In
Borko Furht (Ed.): “Handbook of Multimedia Computing”, CRC Press, Boca Raton, FL, USA, 1998

Chapter 41

RESOURCE MANAGEMENT IN MULTIMEDIA SYSTEMS

Lars C. Wolf
Darmstadt University of Technology
Dept. of Electrical Engineering & Information Technology
Industrial Process and System Communications
Merckstr. 25 • D-64283 Darmstadt • Germany

Abstract: Multimedia systems must be able to support a certain Quality of Service (QoS) to satisfy the stringent real-time performance requirements of their applications. Resource management systems contain mechanisms to administer and schedule system resources to give time-critical multimedia applications access to all necessary resources when needed so that their QoS requirements can be met. Static resource management systems use and extend techniques developed in the field of real-time systems. They perform QoS calculation and resource reservation based on the requirement specifications given by applications during their setup, and schedule the resources in such a way that processing deadlines are met. While this approach can offer strong guarantees for the application’s performance, it has the drawback that it is difficult to determine the amount of resources needed in advance and that it cannot easily cope with a change in the set of running applications during the run-time of an application. Dynamic schemes, also called adaptive resource management, extend the static approach by methods for resource usage monitoring and renegotiation. They gain more flexibility in exchange to the firmness of the reachable QoS. In this chapter we describe the purpose of resource management systems and give an overview about the principles.

1. INTRODUCTION

Multimedia applications integrating audio and video data into distributed computer systems have become possible due to the advances in computer and communication technology such as increased processing speed, network bandwidth, and storage size.

In comparison to media traditionally handled in computers, e.g., text and graphics, audio and video have different characteristics. Humans perceive audio and video as continuously changing, therefore, they are also called *continuous media*, this contrasts with discrete media such as text and graphics. Since audiovisual information is time critical, the processing requirements of such continuous-media data are different from discrete-media data because the value of the applications processing depends not only on the accurateness of the computations but also on the time when this processing has finished. In addition to the time criticality, the processing demands of digital audio and video data are typically large. The treatment and finally the display of audiovisual data must be done with a certain quality in order to provide for a satisfying overall presentation.

An application must operate with a certain *Quality of Service (QoS)* to fulfill its task. The required QoS depends on various issues, for example, the used media (video, audio, etc.), the coding format used to encode the data, the application and the type of the application. For instance, the QoS of a video conference is different from that of a video retrieval application, since the dialogue-mode communication of a conference requires a short delay which is not as important for playback applications.

Processing within multimedia systems is performed in various layers: in application-layer code, in the operating system, in the communication system, etc. And each of these layers can be constructed by layering. In each of these several layers, a different notion of QoS exists, e.g., QoS at the application layer is usually described at a higher level than QoS at the network layer of a communication system and different terms are used. However, the QoS parameters, *bandwidth*, *delay*, and *loss* are used in all layers, sometimes in conjunction with other parameters. The applications specify their requirements in higher-layer QoS terms, these are potentially translated in several steps towards more system-oriented parameters, which are used to control the system parameters.

A *resource management system* (e.g., [17]) provides the means to offer QoS to multimedia applications, e.g., so that the participants in a video conference do not experience large delays or low video frame rates during their interaction. These mechanisms administer and schedule system resources to give time-critical multimedia applications access to all necessary resources when needed so that their QoS requirements can be met. These mechanisms must address the following issues:

- QoS calculation – to check whether the QoS demands of an application can be satisfied.
- Resource reservation – to reserve an amount of resources according to the given QoS guarantee.
- Resource scheduling – to enforce that the given QoS guarantees are satisfied by appropriate scheduling of resource access.

In addition to such static functionality, adaptive resource management systems (such as, e.g., [7]) offer also mechanisms which help the applications to adapt their behavior and their resource demands in case the available resource capacity changes or the reserved resources are insufficient. The former case can occur if the user decides to execute an additional application, the latter case can be due to an incorrect specification of resource requirements. We will discuss adaptive resource management in Section 5.

Multimedia applications have to deal with the processing of time-critical, large volume, audiovisual data on one hand, and on the other hand they have to provide a nice looking, easy to use interface to the user. To ease the task of application programmers, the processing of the continuous-media streams can be encapsulated in specific software modules, as has been proposed by several research groups (e.g., [5], [6]). This allows for code reuse and simplified application development. Examples for such building blocks are modules for decompression, for mixing streams, etc. An application specifies which modules are needed and how they have to be connected, forming a directed (not necessarily linear) graph of modules. As an example, Figure 1 shows an application consisting of three modules. If such an approach is used, the resource management must be devised for such a module-based system.

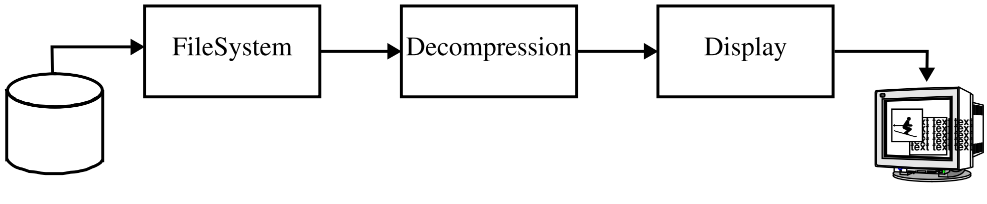

Figure 1: Application containing three modules.

2. RESOURCES AND RESOURCE MANAGEMENT

2.1 RESOURCES

Resources are all the entities which participate in the overall task of the application, i.e., all parts which are required for processing, for transmitting, and for presenting the data. This comprises resources on the path from source(s) via networks including routers and switches to sink(s), both in the local systems and the network.

Resources can be classified whether they are (1) active or passive, e.g., CPU vs. memory, (2) exclusive or shared by several processes at a time, and (3) single or multiple, e.g., how many instance of a resource are available in the system.

Each resource has a certain capacity which can be used by applications to perform their task. The capacity of a resource is administered by the resource management system.

A resource management system manages all the resources which are critical for the execution of the continuous-media data processing of an application, e.g., CPU time, network bandwidth, and memory. This comprises all resources which are involved in the overall task of the application, i.e., those used by the application for local processing, those to move the data to or from the transport system interface and those needed by the transport system to transfer the messages across the network (illustrated in Figure 2). This, therefore, includes all the limited *system resources* through which a media stream passes which are

- bus (resp. switch) bandwidth, e.g., for the movement of data from memory to network adapter,
- I/O devices, e.g., special (de-)compression boards,
- external storage, e.g., hard disks, with file systems,
- network adapters and network resources to transfer packets from one node to another,
- processors to execute application and system software,
- main memory to hold the application and system code and buffer space for the data.

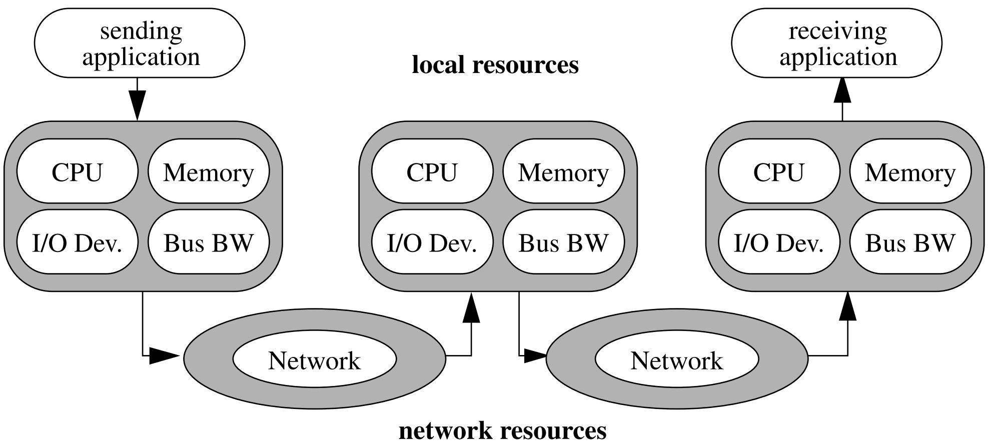

Figure 2: Resources to be managed.

Most work deals with the management of the processors, the network resources, and the file systems, only few work has been done on the management of the bus bandwidth. Within this chapter we will illustrate general methods of resource management and discuss mechanisms for processors and the main memory as examples for resource managers which administer a particular resource.

2.2 RESOURCE CAPACITY

To deliver a particular level of QoS to an application, the system must possess resource capacities which are at least as large as the requirements. In addition to the overall available capacity, these resources must be scheduled in such a way that they are available for the application when needed. Many of today's communication and computer systems offer sufficient resources to handle some continuous-media streams, but the quantity and quality of such streams is still limited since the resources are limited. The "window of scarcity" postulated by Anderson et al. in [1] illustrates this (Figure 3 shows an adapted version).

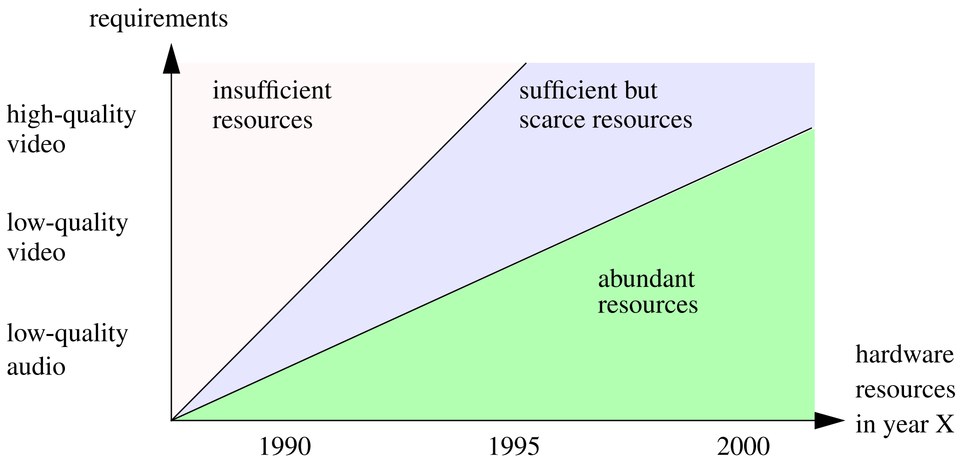

Figure 3: "Window of Scarcity" (adapted from [1]).

The available resources are *insufficient* to provide acceptable service at a specific time and for certain application types (left side Figure 3). Due to ongoing improvements in technology, system resources become sufficient for new applications, however, the available resources are *scarce*, i.e., they must be administrated and scheduled carefully to offer the desired QoS (middle part of Figure 3). After further technology advances, resources are abundant with respect to a particular application, i.e., the service can be offered without specific management mechanisms (right side of Figure 3).

Depending on the particular notion of 'quality', the number of concurrent users and the number of concurrent continuous-media streams of one user, the areas are reached at a different point in time. Nevertheless, at least in the near future, distributed computer systems will have only sufficient but scarce system resources available for, e.g., the processing of multiple continuous-media streams. Due to increased user expectations, e.g., larger video frame sizes, etc., it takes more time until the region of abundant resources is reached, if ever. Furthermore, also in the future, when large system resources might be available for processing and transmitting of data, a time-critical application must be 'shielded' against non-real-time and other real-time applications so that they cannot inhibit its real-time processing. And for shared systems offering services to several users simultaneously, e.g., video-on-demand servers, means to manage the available system resources are needed in future as well because providers of such systems want as efficient resource use as possible, i.e., to serve as many clients using as few systems as possible.

The mechanisms for QoS calculation, resource reservation and scheduling rely on the knowledge of the amount of resources required for the execution of a particular multimedia application. Therefore, methods to determine the resource requirements are needed to successfully apply the resource management mechanisms mentioned above. Because resource requirements depend on the particular computer system and its configuration, the techniques to find out the resource demands of an application must allow for flexible and automatic measurements, e.g., at installation time or even at run-time but without delaying the application start-up which would be annoying to most users. If the application has been constructed using a module-based system as mentioned above, its resource requirements consist of the requirements of each single module. These demands must be collected and combined by according mechanisms.

2.3 RESOURCE MANAGEMENT FUNCTIONALITY

A multimedia application which desires to get support from the resource management system specifies the overall QoS demands it has which in turn results in requirements on the various resources. The resource management system, for instance HeiRAT (**Hei**delberg **R**esource **A**dministration **T**echnique) [15], [16], checks whether this additional workload can be handled and what kind of service it can offer. Furthermore, it reserves the required resources and schedules incoming processing requests accordingly. After the handling of the workload has ended, the reserved resources are released. Hence, it offers the following functionality for both active and passive resources:

- *Admission test*: When a new multimedia stream shall be established, it is checked whether enough free resource capacity is available to handle it. This decision is influenced by the QoS guarantees already given to other streams; with static resource management these guarantees must not be violated by adding the new stream.
- *QoS calculation*: Every resource computes the QoS it can provide for the new stream.
- *Resource reservation*: The resource capacity is reserved that is required to provide the QoS guarantee.
- *Resource scheduling*: Resource access is coordinated so that the respective QoS guarantees of all streams are satisfied.
- *Resource deallocation*: The reserved resources are released.

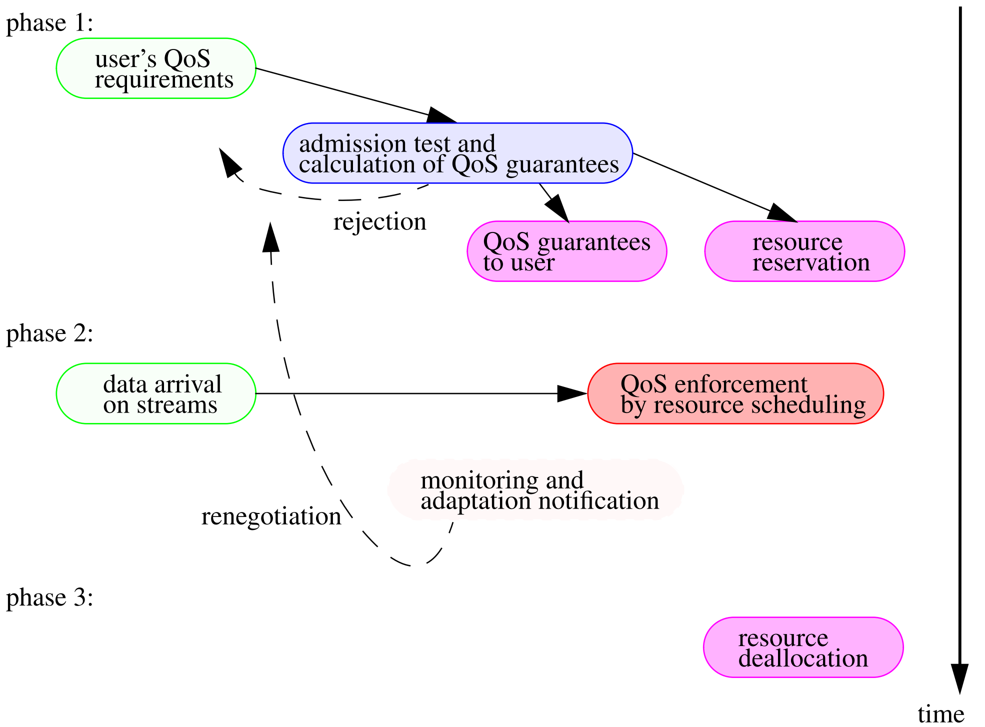

Figure 4: Resource management phases.

As illustrated in Figure 4, these functions can be grouped into three phases. The first three functions belong to the *set-up* or *QoS negotiation phase*. In that phase, applications specify their QoS requirements which are used for the admission test (e.g., the schedulability test for the CPU) and the QoS calculation. This results either in a resource reservation or in the rejection of the request in case that the QoS cannot be met. The second phase is the *transmission* or *QoS enforcement phase*. In this phase, after the successful establishment of a stream, the resources are scheduled with respect to the given QoS guarantees. The usage of the resources is monitored and, if necessary, the necessity to change the behavior is indicated to the applications. In order to perform these last two steps, functions are necessary during the processing of a stream for

- *Resource monitoring*: observe the resource usage of the applications, and monitor the overall load put onto the resource;
- *Adaptation*: inform applications that their resource usage must change and that they should renegotiate their resource reservations.

These functions are especially needed if an adaptive resource management system is used. Finally, in the *deallocation phase*, the reserved resources are released.

In the set-up phase, resource management systems typically offer several options by which applications can specify their QoS requirements. For instance, QoS values can be given in terms of maximum end-to-end delay, minimum throughput needed, and reliability class defining how the loss of data shall be treated. An application can select one of the QoS parameters for optimization by specifying an interval from *desired* to *worst-acceptable* values. For example, a video application might request a throughput between 15 and 30 video frames per second, indicating that video quality would not be acceptable with less than 15 frames, but that more than 30 frames are never needed. The resource management system will then return the best QoS it can guarantee within this interval and make the corresponding reservation. If even the lower bound cannot be supported, the request is rejected in a static resource management system. In an adaptive system, the resource management may ask already running applications to reduce their resource requirements and reservations to provide sufficient spare capacity to serve this additional, new application.

In the transmission phase, data are processed and transmitted according to their urgency. Schedulers handle time-critical multimedia streams prior to time-independent data. They exploit properties of the underlying resources, for example, they are based on the operating system priority scheme for CPU scheduling or the MAC priority scheme of the network.

In the deallocation phase, after the transmission has finished, the allocated resources such as CPU or buffer space must be released. This can be initiated by the application which performed the reservation or by the system, e.g., due to a system failure or a permanent mismatch between negotiated and delivered QoS or resource usage.

The reservation can be made in different ways. One possibility is to distinguish between a pessimistic and an optimistic manner. With the pessimistic approach, the resource capacities are reserved for the worst case, i.e., the maximum demand a stream may have during its lifetime. The advantage of this scheme is that it avoids conflicts and offers deterministic guarantees. However, reserving extensive amounts of capacities for such peak requirements can be rather costly and leads to the underutilization of resources if there is a significant difference between peak and average data rate of a stream. A cheaper alternative is followed by the optimistic approach where resources are reserved on average workload, i.e., they can be slightly overbooked. This implies that while QoS requirements will be met most of the time, occasional QoS violations may occur. Applications which perform optimistic reservations must be aware of temporary resource conflicts, hence, they must be ready to cope with them. This can, therefore, already be considered as a (simple) adaptive scheme.

2.4 RESOURCE RESERVATION PROTOCOLS

In distributed multimedia systems, continuous-media streams are transmitted across a multi-hop network. On their way they are handled by multiple system resources. To obtain an end-to-end QoS guarantee, from source to destination, reservations on all the individual resources handling the stream must be made and the according guarantees must be aggregated. This requires a *resource reservation protocol* such as RSVP [3] to exchange and negotiate QoS requirements across system boundaries. The fact that the network is one of the resources to be managed makes it necessary to integrate the resource reservation protocol with the network layer of the transport system; higher layers have no information about the different resources in the network.

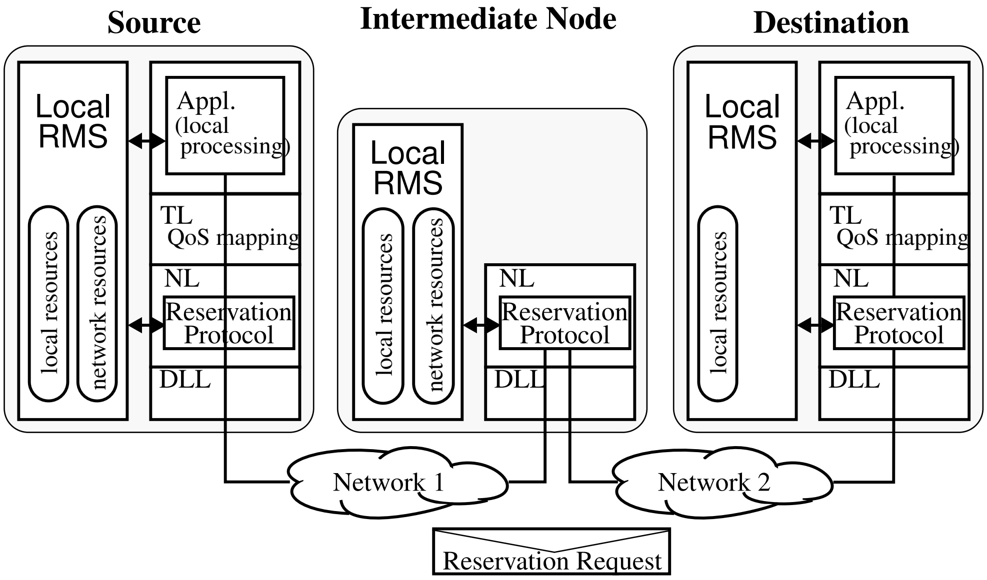

Figure 5: Distributed resource reservation.

Depending on the reservation protocol, the reservation can be made in a receiver- or a sender-oriented way. In both cases, several entities participate in the end-to-end QoS negotiation: sending and receiving applications, agents executing the resource reservation protocol, and local resource managers (Figure 5).

Applications specify their QoS requirements which are possibly mapped by the transport layer on a QoS request in terms of network layer units due to packet segmentation. This request becomes part of a reservation message. Each local resource manager on the path receiving the message checks whether its available resource capacities can serve this request and reserve the resource capacities needed.

2.5 RESOURCE MANAGEMENT SYSTEM STRUCTURE

The resource management contains components used in the enforcement phase, i.e., to schedule the access to the system resources and to monitor their usage, and modules needed in the negotiation phase, i.e., to perform the throughput test, the QoS computation, and the resource reservation. To be able to perform the scheduling, the enforcement components must be located near to the affected resources. The components used during the negotiation must be structured in such a way that the data structures of the resource management system are protected all of the time. To avoid any inconsistencies due to malfunctioning programs, the resource management system must be structured as a daemon which offers the management functions via an IPC interface.

Using IPC mechanisms which can handle remote IPC as well, clients may also reserve resources at a non-local resource management agent, either for all resources or for specified resources only while other resources are reserved at the local agent. It can be useful to reserve all resources of one kind at one agent because this leads to better system knowledge and therefore to better resource allocation decisions. For instance, systems on a shared medium network such as Token Ring can use their local agent to reserve the local resources (CPU and memory) but reserve network bandwidth for the Token Ring only at one central agent which leads to global knowledge at that agent. Drawbacks of this centralized approach are that it is not scalable and that it represents a single point of failure.

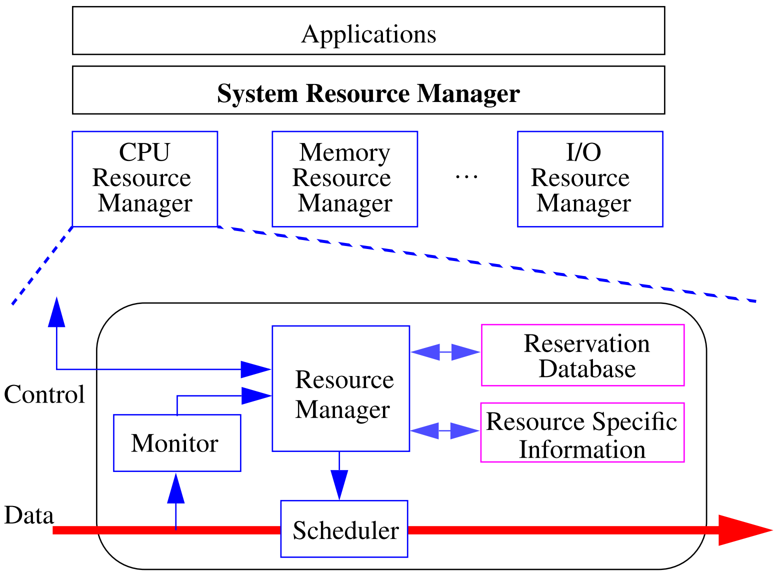

Figure 6: Resource Manager.

The overall resource management consists of a ’System Resource Manager’ which controls the single ’Resource Managers’ for the various resources within the particular system (Figure 6). It contains algorithms for admission control and policy control – to ensure that sufficient resources are available to handle the data stream and that the particular data stream (and the user associated with this) is permitted to use the resources, respectively. Each resource manager keeps information about the characteristics of the resource and its actual reservations. The scheduler selects which packet gets access to the resource. The monitor observes the resource usage by the applications and the overall load of the resource.

At least parts of a resource manager are either directly located inside the operating system or interact tightly with it. For instance, the CPU resource manager cooperates closely with the CPU scheduler of the operating system, e.g., by using the real-time priorities the latter may offer.

3. PROCESSOR MANAGEMENT

To ensure that applications are indeed served with the promised QoS, it must be controlled which work item is processed by a certain resource at a given time. Otherwise it cannot be guaranteed that deadlines can be met. Using resource management techniques, scheduling mechanisms are applied which ensure that an application task gets access to the resource when needed. In case that not all applications can be served, decisions based on importance or criticality are made.

3.1 REQUIREMENTS

The processing of audio and video data is usually performed in a periodical manner due to the periodicity of these continuous-media data. These operations must be finished within certain deadlines to serve the real-time characteristics of these media. Sometimes, multimedia systems for single-user (and especially for single-tasking) machines provide only simple mechanisms to provide time-based operations, e.g., for delaying program execution, but no real-time support. For these systems, it is often argued that this is sufficient since the CPU is used mostly for the multimedia application during its run time and if the user has another time-consuming application running, it is easy for the user to abandon that application. This approach is not satisfying even for single-user systems and falls short for multi-user and server systems such as video-on-demand servers. For such systems, the assumption that bothering applications can simply be stopped is already not valid. Other user applications can disturb multimedia applications in such a way that the QoS falls below an acceptable level.

Real-time CPU scheduling techniques which serve multimedia application processing with respect to their time-criticality provide a solution to these problems. However, the purpose of multimedia systems is the integration of continuous-media *and* discrete-media data into computer systems, hence, multimedia applications rely as much on the processing of media like text and graphics as on that of audio and video. Thus, operations on continuous-media data should not lead to a starvation of the processing of discrete media. Neither should priority inversion occur, i.e., where the handling of discrete-media data disturbs the processing of continuous-media data (in a more general way: a low-priority application should not block a high-priority task).

If the real-time processing is achieved by the usage of preferred operating system priorities, an incorrect implemented application may use 100% of the CPU and stop the system from doing anything else. To avoid this, commercial workstation operating systems which offer such priorities do not allow arbitrary users to run their applications with such a priority. Instead, only a privileged system administrator is able to use them. In order to give users the ability to execute continuous-media applications with appropriate processor scheduling, specific real-time scheduling and monitoring servers can be implemented which control the usage of these priorities by performing admission control, perhaps policy control, and by checking that no process uses (substantially) more CPU than it has reserved.

Traditional real-time methods used in command and control systems in areas such as factory automation, plant and aircraft control, etc. have often stringent demands, e.g., that deadlines are met and that fault tolerance and security are ensured. For most multimedia systems (unless they are used in similar mission-critical scenarios), the requirements towards these issues are less critical. For instance, for many multimedia applications it is not a severe failure to miss a deadline as long as it does occur often. Therefore, multimedia systems are often considered as *soft real time*.

3.2 COMMON METHODS

The processor scheduling mechanisms used for multimedia systems are often similar to the methods derived within the real-time systems field, perhaps modified to provide for adaptive behavior. Especially the well-known *rate monotonic* (RM) and *earliest deadline first* (EDF) algorithms [11] are often used to schedule the processing of periodic, continuous-media data.

EDF scheduling assumes each task to have a deadline at which its processing must be finished. The task with the earliest deadline among the waiting tasks is executed first. RM scheduling is defined in the context of tasks that require CPU processing periodically. Here, the task with the highest rate (i.e. the smallest period) is given the highest priority.

RM is an optimal static technique. As a static scheme, it assigns the priorities to each task of the considered task set once at the begin of the processing of the set respectively at the application establishment time. Priorities are not changed dynamically during application lifetime but only when the task set changes, e.g., due to a newly arriving application with its associated task(s). RM is optimal in the sense that if a task set can be scheduled by any static algorithm, it can also be scheduled by RM.

EDF is an optimal, dynamic scheme. Dynamic means that it schedules every instance of each incoming task independently according to its demands. Hence, the processing order between the various tasks may change permanently during the lifetime of the tasks. If a scheduler, which follows the EDF scheme, is implemented by mapping deadlines on operating system priorities, the task priorities may be rearranged frequently; inducing some additional load into the system. EDF is optimal in that sense that if a set of tasks can be scheduled by any priority assignment scheme, it can also be scheduled by EDF.

For both schemes exist a simple schedulability test which decides whether a set of tasks can be scheduled, i.e., whether the processing of these tasks can always finish within the deadlines. As has been shown in the classical work from Liu and Layland [11], a new task can be accepted (i.e., no overload condition occurs) if the following inequality holds:

$$\sum_{i=1}^{n} R_i \cdot P_i \leq U_n$$

In this inequality, the index i runs through the task set $\{T_1, \ldots, T_n\}$ containing all existing real-time tasks and also the new task. R_i denotes the rate of task i, P_i its processing time per period, and U_n is a non-negative real number representing a schedulability bound. The meaning is that if the sum on the left hand side, i.e., the load generated by all real-time tasks, does not exceed U_n, the processing of all instances of all tasks is guaranteed to terminate within their respective deadlines $1/R_i$. If the sum is greater, this may still be the case, but no guarantees can be given with this test.

For EDF scheduling, the limit for U_n is 1, i.e., as long as the total workload is less than the overall capacity of the processor, it can be guaranteed that all deadlines will be met. For RM scheduling, the limit for U_n for any task set (with arbitrary rates) is

$$U_n = n(2^{1/n} - 1)$$

which approaches ln(2) ($\approx$ 0.69). It should be noticed here that the schedulability boundary U for RM scheduling can be relaxed in certain cases. If the periods of the tasks have a certain ratio, U can be larger than ln(2): E.g., if the periods are (integer) multiples of the smallest period in the task set, then $U = 1$ can be chosen. Also, it has been shown in [9] that the maximum CPU load which can be accepted for RM scheduling is in the average case notably larger than ln(2).

However, the restriction of the maximum CPU utilization U for multimedia processing scheduled by the RM algorithm to a value smaller than 1 is not such a strong limitation as it might seem. It is a limit for the real-time task set only and not for the total CPU utilization (as the sum of real-time and non-real-time processing). And in most cases, some CPU capacity has to be left for tasks performing other than multimedia related processing anyway, e.g., for control operations, to avoid starvation of such non-real-time tasks. Thus, in general, also for EDF it is advisable to restrict the utilization U to values smaller than 1 in order to provide some residual CPU capacity to other non-multimedia tasks.

3.3 LIMITATIONS AND EXTENSIONS

The discussion above assumed that the processing of the tasks can be preempted, i.e., that the currently executed task is suspended if a task with higher priority becomes 'ready to run'. Nowadays, for most operating systems this assumption is true for processor scheduling. This is, however, not always the case for the scheduling of other resources, e.g., network access, because there the processing cannot be suspended for a while and resumed later without changing the semantics of the operation. A non-preemptive scheduling algorithm can be used in such a case, e.g., as discussed in [14]. But, unfortunately, the achieved processor utilization is much lower in comparison to a preemptive scheme.

The rates R_i and the processing time requirements P_i are needed for the schedulability test. While the rates can be usually derived directly from the properties of the application, the processing times have to be measured. For a module-based application, the processing requirements can be determined for any single software module and these values can be gathered and combined to calculate the overall P_i value [17]. For applications which are not built in this way, the processing times can be measured by executing the application several times with varying input data. The measurement of these processing times is not trivial because they depend, e.g., on the used hardware platform, the installed system software, and the input data – additionally, they are influenced by other concurrently executing applications. Moreover, these times can vary significantly, e.g., consider the processing requirements for an MPEG compression which encodes an I-frame in period k followed by an B-frame in period $k+1$ where significant time is needed for the determination of motion vectors.

Algorithms such as RM need the time requirements P_i for the worst case to ensure the schedulability. If the average execution time is significantly lower than the worst-case time, the resulting CPU utilization is low. Furthermore, as said above, it is difficult to determine the processing times in general and the maximum of that in particular. Hence, it might happen that more time is needed than specified. To handle such situations, extensions to this algorithm have been developed. One approach is the division of the overall task into a mandatory part and an optional part [12]. After the processing of the mandatory part has been finished, an acceptable result has been gained which can be refined by processing the optional part. The mandatory part can be scheduled by RM, various policies can be applied to the scheduling of the optional part.

Real-time systems which contain not only periodic but also aperiodic tasks must be able to schedule both types of tasks. One approach is to apply a *sporadic server* for the processing of aperiodic requests. This server has a budget of computation time which is refreshed a specified time after it has been exhausted. This budget is used to process aperiodic requests with a certain, specified priority; if the budget is exhausted, the processing is done with a background priority.

Nevertheless, the rate-monotonic algorithm is often used for the scheduling of multimedia applications because it is a simple method, it avoids large administrative overheads, and it provides a controlled system behavior in case of (unforeseen) overload, the tasks with higher rates will be served first. Additionally, if rates are mapped to operating system priorities, a classification method can be introduced which distinguishes the tasks not only with respect to their rates but also among a further 'importance' scheme. Tasks which perform reservations based on worst-case assumptions (needing deterministic guarantees) get a higher priority than tasks with weaker reservations followed by non-time-critical tasks.

3.4 TASK DEPENDENCIES

It occurs often that a task which finishes its periodic processing of time-critical data gives its results to another task to perform further operations on these data or that the task uses the services provided by another task during its own processing of the data. This second task can be an application specific task, e.g., the total application has been split into parts, or a system provided task. Problems can occur if no coordination among the priorities of the participating tasks is done, e.g., if the follow-on task is executed without an according priority, the real-time characteristics of the first task are discarded. Such a situation can occur especially within microkernel systems, but also with other systems, where much of the processing is performed in user-level server tasks (e.g., a server to process incoming network packets).

One example is the X Windows System – here, the X server task displays the images which it receives from a video playback application to present them to the user. Even if shared memory between server and client is used, a non-real-time X server can introduce deadline violations, especially, if it is single threaded and several requests from other applications have to be executed as well. For simple scenarios, tests showed that just increasing the priority of the X server slightly (e.g., via UNIX 'nice' mechanism) can be sufficient. A better solution would be the provision of a real-time X server which allows the specification of processing requirements for requests.

For the problem of displaying video data via the X server, a special solution can be devised. I.e., a mechanism which allows a user program to bypass the X server by writing directly to a specific area on the screen (the display adapter memory). This could be implemented in a way that the window manager allows the user program to write to the particular memory area where its window is mapped by attaching the memory to the program's address space via a special system call, other memory areas may still be protected. Any change in the visibility, size, or location of the window is known inside the window manager which can change or withdraw the memory from the program's address space accordingly.

A more general approach is the 'transfer' of a CPU reservation and the according priority to the server process as suggested in [13].

4. MEMORY MANAGEMENT

Memory has always been a scarce resource. Virtual memory mechanisms (*swapping* and *paging)* have been applied successfully to give applications the illusion that they are working in a world not constrained by memory sizes. These mechanisms swap parts of the data (which has not been used for a while) from main memory to an external storage device, typically a disk. Into this empty area other data, which has been requested by an application, is swapped in from the external storage. If the data which has been swapped out is needed again, the application which tried to access that data generates a 'page fault' and this data is swapped in again. Nowadays, not only applications but also parts of the operating system kernel can be swapped. Swapping operations take some time, depending on the used disks, interfaces, etc.; yet, this is acceptable for non-time-critical applications – their run time is increased but their semantics are not changed.

Within multimedia systems, as in other systems as well, main memory is needed for several purposes:

- to store the code of the applications and the system components such as the operating system kernel,
- to store data structures used to hold, e.g., the state of this software,
- to store the data on which the processing is done, e.g., a video frame.

In opposite to non-time-critical applications, swapping operations should not be applied to memory areas used by continuous-media applications. If a page fault occurs (due to an application accessing a swapped out data area) it takes too much time to transfer this data from an external storage device such as a disk to main memory. Furthermore, such delays introduce large variations into the processing times of applications which must be avoided as well. To avoid swapping operations on memory areas touched by the processing of continuous-media data, these data areas can be *locked* or *pinned* into memory. A problem is that the amount of memory which must be pinned can be quite large because not only the application code performing, say, the video decompression, but the functions used by it inside libraries, operating system kernel, etc. – everything which is used by that code – must be pinned as well. This is not always possible and has also the drawback that pinning large memory areas reduces overall system performance. Further, it is also contrary to the trend in workstation operating systems to provide for the swapping of kernel code.

Continuous-media data has typically large space requirements which also implies large data movement costs. Therefore, it is important to handle continuous-media data carefully and avoid unnecessary physical data movements. Similar to protocol processing, where headers are prepended by the sender to the actual data before transmission which are removed by the receiver, multimedia systems require operations for concatenation and segmentation of data, etc. To handle the data efficiently, without copy operations, buffer management schemes can be applied which use, e.g., scatter/gather techniques.

Such approaches have been developed already over a couple of years as support infrastructure for the implementation of communication systems. Examples are the mechanisms used in the x-kernel and mbuf scheme used in BSD UNIX operating systems. The latter is widely used in kernel space protocol implementations. Messages are stored in one or multiple chained mbufs. Each mbuf consists of offset fields, pointers, and a small data area internal to the mbuf. To store larger messages, a memory block of fixed size can be attached to the mbuf. Additionally, mbufs can be chained together using another pointer. This way, headers can be prepended to data without copying the data.

These management schemes have been designed for kernel level implementations only. This means that data must be copied between kernel and user level to exchange the data with applications. Alternatively, the memory areas holding the data areas may be remapped by virtual memory operations, e.g., from the kernel into the applications address space. Another issue is that such systems do not provide for the reservation of buffer space areas for certain streams but serve all requests (non-real-time and real-time) from the same pool.

In traditional, discrete-media data processing applications, the application executes a system call to read data, the kernel performs the actual steps to get the data from the device, and copies (or remaps) the data to the application's address space; then the application operates on the data, and finally writes it via the kernel (potentially copying the data again) to the output device. This structure applies for many multimedia applications which perform operations on the continuous-media data as well. However, for several multimedia systems, especially for video servers, there are no data manipulation operations or other 'application' steps performed on the audiovisual information. Instead of this, a more simplified model applies where the application reads the data from one device, e.g., disk, and writes it to another device, e.g., the network, without performing any modifications to the data – the application adds no value. The performance of this continuous-media data stream and the system as a whole is degraded due to the necessary context switches and the (potentially) needed copy operations. Future systems will improve this by offering a different 'streaming mode', several approaches have been suggested by researchers, see e.g., [4].

Hereby, the data flows directly from the source device to the sink device in an application specified manner. This can be achieved in two different ways. One approach is that new system calls (`read_stream`, `write_stream`) are used which read the data from a device into a kernel buffer (and leave the data inside the kernel) and write it from that buffer to a device respectively, the application is responsible for the timing of these I/O operations. The other approach, 'kernel streaming', is to create a new kernel thread per stream which performs the read and write operations; the application specifies the timing of the stream and the thread ensures that this is met. The role of the application is mainly to control the thread. Figure 7 illustrates the different styles.

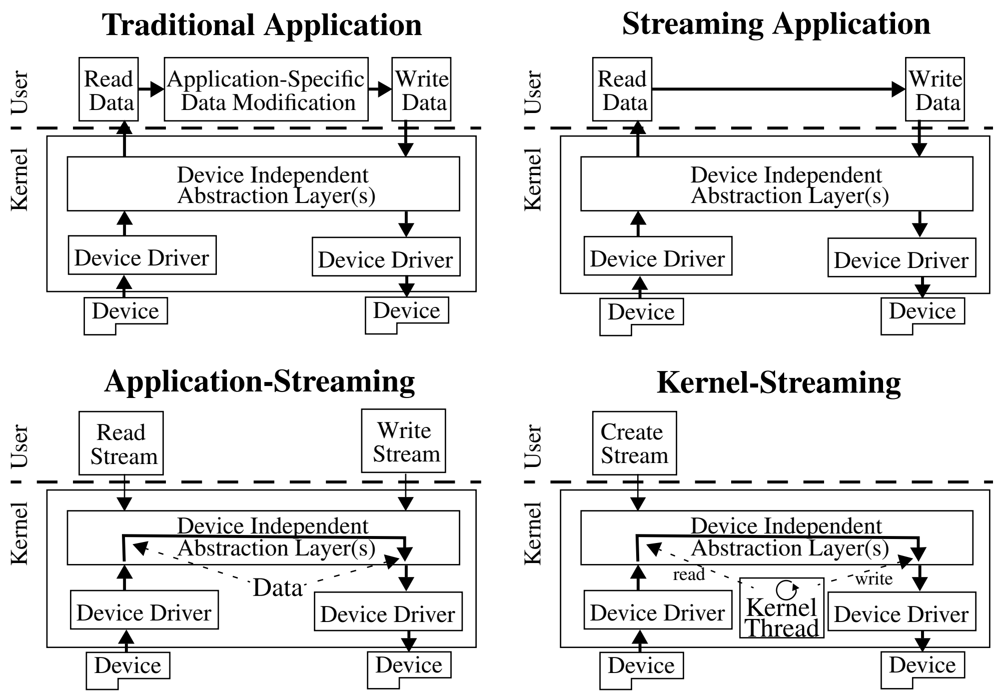

Figure 7: Data movement styles.

5. ADAPTIVE RESOURCE MANAGEMENT

Static resource management can provide reliable QoS in principle, i.e., it ensures that all resource requirements of multimedia applications can be met. For this, information about the resource demands of applications must be available and these requirements must be relatively constant. These two conditions can be satisfied for certain application classes, for instance, for most video server architectures.

However, for several (endsystem) multimedia applications these demands are not known or they do not have this characteristic but require a varying amount of resources over time. For example, a video playback application which performs software decompression of the audiovisual data needs a varying number of CPU cycles over time, e.g., to decode the various types of MPEG frames and also due to scene changes.

With static resource management, a change in the set of running applications is only possible if the total amount of required resources is less than the overall available capacities. Hence, if the spare capacity is less than needed for an additional application, it can only be started if another application is terminated. Yet, several scenarios exist in which a user would like to start an additional application and keep all others running, perhaps with reduced quality. For instance, a user watches some news stories and receives a videophone call; she would like to keep the news stories running (probably with a reduced video size and without audio) and talk to the caller. Yet, the system resources are not sufficient to serve all these applications with unchanged quality (e.g., there might be enough spare resources to serve the videophone application with low quality). With static resource management the user may stop the news story or accept the videophone connection with low quality – which is different or even opposite to the intention of the user (low quality news, high quality videophone) because the videophone will receive most attention.

Finally, distributed multimedia applications use several resources, e.g., CPU, memory and network. As we have seen in Section 3.4, to resolve access interdependencies among them, the scheduling of these resources must be coordinated in order to achieve an overall satisfying presentation quality. If this is not always possible, the result is a temporary unavailability of required resources which potentially leads to a missed deadline.

Adaptive resource management address' these issues. The goals of adaptive resource management systems are, e.g.,

- support of variable-bit rate streams which have dynamically varying resource requirements,
- adaptation to changes in the set of applications to be served,
- allowing for a dynamic change in the relative priority of applications,
- serving more applications concurrently as is possible with hard (worst-case assumptions) based QoS provisioning,
- handling of changes in resource availability.

As a drawback, due to the adaptation mechanisms, adaptive resource management system are usually not able to provide guaranteed, constant QoS[1]. They typically assume that multimedia applications are soft real-time and are tolerant to graceful adaptations, which is in difference to traditional hard real-time applications. This assumption is usually acceptable for desktop multimedia applications. However, for some recording and production scenarios and special applications, used in, e.g., telesurgery or other mission-critical scenarios, this is, of course, not the case.

While multimedia applications executing under a static resource management system may assume that their resource requirements will be fulfilled all of the time and that they always have access to the resources when needed, this is not the case with adaptive resource management systems. Here, applications must be prepared that the amount of resources available for them varies over time, due to

- changes in the size and mixture of the application set,
- modifications in the priority among applications,
- varying resource requirements because of inexact resource specifications.

To limit the changes in accessible resource capacities, the adaptive resource management system can provide applications with the ability to specify their requirements as a range [*min, max*], i.e., the minimum amount of resources needed for proper operation is *min*.

The basic scheme applied by adaptive resource management systems involves the *system resource manager* and *resource monitor* components and the applications (as illustrated in Figure 8).

- The resource monitor is a part of the resource manager for a particular resource. It observes the resource usage of applications and the overall load of the considered resource. It delivers according state informations to the system resource manager.
- The system resource manager gathers the state informations from the resource monitors. This component is also responsible for the negotiations with the application (at the beginning and during run-time for renegotiations), thus it has information about the QoS requirements and resource demands of all applications. Based on this information about system state and application characteristics, it can decide which application should adapt its resource usage and to what extent.

---

1. Of course, mixtures of both approaches are possible, e.g., serving only some applications based on worst-case assumptions. The ability to reach the described goals are reduced in that case nevertheless.

- The applications receive adaptation notifications from the system resource manager. Based on that, they decide how they change their behavior to adapt their resource demands. Additionally, they monitor the QoS they can achieve, if this becomes too low or too high, they start a QoS (and hence resource requirements) renegotiation with the system resource manager.

Instead of choosing just one application for adaptation, the system resource manager can balance the need for adaptation over several or even all applications. This way, it can provide for fairness among the applications or it can ensure that applications which are critical for the whole system or important for the user are preferred, i.e., they do not have to reduce their resource requirements. In collaboration with the applications and their QoS requirements, the system resource manager can also perform a balancing among various resources, e.g., trading network bandwidth vs. CPU.

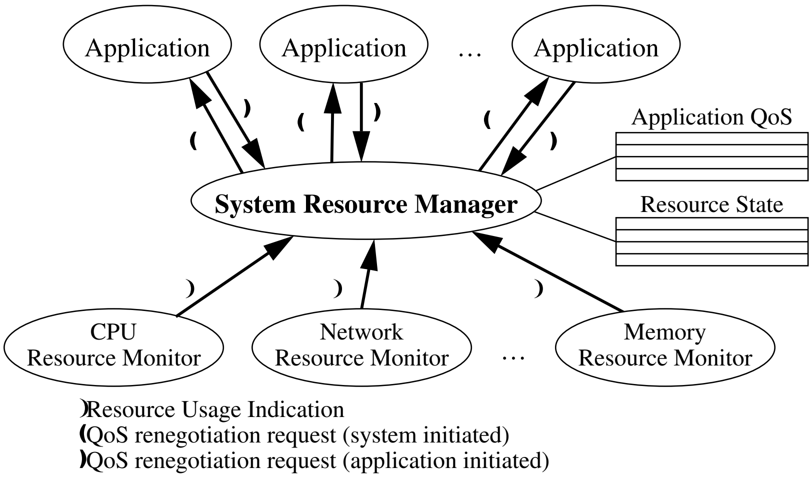

Figure 8: Basic adaptation scheme.

Adaptations do not always have to be towards lower resource usage. For instance, if an application finishes, the resources used by it so far are deallocated. Hence, they become available for other applications, either for the running applications which can then execute with better quality or for new applications which will be started in future. To support the former case, the system resource manager can perform a QoS renegotiation with the applications (indicating the amount of resources available for them). Again, the adaptations can be balanced, to increase the quality of all applications or only a subset.

Various architectures have been designed for adaptive resource management. One example is AQUA (**A**daptive **Qu**ality of service **A**rchitecture) [8].

A cooperative model of resource management is applied by AQUA where the application and the resource management cooperate to manage the resources. For CPU scheduling, AQUA uses a 'rate-based adjustable priority' policy. This extension of the simple rate-monotonic scheduling method, accounts for unknown and varying compute times and global adaptation across many applications. An application which starts its operations gives only a partial specification of its resource requirements. For example, the application specifies the execution rate but not the compute requirements. The resource manager allocates resources for this application based on an estimate of the available capacity. A regulator ensures that a task does not execute more often than specified by that initial rate.

During the execution of the application, the application estimates its resource requirements and the QoS it receives. The resource management performs similar operations, monitoring is performed inside the system (by the scheduler) to detect the resource usage. Based on the gained information about resource usage, the resource management potentially requests a reduction of the execution rate from the application. Changes in the measured QoS occur if the requirements of the application vary or if the resource availability alters. Then renegotiations are performed between the application and the resource management; further, according adaptations in the processing steps performed by the application are made to ensure that a predictable service is provided.

6. FURTHER ISSUES

6.1 RESOURCE ACCOUNTING

Knowledge about resource usage is necessary for several issues. It is required for the admission control, for instance, for the rate-monotonic schedulability test the processing times and rates are needed. And it is needed to 'charge' the user for its consumption of resources. Additionally, having exact information about current resource usage with fine granularity allows for better resource allocation and also scheduling decisions (mostly with adaptive schemes) and enables the resource management to detect miss behaving applications. (i.e., which use much more resources than originally specified).

The determination of the requirements used for admission control can be performed in advance, i.e., before the application is actually executed. Yet, the measurements must be done on the system which will execute the application because the demands depend on several issues such as hardware platform and operating system version. Such estimations may be performed in an installation phase of the application. Nevertheless, bounding the application requirements to a fixed measurement phase on each single computer complicates the system administration in large environments where many computers share applications stored on central file servers.

To charge the user for resource consumption and especially to use resource usage information for allocation, scheduling, and policing decisions, on-line measurements must be made during the applications run-time. Since such a measurement influences the performance of the system permanently, it must be possible with very low overhead. On the other hand, the yielded values must also be of fine granularity. Current operating systems do not provide sufficient support for this purpose. At best it is possible to see when a particular task *started* and when it *stopped* its execution in a period (often with coarse granularity in the order of several milliseconds only) but not how long it used the resource. These values can be quite different because other tasks or system activities might have been executing in the meantime.

This could be simplified if the operating system would provide appropriate and better support mechanisms. One relatively simple and cheap approach is to introduce a task state variable D_i which contains the run-time of the task i. It can be implemented in the following way:

- A system-wide variable E holds the time stamp of the last context switch or interrupt.
- As part of the creation of a new task j the variable D_j is set to 0.
- If the operating system dispatcher deactivates a task k and activates a task l, thus, while performing a context switch, the time elapsed since the last interrupt respectively context switch is added to D_k and E is set to *current_time*:

$$D_k \leftarrow D_k + (\text{current\_time} - E)$$

$$E \leftarrow \text{current\_time}$$

While such a method can help to determine the processing time requirements of tasks and to check whether they stay (reasonable) within their specifications, it does not provide support to accumulate the resource amount used in summary for a particular application. For this, the resource usage of server tasks which are executing on behalf of this application must also be taken into account (see also Section 3.4 where we noticed already that such a system structure leads to difficulties due to task dependencies).

Support for the processing of time-critical multimedia applications in an existing operating system is often restricted by the basic design and structure of that system. While enhancements can be made, there are often limitations which cannot be relieved without major changes in the base system. A more radical but also more general approach is to develop a new operating system where support for continuous-media data is incorporated in the design from the beginning.

6.2 RESOURCE MANAGEMENT IN FUTURE OPERATING SYSTEMS

With todays operating systems, resource management for multimedia processing is mostly an add-on feature but not fully integrated. Therefore, several limitations exist with respect to the ability to handle concurrently several high-performance, high-quality continuous-media streams.

Future operating systems, where the requirements of multimedia applications are taken into consideration already from the very beginning of the design, may offer enhanced capabilities. With them, more efficient resource management support and, hence, simpler and better handling of continuous-media data can be provided. And less limitations will exist for the distributed multimedia systems based on them.

Such operating systems and the according resource management mechanisms within them will offer the ability to perform exact accounting of resource consumption, avoid dependencies among tasks, and will be able to circumvent the interference between multimedia applications.

Nemesis [10] is an operating system which has been designed to support distributed multimedia applications. Nemesis takes a revolutionary approach of starting the operating system design from scratch and to not just make incremental changes. It offers facilities for the dynamic allocation of resources to applications, it ensures that resource consumption is accounted to the correct application, and it allows that applications avoid the use of shared server tasks – as much processing as possible is done within the application itself, whereby protection among appplications and security is nevertheless be given.

6.3 RESERVATION IN ADVANCE

For several application scenarios, the model of 'immediate reservations' applied so far is not fully appropriate. As in our daily lives, it must be possible to perform a reservation in advance to ensure that our application can be executed with sufficient QoS. If there is a noticeable blocking probability for immediate reservations, it must be possible to reservations for in advance for a specified start time and duration. This means that Resource Reservation in Advance (ReRA) [18] mechanisms are needed, but by now, only preliminary results are available on this topic. There are subtle problems to be solved, besides a more complex resource management, some of the problems occurring in ReRA systems are state maintenance and failure handling.

ReRA must be performed on an end-to-end basis. Appropriate resource requirement information must be exchanged in advance among the participating systems by extending the FlowSpecifications distributed via the reservation protocols. The resource management on each node needs extended admission control tests which check whether the required resources can be provided during the requested time interval. And the data structures used to store the reservation information must contain time values (begin and duration of a reservation).

The state associated with an advance reservation must be kept on all the participating systems for a potentially long time. It must either be stored in non-volatile memory to survive system failures and regular shutdowns, e.g., due to system maintenance, or a soft-state approach must be used where the reservation is refreshed from time-to-time: the closer the actual usage time comes the higher the frequency of the refresh messages. The handling of failures which occur between the reservation setup and its use must differ from the steps taken to resolve errors of running applications. The reason is that the application which had lost its reservation is not running. So, it is not immediately clear which entity is to be notified and by which means. Further, potentially the failure situation can be cleared already before the resources are needed.

7. SUMMARY

Multimedia applications need the integrated treatment of continuous-media and discrete-media data in distributed computer systems. The handling of continuous-media data such as audio and video places requirements on the multimedia computing and communication infrastructure which are uncommon for workstations and most other computer systems. To deal with the timing demands of these applications, new mechanisms must be used.

Resource management has the goal to provide reliable QoS to distributed multimedia applications. It uses admission control and scheduling mechanisms to manage the resources needed during the processing and transmitting of the audiovisual data. For that, resource management has to interact in a tight manner with operating system and communication system mechanisms to achieve its goals. Furthermore, according services must be available in the endsystems and the networks.

Resource management has been an area of active research for several years and it is still a field where research and development is proceeding. Various mechanisms have been designed over the time, both for static and for adaptive resource management, and several example systems have been designed, implemented, and evaluated.

Increased processing speeds, higher network bandwidth, larger memories, and other hardware improvements might reduce the need for resource management techniques in future systems. However, it is likely that application demands will increase as well. Additionally, for systems and components shared by several users, resource management mechanisms will be an important piece of future multimedia infrastructures to provide a reliable QoS.

Resource management designed within the context of current operating systems is somewhat limited in the functionality it can provide. Future operating systems which are designed for the handling of multimedia applications will be able to relieve these limitations.

REFERENCES

1 Anderson, D. P., Tzou, S., Wahbe, R., Govindan, R., Andrews, M., “Support for Continuous Media in the DASH System”, Proceedings of the 10th ICDCS, Paris, France, May 1990.

2 Anderson, D. P., "Metascheduling for Continuous Media", *ACM Transactions on Computer Systems*, Vol. 11, No. 3, 1993.

3 Braden, R., Zhang, L. Berson, S., Herzog, S., Jamin, S., “Resource Reservation Protocol (RSVP) - Version 1 Functional Specification”, RFC 2205, September 1997.

4 Fall, K., Pasquale, J., “Improving Continuous-Media Playback Performance with In-Kernel Data Paths”, Proceedings of the IEEE ICMCS, Boston, MA, USA, May 1994.

5 Herrtwich, R. G., Wolf, L.C., “A System Software Structure for Distributed Multimedia Systems”, Proceedings of the ACM SIGOPS European Workshop, Le Mont Saint-Michel, France, September 1992.

6 Huang, J., Kenchammana-Hosekote, D., Agrawal, M., Richardson, J., "P*resto* – A System for Mission-Critical Multimedia Applications", *Journal of Real-Time Systems*, July 1997.

7 Jones, M., "Adaptive Real-Time Resource Management Supporting Modular Composition of Digital Multimedia Services", Proceedings of the NOSSDAV Workshop, Lancaster, UK, 1993.

8 Lakshman, K., Yavatkar, R., Finkel, R., "Integrated CPU and Network-I/O QoS Management in an Endsystem", Proceedings of IWQoS, New York, NY, USA, May 1997.

9 Lehoczky, J., Sha, L., Ding, Y., "The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior", Proceedings of the IEEE Real-Time Systems Symposium, Santa Monica, CA, USA, 1989.

10 Leslie, I., McAuley, D., Black, R., Roscoe, T., Barham, P., Evers, D., Fairbairns, R., Hyden, E., "The Design and Implementation of an Operating System to Support Distributed Multimedia Applications", *IEEE Journal on Selected Areas in Communications*, Vol. 14, No. 7, September 1996.

11 Liu, C. L., Layland, J. W., "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment", *Journal of the ACM*, Vol. 20, No. 1, 1973.

12 Liu, J. W. S., Lin, K.-J., Shih, W.-K., Yu, A. C., Chung, J.-Y., Zhao, W, "Algorithms for Scheduling Imprecise Computations", *IEEE Computer*, Vol. 24, No. 5, May 1991.

13 Mercer, C. M., Savage, S., Tokuda, H., "Processor Capacity Reserves: Operating System Support for Multimedia Applications", Proceedings of the IEEE ICMCS, Boston, MA, USA, May 1994.

14 Nagarajan, R., Vogt, C., "Guaranteed-Performance Transport of Multimedia Traffic over the Token Ring", IBM Tech. Rep. No. 43.9201, IBM ENC, Heidelberg, 1992.

15 Vogt, C., Herrtwich, R.G., Nagarajan, R., "HeiRAT: The Heidelberg Resource Administration Technique - Design Philosophy and Goals", Proceedings of Kommunikation in Verteilten Systemen, Munich, Springer-Verlag, 1993.

16 Vogt, C., Wolf, L. C., Herrtwich, R.G., Wittig, H., "HeiRAT – Quality-of-Service Management for Distributed Multimedia Systems", *ACM Multimedia Systems Journal* – Special Issue on QoS Systems, 1998.

17 Wolf, L.C., "Resource Management for Distributed Multimedia Systems", Kluwer, Boston, MA, USA, 1996.

18 Wolf, L.C., Steinmetz, R., "Concepts for Resource Reservation in Advance", *Multimedia Tools and Applications*, May 1997.