routine (ISR). Only critical processing is performed in the ISR and the bulk of the processing is done in a Deferred Procedure Call (DPC).

DPCs are queued in the system DPC queue, in a First in First (FIFO) manner. While this separation of ISR and DPC ensures quick response to any further interrupts, it has the disadvantage that the priority structure at the interrupt level is not maintained in the DPC queues. A DPC is not preemptable by another DPC, but can be preempted by an (unimportant) interrupt. Hence, the interrupt handling and the DPC mechanism introduce unpredictable delays both for interrupt processing and for real-time applications.

Threads executing in kernel mode are not preemptable by user level threads and execute with dispatching temporarily disabled, but interrupts can occur during the execution of the kernel. As kernel level threads can mask some or all interrupts by raising the CPU current IRQL (Interrupt request levels), the responsiveness at any point in time to an interrupt depends on the mask set by kernel entities at that time, and the execution time of the corresponding kernel entities. Also, since only kernel level threads allowed to mask and unmask interrupts, even an unimportant interrupt can adversely affect a real-time priority user level thread. All of these do not code well for real-time processing. [3,4]

## 5.3 Real-Time Features Provided

A NT process belongs to one of the following priority classes: IDLE, NORMAL, HIGH and REALTIME. By default, the priority class of a process is NORMAL. Processes that monitor the system, such as screen savers or applications that periodically update a display, use the priority class IDLE. A process with HIGH priority class had precedence over a process with NORMAL priority class. The REALTIME priority class is provided as a support for real-time applications.

Windows NT assigns a scheduling base priority to each thread. The base priority is determined by the combination of the priority class of its process and the priority level of the thread. A thread can have any of the following seven priority levels: IDLE, LOWEST,

BELOW\_NORMAL, NORMAL, ABOVE\_NORMAL, Highest and TIME\_CRITICAL. The base priorities range from zero (lowest priority) to 31(highest priority) [4].

Given this priority structure, Windows NT performs priority based preemptive scheduling. Then two threads have the same base priority, a time sharing approach is used. REALTIME priority class threads have non-degradable priorities, while NORMAL and HIGH priorities can be decayed by the NT scheduler [4]

## 6 Implementation Details

Implementation of the real-time multiple video player system is focused on the priority scheduling mechanism. The objective of the implementation is 1) study the real-time features provided by Windows NT; 2) study how these real-time features influences our mechanism; 3) study how real-time win32 API cooperate with multimedia applications.

NT process can be classified as four different priority classes that are IDLE, NORMAL, HIGH and REALTIME [4]. If we develop our application as a multi-process orientated, four priority levels can be assigned to each video handler. However, this may lead to two problems: First, video handlers are needed to be cooperating tightly, hence a lot of resources need to be shared between them. If video handler is created as a new process, context switching between different video handler can result in a large performance down gradation. We also need to handle the IPC between different handlers. This definitely wastes the resources and processing time. Second, if our application is multi-process based, we only have four priority levels. Each video handler only has four choices to have its priority. Such a small scale of priority level is not suitable for a high sensitivity environment. Hence, we need to develop our application to a multi-thread model.

Different threads using the same memory address spaces of the same process, it can solve the problem of the high overhead of context switching and inter-process communication (IPC) problem. However, how can we use them to scale up a more concise priority levels. As recalled from the above section, threads can be assigned to seven different priority level that are IDLE, LOWEST, BELOW\_NORMAL, NORMAL, ABOVE\_NORMAL, HIGHEST and TIME\_CRITICAL. If we want to have the largest different of priority level, we can simply create different video handler as different processes and each process have different display thread and receive thread. As a result, each thread can have a combination of four priority classes and seven threads priority level. A totally twenty-eight priority levels can be delivered. This may sounds good if we want to have as large as priority level as possible but it lead to a same problem that different

video handlers are in different processes and it wastes resources and processing time. It is not worth to implement like this in order to have large different priority level.

After having considered the tradeoffs, we proposed to create only one process, which contain different video handlers. Each video handler has its own display threads and receive threads. Seven different priority levels (IDLE, LOWEST, BELOW\_NORMAL, NORMAL, ABOVE\_NORMAL, HIGHEST and TIME\_CRITICAL) can be assigned to threads. This is a single process multi-threads model which have a disadvantage that only provides seven priority level. However, it saves resources and processing time. It is not perfect but it may be the best configuration due to the weak real-time features provided by Windows NT.

## 7 Performance Measures

### 7.1 Percentage of Displayed Frames

One metric to measure the actual QoS level in a video player system is to measure the percentage of displayed frames (PD). PD should not be confused with play speed, which is also specified in frames-per-second. A valid PD is always equal to or lower than the current play speed.

In the experiment, percentage of displayed frames (PD) is used to measure the performance of the proposed mechanism. PD is defined as:

$$PD = \frac{\sum_{i=1}^n D_i}{n \times D_{total}}$$

where  $n$  is the number of clients.  $D_i$  is the number of displayed frames by client  $i$ , and  $D_{total}$  is the total number of frames in the video file.

### 7.2 Monitor Factor

Video handler monitors the PD of clients in a regular interval. The monitor interval is defined in terms of the monitor factor, and they are defined as:

$$\text{Monitor Interval} = (1 / \text{Frame Rate}) \times (\text{Total Number of Frames} \times \text{Monitor Factor})$$

Monitor Factor (MF) is the percentage of the Total Number of Frames.

That is, if the monitor factor = 10% and total video frames = 100, video handler has a monitor period for every 10 frames time. In the experiment, we will study how the variation of monitor factor affects the system.

### 7.3 Priority Group Mapping Difference

We have two thread groups that are the receive thread and the display thread in a video handler. Recalled from the above section, these two groups should have different priority mapping function since their behavior is quite different. In our mechanism, they are all in linear function but the priority of one group should be greater than the other one. For examples, if the mapping function of display thread is:

- Mapping Function  $_{display} = f(x)$ .

Then mapping function of receive thread should be,

- Mapping Function  $_{receive} = f(x) + k$   
where  $k$  is positive or negative constant

As a result we define Priority Group Mapping Difference (PMD) that is a priority difference between receive thread and display thread as:

$$PMD = \text{Priority}_{receive\ thread} - \text{Priority}_{display\ thread}$$

That is, when  $PMD = 1$ , priority of the receive thread always greater than the priority of the display thread by one. Also, when  $PMD = -2$ , priority of receive thread always smaller than priority of display thread by two.

## 8 Experiment Result Discussion

### 8.1.1 Impact on Monitor Factor

In this set of experiment, we vary the value of the monitor factor in each experiment and see how it affects the mean of percentage of frame displayed (PD). We also recorded the standard deviation of the PD of the five playing videos. There was a test that no monitoring as a control experiment for comparison purpose.

Monitor Factor	% Frame Displayed (PD)	Standard Deviation (SD)
5	31.80	14.62
10	50.20	18.07
20	53.10	12.27
30	50.90	17.26
50	40.60	37.17
No monitor	25.02	37.48

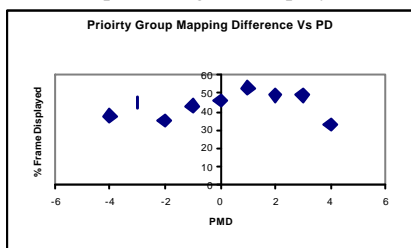
Table 1: Impact on Monitor Factor

From table 1, we observe that when the monitor factor is 20, the value of PD is the best. If the monitor factor is small (e.g., equal to 5), the PD is low. However, when the monitor factor is large, the PD value is also low. During the monitoring, the priority of the thread is set according to the current PD value. From the PD, it calculates and set new priority of the thread. These operations may spend a lot of CPU time. As a result, a frequently monitor may degrade the system performance significantly. On the other hand, when there is no monitoring in the experiment,

the performance is the lowest. When there is no monitoring, all the threads were only scheduled in a round robin manner. The system can not adapt to the system environment dynamically.

### 8.1.2 Impact on Priority Group Mapping Difference

In this set of experiments, the client requests video files with 30 fps and the monitor factor is 20. That meant the video handler monitors the system five times the display frame interval during the whole experiment. We change the thread mapping difference from -4 to 4 to see how it affected the percentage of displayed frames.



**Figure 3: Priority Group Mapping Difference Vs PD**

From figure 3, we can see the left hand side represents a negative PMD (priority group mapping difference) that means the priority of the display thread is always higher than the receive thread. The overall performance in the right region is better than in the left region. If the priority of the display thread is higher than the receive thread, the CPU always serves the display thread. Since the display thread consumes a lot of CPU time and the receive thread is always in a lower priority than the display thread, when there are video frames arrived and wanted to be putted into the buffer, they do not have time windows to do so. Hence, the video frame needs to be dropped. As a result, the performance is not good. On the other hand, if the priority of the receive thread is always greater than the display thread, the circumstances will be better. The receive thread is an I/O bounded thread and it consumes CPU time not too much, raising its priority does not cause the display thread to wait for CPU for a long time.

## 9 Conclusions

Real-time multiple video player system is introduced in this paper. A priority scheduling mechanism and a feedback video capture management are proposed. The priority scheduling mechanism dynamically adjusts the priority of the video handler in the video player according to the observed status in the playing video. The feedback video capture management can change the quality of the video in response to the system changes in order to save the system resources. The objective of both mechanisms is to provide a good quality of services while multiple videos are played in client video player while a dynamic environment. Simulation programs are implemented to study the priority scheduling mechanism. Windows NT is chosen as the platform and we also study on its real-time features. Experiment results show that the priority scheduling mechanism can effectively improve the system performance in terms of percentage of displayed frames.

## References

- [1] Shanwei Cen, Calton Pu and Richard Staehli, "A Distributed Real-time MPEG Video Audio Player", in *Proceedings of the 5th International Workshop on Network and Operating System Support of Digital Audio and Video (NOSSDAV'95)*, April 18-21, 1995.
- [2] Kam-yiu Lam, Chris C.H. Ngan, Joseph K. Y. Ng, "Using Software Feedback Mechanism for Distributed MPEG Video Player Systems", *Computer Communications*, vol. 21, pp. 1320-1327, 1998.
- [3] Krithi Ramamritham, Chia Shen, Oscar Gonzalez, Subhabrata Sen, Shreedhar Shirgurkar, "Using Windows NT for Real-Time Applications: Experimental Observations and Recommendations", in *Proceedings of the Fourth IEEE Real-Time Technology and Applications*, Denver, June 1998.
- [4] Microsoft Developer Network Library, Microsoft, July 1999.
- [5] Cripsin Cowan, Shanwei Cen, Jonathan Walpole and Calton Pu, "Adaptive Methods for Distributed Video Presentation", *ACM Computing Surveys*, volume 27, number 4, pp. 580-583.
- [6] A. Vogel, Brigitte Kerherve, Gregor von Bochmann and Jan Gecsei, "Distributed Multimedia and QOS: A Survey", *IEEE Multimedia*, volume 2, number 1, pp. 10-18, 1995.
- [7] Shuichi Oikawa and R. Rajkumar, "A Resource-Centric Approach to Multimedia Operating Systems", in *Proceedings of Workshop in Multimedia Resource Management*, December 1-2, 1996.