Quality of Service in Distributed Multimedia Systems

Sape J. Mullender and Paul Sijben

University of Twente, Faculty of Computer Science, Enschede, The Netherlands
{mullender,sijben}@cs.utwente.nl

Abstract. The Unix operating system made a vital contribution to information technology by introducing the notion of composing complicated applications out of simple ones by means of pipes and shell scripts. One day, this will also be possible with multimedia applications. Before this can happen, however, operating systems must support multimedia in as general a way as Unix now supports 'ordinary' applications. Particularly, attention must be paid to allowing the operating-system service to degrade gracefully under heavy loads.
This paper presents the Quality-of-Service architecture of the Huygens project. This architecture provides the mechanisms that allow applications to adapt the level of their service to the resources the operating system can make available.

1 Introduction

If one defines a multimedia system to be a computer system that allows applications to adequately process text, graphics, images, sound and vision and convert between these media, then most systems advertised with 'multimedia' among the adjectives do not deserve that qualification for one of two reasons. The first and most common reason is that they do not or they hardly allow processing the time-dependent media of audio and video. The second, more subtle, reason is that they do not allow *applications* to process audio and video, but leave it to the operating system instead.

Delivering the real-time performance necessary to capture, transport and render audio and video on time is much easier to do in the bowels of an operating system than it is for an application. The operating system merely has to assign a higher priority to its internal real-time processes than to the (non-real-time) applications. The price paid here, however, is lack of flexibility: the only multimedia processing possible is that built into the operating system and the multimedia load that can be placed on the system is fixed *a priori*.

Multimedia applications are starting to play an important rôle as a vehicle for improving the quality of interpersonal communication over long distances. As a result, it is essential that operating systems support *distributed* multimedia applications.

Multimedia may be expected to be the next quantum leap in making information processing accessible to the layman: having to control computer systems

through keyboards and mice is not a natural way to do business and providing instant feedback to mouse commands only in the form of an hourglass-shaped mouse icon, while taking a highly variable amount of time for the real work is, for the uninitiated, completely bewildering.

Multimedia, therefore, should not merely be about adding audio and video to computers, it should just as much be about designing human interfaces on human reaction-time scales. Having made transatlantic phone calls via satellite links, one realizes just how much the modified timing — a satellite link adds a quarter second to the end-to-end communication latency — throws one off.

The issues in designing operating system architectures for the support of distributed multimedia applications are thus providing low-latency communication and processing, and predictable real-time behaviour for unpredictable multimedia loads placed on the system.

This is, in a nutshell, the goal of the University of Twente Huygens project: the design of an architecture for distributed multimedia systems. The remainder of this paper describes our research of operating-system support mechanisms for distributed multimedia applications and presents the Huygens Quality-of-Service scheduling architecture.

The Huygens project is closely linked to the Pegasus Esprit projects[1] [LMM94] which investigates multimedia support on a broader scale: new operating-system structures, multimedia data storage, multimedia local and long-haul networking, and Quality of Service.

2 The Nature of Multimedia

From the viewpoint of an operating-system designer, multimedia support is about handling time-dependent media. The most prominent examples of such media are digital audio and video, but sensor data, such as the output of an ILS[2] in an aircraft are often time dependent too. A frequently used technical term for time-dependent media is *continuous media.*

We may consider it an attractive property of multimedia systems if they support *ad hoc* compositions of 'multimedia building blocks' into multimedia applications. Consider, for instance, how a teleconferencing application, a multimedia-document player, and a multimedia recording application can be combined to record the reactions of the participants in a multimedia conference to playing a multimedia document.

When the sum of the resources requested by the applications exceeds what the operating system can offer, then things become interesting. Since the applications have real-time requirements, *virtualizing* resources in the way of time-sharing

[1] The Pegasus Project is a project, initially of the Universities of Twente and Cambridge, now also of the University of Glasgow, the Swedish Institute of Computer Science, and APM Ltd., supported by the European Communities' ESPRIT Programme through BRA project 6586 (1992 – 1995) and LTR project *21917* (1996 – 1999).

[2] Instrument Landing System

systems is not possible. The *only* way for applications to cohabit on a host is by making certain that the resources consumed by them do no exceed the available.

This is where multimedia systems essentially differ from real-time systems: real-time behaviour has to be maintained even in a '*resource-overload* situation'. The basic technique is, confusingly, alluded to using the term *Quality-of-Service*, or *QoS*. The idea is that applications adjust the quality of the service they provide to the types and amounts of resources available. Over a narrow-band network, a tele-conferencing application will thus display fewer video channels of lower resolution than over a broad-band network.

The big challenges in multimedia system design are, first, to design multimedia applications that can indeed adapt to a wide range of resource offerings and that maximize their QoS under a given resource allocation, second, to design the operating system such that the amount of resources available to actual multimedia processing is maximized and that the amounts needed for the allocation itself is minimized, and, third, to define algorithms and mechanism for allocating resources to a set of applications — or even users — competing for them.

3 Quality of Service

Applications can adapt to the resources available by adjusting the quality of the service they provide. When there are ample resources, applications can give a high quality of service, when resources are scarce, applications can provide only limited quality of service.

Such adaptation by applications is, of course, limited. At the high-quality end of an application's spectrum, a point will be reached where the supply of more resources can no longer be used to improve quality. At the low-quality end of the spectrum, a point is reached where any further reduction of resources renders an application useless. In multimedia applications with no QoS adaptation, the ends of the spectrum coincide [DHH+93,EA95].

Both ends of the spectrum give rise to observations. At the high end, it must be noted that a point can be reached where pure *continuous*-media applications do not run better or faster when more CPU bandwidth or network bandwidth is made available to them. Traditional applications usually do run better in that they return results more quickly. Multimedia system design, therefore, is not simply about performance optimization — enough resources is usually enough.

At the low end of the spectrum, we find that there is a lower limit on the amount of resources needed by an application to do something useful. If those resources cannot be found, it is better not to start the application at all. Before making such a decision, however, it is a good idea to see if running applications can reduce their QoS to make room for the newcomer.

Multimedia applications can adapt their resource consumption to what can be allocated in basically three ways. (1) They can change their functionality; e.g., change from stereophonic sound to monophonic, change interactive document-sharing semantics, drop one of the video channels, introduce compression. (2) They can change network and processing bandwidths; e.g.,reduce image size,

reduce audio-sampling size, change the quality parameters of a compression algorithm. (3) They can change the frequency at which they operate; e.g., drop the frame rate from 25 to 15 fps, reduce audio sampling rate, adjust camera position once per second instead of five times per second.

QoS adaptations should be carried out to maximize QoS improvement for a given increase in resources, or minimize QoS degradation for a given reduction in resources. The quality of service that counts is often the quality of service as perceived by a human user. It is, therefore, important that the user can influence how QoS adaptations are carried out — in other words, the user must be allowed to determine which media may enjoy additional or must suffer reduced resources and which QoS parameter of such media should be affected.

Perceived Quality of Service for video depends on image size (number of pixels), image quality (effects of lossy compression and lost or late data), frame rate (images per second), jitter (irregularity of frame arrival times), and, for interactive applications, latency (time *en route* for video data). For audio, perceived QoS depends on sampling frequency, sample size, data loss (due to compression or transmission), and latency.

Achieving a certain quality of service requires the allocation of a certain amount of resources. Obviously, higher frame rates and better image resolution uses up more network bandwidth, to name just one resource. In general, a higher quality level requires a bigger allocation of resources.

But note that modern image-compression techniques can dramatically reduce the bandwidth required to transport a video image, while barely affecting quality. Such compression potentially improves latency (less data to transmit) at the cost of very minor loss in image quality. The same perceived quality can often be achieved with or without compression. With compression, a substantial CPU allocation is necessary to perform the calculations for compression and decompression, while the network bandwidth allocation can remain low. Without, the CPU allocation can be less, but more bandwidth is needed. There is, as we can see, no obvious monotonic relationship between QoS and resource allocations.

An application can apply the resources it obtains to achieve optimum QoS. Resource scheduling within an application is up to the application itself. When there are multiple, independent applications, however, both resource allocation and resource scheduling must be done by the operating system.

Allocation strategies need to be in place that allocate resources to applications in such a way that the overall QoS — the QoS of all applications combined — is optimal. How overall QoS can be determined is an interesting problem of cognitive ergonomics which, for now, we shall not attempt to solve.

Applications can, in principle, obtain more resources than they deserve by over-claiming. Users of time-sharing systems were known to apply this technique to get better service: they would start up a large number of parallel jobs to get better personal service, even though it was detrimental to other users and system throughput.

This technique of over-claiming resources in order to get a larger share has become less useful now that most systems are personal workstations. The same is

true of multimedia applications: they run, to a large extent, on personal workstations where the user is best served by having the applications work harmoniously together. It is, therefore, entirely reasonable to assume that operating systems do not have to operate in a *competitive* environment, but rather in a *cooperative* one.

When two applications both need, say, 60% of the CPU to perform optimally, and only one of them is capable of QoS adaptation and make good use of only 50% or 40% of the CPU, then it is clear that fairly dividing the CPU over the two applications by giving them 50% each is not nearly as useful as giving the non-adaptive one the 60% it needs and the other the remaining 40%.

A QoS architecture is a collection of interfaces and algorithms for an operating system, that allows applications to describe the Qualities of Service they can deliver and the resources they need to deliver them, that allows the operating system to determine the best possible overall Quality of Service with its attending resource allocation, and that allows the applications to adapt to that allocation. Preferably, all this takes place in a dynamic setting, where, resources allocations can change whenever the QoS settings of the applications change.

QoS architectures are a a topic of many multimedia research groups. Approaches roughly follow one of two paths: *reservation* and *adaptation.*

The reservation-oriented groups assume a full knowledge of the properties of the running multimedia applications and try to reserve exactly the right amount of resources for this. Capacity reserves, as used in [MST94], are a way to try to implement an estimation of the resources.

Adapatation of a multimedia application can be achieved by algorithms that can produce usable partial results and alternative implementations that take less resources than the primary implementation does. This kind of processing is known as imprecise computations [ABRW91].

The adaptation path is used when the researches find that the load of multimedia applications is not predictable. The application is allocated some resources and may find more available at run-time [Ros95]. This approach can produce satisfactory results when resources are not really scarce or the application is highly adaptable like an MJPEG [Wal91] decompression application [Hyd94].

The Hyugens approach tries to use the best of both apporaches. Estimates are made for the nominal and worst case resource needs. A suitable amount is preallocated gambling on the fact that the resources will be present when the peak need occurs. This apporach tries to maintain statistical QoS guarantees at run-time.

The Dynamic QoS control approach of [FN96] has a similar run-time concept. But lacks the central entity we call a *QoS manager* to prevent oscillation of QoS levels. Others have the central entity but not the fluent adaptation [FHS96].

4 Multimedia Scheduling

The observation was made earlier that one of the biggest problems of continuous-media processing is that one must make do with the resources one has, even when

the combined applications would be better served with much more. Somehow, we are prepared to accept this much more easily when considering network resources, perhaps because network resource scarcity has always been a problem primarily in public networks where one has little control over the allocation in any case.

When it comes to the CPU resource, we are used to time-sharing systems in which the CPU is virtualized — the API presents the illusion that each process has the whole CPU to itself; it only becomes a slower CPU when it gets shared with other processes. In non-real-time applications, this doesn't matter, but in real-time applications such as those that process continuous media, it matters a great deal.

The time-dependency of continuous media suggests that operating system support for multimedia might well be provided by real-time systems. This is not the case, even though there are many similarities in the techniques used in multimedia systems and real-time systems.

Real-time systems can meet their deadlines because the real-time load placed on the system is bounded *a priori* and upper bounds are known for the processing times of all real-time tasks. In multimedia systems, demanding such *a priori* bounds is not realistic: multimedia applications will be 'ordinary' user-space applications that cannot be expected to meet processing-time limitations dependably, and users will not put up with low limits on the number or mix of multimedia applications they can run together.

To compute schedules that meet the deadlines, real-time systems require *a priori* known and bounded loads and predictable and bounded run times for real-time processes. None of these conditions are met in the operating systems that we are interested in: Continuous-media processes run in user space and may or may not meet the run times promised by their developers; the system load is not bounded, users will fire up more simultaneous multimedia applications until the system stops giving reasonable service.

If a conventional real-time system would be used to schedule multimedia applications, it would work fine as long as loads are within bounds, but when those bounds are exceeded it would probably break horribly: real-time systems do not specify what to do in an overload situation because it is assumed that, by design, overload cannot happen.

In most multimedia applications, missing a deadline as a consequence of overload is not at all disastrous, as long as the system only misses an occasional one. If a single frame is missing from an incoming video stream, one can leave the previous frame displayed until the next one comes in and nobody will notice. If this happens to a number of frames in a row, however, or to every second frame, then it would be noticeable. The same is true for audio, when the unit of loss is small.

Data loss will happen in the network (e.g., as a consequence of policing or transmission errors) and retransmission is rarely an option due to lack of time to do so. Data loss is, therefore, a fact of life in multimedia systems and it can happen in transmission as well as in processing. The possibility of allowing a

deadline to be missed occasionally can be exploited in operating system scheduling algorithms.

Interactive multimedia applications seek end-to-end latencies of no more than a 100 ms or so. This implies that, for processing steps, only a few milliseconds are available. Multimedia applications will, therefore, need to be scheduled with intervals and deadlines in the millisecond range.

On a host executing several multimedia applications, there will be a substantial amount of context switching, typically on the order of a thousand times per second. It pays to optimize the operating system for this.

Making scheduling decisions at this rate is likely to impose a significant load on an operating system. The scheduling algorithm will either have to be very simple or it should be run off line. In the Huygens project, we have opted for the latter. We made the observation that, after assigning resources to processes for their particular QoS settings, the system has information about the frequencies, deadlines and run times for all tasks. This allows the calculation of a schedule in advance. Since all tasks are periodic, the schedule will be periodic as well [SM95].

5 Scheduling in Huygens

In Huygens, the calculation of such a periodic schedule results in a table that the dispatcher uses to identify the next task to run. The dispatcher is activated by the system clock which generates interrupts at a rate of between one per ten milliseconds and one per 100 microseconds. At every clock tic, a number of periodic tasks can be invoked. When one returns control, the next one is invoked. When all tasks of a particular clock tic have been run, the dispatcher returns control to the process that was interrupted by the clock.

This works fine until a periodic tasks misbehaves and takes more time than allocated to it. Once a task has the CPU, it can hang on to it until the next interrupt returns control to the operating system. In Huygens, we catch such processes at the next clock interrupt. Periodic tasks that should have been run behind the offending task will miss their opportunity to run, but only once: the offending task can be removed from the schedule, or, less drastically, it can be scheduled as the last task of the group for a particular clock tic.

At this level, all periodic tasks run to completion. This ensures the best system throughput. Periodic tasks with long run times of ten milliseconds and more, however, cannot be run to completion without starving high-frequency periodic tasks of the CPU. Viewed from the low-level dispatcher, tasks with long periods are run as background processes that are preempted by the system clock.

The low-frequency long-run-time tasks are dispatched by a higher-level dispatcher. This dispatcher is invoked by a low-level periodic task at a frequency of 10 Hz or so. To the low-frequency dispatcher, the actions of the high-frequency dispatcher only manifest themselves as variations in the number of CPU cycles

available between low-frequency clock tics. Otherwise, the low-frequency dispatcher operates exactly like the high-frequency one.

During the calculation of the schedule, the time available at each dispatcher level is known and taken into account. The fact that a task runs at a higher frequency and thus preempts another task at a lower frequency does not imply it has a higher priority or a right to more CPU cycles.

The hierarchy of dispatchers can be made more than two levels deep, although this is not usually necessary. The scheduler that computes the dispatch tables can be run as a non-real-time background process. The scheduler that schedules the background processes can be run as a low-frequency periodic real-time task.

Applications provide the operating system with a list of their possible QoS settings. This list typically contains between one and half a dozen settings. An applications that cannot adapt has one setting. Multimedia background applications, such as an application that shows the current five-day test match, but only appears when the resources are available, may have its lowest-quality setting have no resources associated with it.

For each QoS setting, the application provides the following information:

- The QoS rating of the setting, a number between 0 and 255 — the higher this number, the more desirable this setting is.
- Two flags, named *deliverable* and *desirable*. The former indicates that the system (the QoS scheduler) is currently prepared to schedule this QoS setting. The latter indicates that the application is currently prepared for the system to schedule the QoS setting. The use of these flags is explained below.
- A list of tasks (periodic threads). Each task has an entry with the following information:
  - Task entry point. This tells the system how to invoke the task.
  - Sources and sinks. These are (possibly empty) lists of references to other tasks or connections and tell the scheduler where the task fits in a continuous-media pipeline. This information defines a partial order that lets the scheduler determine a scheduling order for the tasks.
  - *Optional* flag. This indicates that the task need not be invoked when CPU cycles are scarce.
  - Task period in μs.
  - Task CPU consumption in cycles per second.
  - Task memory consumption in kilobytes.
- A list of connections (virtual circuits). Each connection has an entry with the following information:
  - Connection type and direction; e.g., inbound MJPEG-compressed video over AAL5.
  - Peer; the address or name of the peer entity at the other end of the connection.
  - Sources or sinks (depending on direction). These are (possibly empty) lists of references to other tasks or connections and tell the scheduler where the connection fits in a continuous-media pipeline.
  - Connection period in μs. This is the time between transmission of logical units (e.g., a video frame).

- Connection bandwidth in kilobytes per second.
- Connection buffer size in kilobytes.

6 QoS Resource Allocation

The QoS scheduler attempts to allocate the available resources to applications in such a way that the overall QoS is maximized. The overall QoS is simply defined as the weighted sum of the QoS ratings of all running applications. We refer to the *weight* of an application as its *importance*. It is an integer between 1 and 255. If the importance of application i is I_i and its QoS rating is at setting Q_i, then the QoS scheduler attempts to maximize $\sum_i I_i \times Q_i$.

The number of multimedia applications on a typical host will be a small number, as well as the number of QoS settings for each of these applications. Finding an optimum or near-optimum setting, therefore, is not overly compute intensive.

The dispatching tables produced by the QoS scheduler will contain for each application the threads of the selected QoS setting. The order in which the threads are invoked by the dispatcher matches the partial order given by the *sources/sinks* specification. As a result, a group of threads forming a pipeline will see the data pass through the complete pipeline in period of the group.

This arrangement of the dispatching table not only reduces the latency of continuous-media data through the pipeline, it also helps in dynamically switching from one QoS setting to another. This works as follows.

When a new QoS setting for an application is chosen, a new schedule is computed in the form of a new dispatching table that contains the invocations of threads as specified by the newly chosen QoS setting. This dispatching table is than activated by having the dispatcher run itself off the end of the old dispatching table into the beginning of the new one. When the dispatcher has finished dispatching the last thread of the old table, all pipelines formed by multiple threads are empty and a fresh start can be made by starting the first thread in the first pipeline in the new table.

If the QoS setting changes not too drastically, it is unlikely that the user will notice such QoS changes much. Audio and video streams continue uninterruptedly.

This type of dynamic QoS resource management works fine in a centralized setting where the changeover from one dispatching table to another can be made atomically. In a distributed setting such atomic changeovers are not possible. They must be initiated in one host and trigger the changeover in the other hosts involved. Before this can happen, however, it must be ascertained that each host is prepared to support the new QoS setting.

To this end, a QoS setting contains two flags, named *deliverable* and *desirable*. The former is set by the operating system and read by the application, the latter set by the application and read by the operating system. The *desirable* flag indicates that the application is prepared to have the system schedule the QoS setting. The application may need to set this flag to `false` when peer

applications on other hosts currently cannot deal with the QoS setting — usually as a consequence of lack of resources.

As a concrete example, consider a distributed application that can send compressed or uncompressed video. A single process cannot unilaterally decide to switch from one to the other because the other processes may not have the resources to deal with that. As long as it is not clear whether the peer processes can deal with a particular QoS setting, the *desirable* flag is kept set to `false` and the system will not schedule the QoS setting.

The *deliverable* flag indicates that the system has the resources to run the QoS setting (but it will not do so unless the *desirable* is also set). Distributed applications will communicate this fact to their relevant peer processes which can then set their *desirable* flags to `true`.

When a QoS setting is both *deliverable* and *desirable*, we call the setting *possible*. The scheduler and the application together choose one of the *possible* settings to run. The details are given in the following section.

7 Huygens QoS Application Programmers Interface

The *QoS manager* is the application programmer's interface for an application's resource allocation and QoS management. The interface is implemented by the operating system, either as a system process in user space, or as part of the operating-system kernel.

For maximum flexibility, it is conceived as a message-passing interface allowing RPC from the application into the QoS manager and *callbacks* from the QoS manager to the application.

Application processes are assumed to be multithreaded with *management* threads in the non-real-time domain and *continuous-media* threads or *tasks* in the real-time domain. Although all communication *about* QoS settings takes place in the non-real-time domain, there are time-outs on all operations. If a process fails to respond to, for instance, an upcall within the timeout period, that is viewed as a programming error and may lead to the removal of the process.

The tasks (real-time threads) are essentially subroutines that are invoked by an upcall from the dispatcher. These subroutines are expected to return control to the dispatcher within the time specified in the relevant entry of the application's QoS table. Failure to return in time can lead to the removal of all tasks in the current QoS setting from the dispatching table. When this happens, the application is informed through an upcall to a management thread.

Processes belonging to distributed applications, especially, cannot simply be switched over by the operating system from one QoS setting into another. Remote peer processes may not be able to cope with such a switch.

The first stage of the QoS negotiations is an exchange between the processes in the application in which they determine the QoS settings they can support. If two processes are connected through a 64 Kbps ISDN link, for example, they have no choice but to send video in compressed form — there is no point in specifying

QoS settings without compression (unless, of course, such setting are without video as well). This determination involves querying the operating system for performance data of operating-system, CPU, device, and network.

This stage culminates in sets of possible QoS settings for each of the participating processes and knowledge within the application of the possible combinations of QoS settings in the set of processes. How this negotiation takes place is up to the application. The operating system offers a standard interface for resolving queries concerning the system's capabilities.

The second stage starts when the application, using its knowledge from the first stage and flagging the initially *desirable* settings, communicates the table of negotiated QoS settings to the QoS manager.

The QoS manager then calculates a new schedule that accommodates the new application process. When such a schedule cannot be found, the (real-time part of the) application process is not admitted. The process may notify the rest of the application of this failure and exit. When such a schedule can be found, however, it is installed and the process is informed of the QoS that was selected. The QoS manager also informs the application which other QoS settings are *deliverable*.

Before the QoS manager installs a new resource allocation, it notifies the applications of the new allocation. When the applications signal their readiness, the new allocation can be installed by replacing the dispatching table. In centralized applications that communicate with a single QoS manager, this method works fine.

Seamless switch-over from one QoS setting to another in a distributed setting, however, requires a sort of *two-phase commit* (2PC) protocol: during the first phase, the processes and operating systems prepare for the new setting and then, when all parties are ready, one of the application processes (typically a continuous-media data source) actually switches over, e.g., by starting to send compressed video instead of uncompressed. As processes notice the changeover, they tell their systems that the old QoS setting is no longer active. The details are as follows

Either the application, or one of the QoS managers takes the initiative for the change by suggesting a new QoS setting from the *possible* set. The application, acting as coordinator of the 2PC, informs all QoS managers involved of the suggested new setting. When a manager cannot accommodate the change, then the changeover fails and the QoS managers are duly informed. Note that applications can thus refuse changing their QoS setting.

When a manager *can* accommodate the changeover, it installs a *dual schedule*, a dispatch table in which both the old and the new setting are present. The condition of this installation is that the application will either use the resources of the old setting or that of the new, but not both. When all QoS managers have installed the *dual schedule* — and notified the application of this — the changeover happens as explained earlier.

When an application process has switched to the new setting, it informs the QoS manager and a schedule can be installed containing only dispatch instructions for the new QoS setting.

It is possible that, during the changeover from one QoS setting to another, resources are temporarily overcommitted and some deadlines are lost. This will be likelier when multiple applications have to change their QoS setting simultaneously, for instance, for admitting a new application. However, the changeover will happen quickly enough that the disruption of service is barely noticeable.

8 Conclusions

We have presented the Huygens Quality-of-Service architecture for general-purpose multimedia applications. Parts of the architecture have been implemented in the Nemesis operating system which has been developed in the Pegasus Esprit project of the Universities of Twente and Cambridge.

The Pegasus project is about to continue into its second phase, Pegasus II; this time Cambridge and Twente are joined by the University of Glasgow, the Swedish Institute of Computer Science and APM Ltd., the producers of ANSAware. The goal of Pegasus II is to bring the multimedia solutions of Pegasus to a broad audience and to incorporate it into industrial-strength operating-system platforms.

As a demonstration of the sort of applications that Nemesis will support, Twente developed a digital TV director — an application that controls several cameras in a meeting room in order to broadcast a report of the meeting taking place. The TV director detects and locates speakers (using triangulation over multiple microphones) and tracks them on camera (using a pan/tilt/zoom device and a combination of motion detection and skin-colour detection[3]). The TV director demonstrates that it is possible to write an application that carries out significant continuous-media data processing (audio triangulation and camera tracking) in real time. It adapts dynamically to the availability of resources by prioritizing its actions and the frequency with which it carries them out.

References

[ABRW91] N.C. Audsley, A. Burns, M.F. Richardson, and A.J. Wellings. Incorporating unbounded algorithms into predictable real-time systems. Technical report, University of York (UK), Sep 1991.

[DHH+93] Luca Delgrossi, Christian Halstrick, Dietmar Hehmann, Ralf Guido Herrtwich, Oliver Krone, Jochen Sandvoss, and Carsten Vogt. Media scaling for audiovisual communication with the Heidelberg Transport System. In *Computer Graphics (Multimedia '93 Proceedings)*, pages 99–104. ACM, Addison-Wesley, August 1993.

[EA95] Alexandros Eleftheriadis and Dimitris Anastassiou. Meeting arbitrary qos contraints using dynamic rate shaping of coded digital video. In *5th NOSSDAV ,Durham, NH, USA*, apr 1995.

[3] Our skin-colour detection algorithm has been checked for absence of racial bias.

[FHS96] Anne Fladenmuller, Eric Horlait, and Aruna Seneviratne. Qos management scheme for multimedia applications in best effort environments. *Journal of Electrical and Electronics Engineering,*, Mar 1996.

[FN96] S. Furuno and T. Nakajima. A toolkit for building continuous media applications using a new dynamic qos control scheme. In *Proceedings of the Multimedia Japan 96 - Yokohama March 18-20*, pages 268–277, Mar 1996.

[Hyd94] Eoin Andrew Hyden. *Operating System Support for Quality of Service.* PhD thesis, Wolfson College, University of Cambridge, February 1994.

[LMM94] Ian M. Leslie, Derek R. McAuley, and Sape J. Mullender. Operating-System Support for Distributed Multimedia. *Proceedings of the Summer Usenix Conference*, Boston, MA, pages 209–220, June 1994.

[MST94] C.W. Mercer, S. Savage, and H. Tokuda. Processor Capacity Reserves: Operating System Support for Multimedia Applications. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, May 1994. To appear. (This is a condensed version of tech report CMU-CS-93-157.).

[Ros95] Timothy Roscoe. *The Structure of a Multi-Service Operating System.* PhD thesis, Queens' College,University of Cambridge, April 1995.

[SM95] Paul G.A. Sijben and Sape J. Mullender. An architecture for scheduling and qos management in multimedia workstations. Technical Report Pegasus paper 95-05, University of Twente, Dec 1995.

[Wal91] Gregory K. Wallace. The jpeg still picture compression standard. *Communications of ACM*, 34(4):30–44, Apr 1991.