A Resource-Centric Approach To Multimedia Operating Systems

Shuichi Oikawa Ragunathan Rajkumar
Real-Time and Multimedia Laboratory
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue, Pittsburgh, PA 15213
{shui+,raj+}@cs.cmu.edu

1. Introduction

Resource management is central to real-time and multimedia systems. Currently deployed real-time and multimedia systems employ off-line mechanisms to ensure that sufficient resources are available to various tasks comprising the system. Firstly. in almost all these cases, given that the characteristics of the system are defined a priori, the demands of the system are mapped to a rather static design. [1] As a result, the system is not dynamic and cannot adapt itself to any changes in

- the system environment which cannot always be under the control of the application designer,
- changes in the system itself due to failures of resources and components and
- changes in the needs of the end-user.

Secondly, more often than not, these systems treat processor scheduling as the dominant resource scheduling problem, since a wide range of scheduling algorithms ranging from timeline scheduling, non-preemptive scheduling to priority-driven preemptive scheduling is available to perform processor scheduling. However, many resource types other than the processor are always needed for these applications. For example, data buffers and virtual memory pages are often statically allocated or wired down such that no dynamic decisions need to be made. Other resource needs such as the need for communication protocol processing and disk scheduling are often ignored or not used. These fundamental drawbacks of the static pre-defined approach have led to the recent advocacy of dynamic, adaptive system designs which can dynamically adapt the Quality of Service (QoS) guarantees available to the system user.

1.1. Distributed Video-Based Decision-Making Example

Consider a distributed video-conferencing application with support for group-collaborative decision making and whiteboarding. One would prefer

R1. A tight end-to-end audio delay (typically less than 100ms) that facilitates a normal voice conversation between participants,

R2. A reasonable video frame rate (> 12 frames/sec),

[1] The system may operate in one of many possible modes, but these modes are also pre-defined and enumerated.

R3. The whiteboarding feature should allow the conferencing participants to share and view a common workspace (such as the trajectory of a moving object being tracked).

1.1.1. System Resources

Requirements R3 through R6 impose demands on systems resources like the audio/video input/output devices, CPU processing bandwidth, protocol processing bandwidth, network bandwidth, the network interface on each participating node and connecting links. It must be possible to translate application level QoS demands from end-user-specific needs (e.g. high quality audio) to system demands (e.g. 16KHz sampling with 512 byte buffering, 12 frames per second at 1/4 of a screen) to node demands (e.g. 5% of a Pentium 120 Mhz and 2% of a 10Mbps link). Similarly, a medium quality video stream at the user-level can imply a 12 frame per second (at 1/4 screen and 8-bits per pixel) demand at the system level, and 30% of a Pentium 120 and 20% of a 10Mbps link.

1.1.2. Concurrency Management

It is also desirable in this conferencing application that the following requirements are met.

R4. Allow other real-time applications such as sensor data fusion and target tracking to be running concurrently and with their QoS guarantees,

R5. If the system resources are overloaded, maintaining the audio stream quality and latency is often more important than maintaining a high quality video stream.

R6. Any other application which misbehaves or demands more resources must not be allowed to compromise the quality of the conferencing application.

Requirements R4 through R6 imply that a set of mechanisms must be available

(i) to allow a concurrent mix of real-time, multimedia, soft real-time and non-real-time applications each with (some level of) QoS guarantees,

(ii) to track system load conditions,

(iii) to trade off among conflicting resource demands to best meet the application-level requirements, and

(iv) to isolate each application from the resource demands/misbehavior of other applications.

1.1.3. Networking

The collaborative conferencing application may also be using both wired and wireless networks (such as ATM/FDDI within a ship and NTDS across vessels). It is desirable under this scenario that the application framework and its OS interfaces are consistent and uniform across these wired/wireless links and high bandwidth/low bandwidth distinctions. The microkernel-based run-time system must export mechanisms from within the kernel such that the available bandwidth, signal strength, and noise sensitivity can be obtained for any and all of the underlying network links. The performance metrics in networking to be optimized include the latency in obtaining information about any given network, and the latency in detecting possible bandwidth/signal strength problems.

2. Key Requirements of Future Multimedia Kernels

With the advent of time-sensitive data types, namely audio, video and speech, into mainstream computing (vis-a-vis desktop computing, intranet computing and internet computing), we believe that future microkernels by necessity must provide primitives and abstractions that multimedia

applications with real-time applications to be run concurrently with conventional non-real-time desktop applications. We propose that the following requirements must be satisfied by such kernels:

- *Flexible and dynamic real-time support*: This requirement, while it sounds basic, has many profound implications. First and foremost, usage of all resources types in the system must be specifiable, controllable and observable. Secondly, some applications would require guaranteed timing behavior, others just acceptable (soft) timing behavior and still others would not demand any well-specified timing behavior, and all these applications must co-exist on the same machine. Thirdly, the dynamic nature of the application environment will require that resource usage and allocation must be dynamic and relatively quick [Lee96a].
- *Enforced and secure resource usage*: Just as address space has (albeit slowly) migrated to real-time operating systems, we expect that enforcement in the temporal domain will be both useful and necessary for future real-time and multimedia kernels. This will not only enable peaceful co-existence of applications developed by different vendors/organizations but also enable untrusted applications to be executed within well-protected *virtual firewalls* on a machine.
- *Real-time-aware protocol processing*: High-bandwidth switches such as ATM and Fast Ethernet switches use custom ASICs to support the high-speed switching of bits to higher and higher bandwidths. However, when these bits arrive at a network end-point, the number of software layers in the system (e.g. interrupt handlers, drivers, and packet filters) and the protocol stack (e.g. OSI) become significant bottlenecks and the potential for uncontrolled priority inversion is also enormous. Protocol stack layers, in particular, due to the disproportionate amount of time spent in them, must be real-time aware and must be able to discriminate between packets with different timing constraints and process them in appropriate order [Lee96b].
- *High performance*: The dramatic increases in processor speeds and network bandwidths notwithstanding, real-time and multimedia kernels must also provide high-performance inter-process communications, synchronization primitives and memory management support. Traditional microkernels such as Mach extract a heavy price for optimizations such as copy-on-write, which are completely unnecessary for real-time environments. A lean and mean infrastructure for IPC and memory management will make real-time applicaitons be more efficient without compromising the other benefits of microkernels such as extensibility, OS emulation capability and smaller overall size.

3. Processor Reservation in RT-Mach and Lessons Learned

Real-time and multimedia operating systems must provide new primitives and abstractions to support the dynamic range of real-time multimedia applications that is becoming possible. The Real-Time Mach operating system being developed at Carnegie Mellon University has introduced the notion of OS resource reservation to provide temporal protection in real-time/multimedia systems analogous to spatial (address space) protection in general-purpose systems [Mercer93, Mercer95]. A simple version of a “CPU reserve” abstraction has been studied in depth in this context and has given us valuable insight into what is useful and what needs to be extended. In particular, we have learned the following lessons:

- The simple {C,T} (computation time per specified time interval) 2-tuple of the CPU reservation model is useful but also limiting. For example, it does not permit the usage of reserves for threads with a tight jitter requirement. The reservation application programming

interface exported to the programmer must be modified to be extensible and relatively insensitive to future changes in the parametric model used by a reserve model.

- Resources other than the CPU such as disk bandwidth and network bandwidth must also be reserved, and there needs to be mechanisms to group together different reserves for the sake of consistency, efficiency and convenience.
- The enforced behavior of the resource reservation scheme is very powerful and if its power can be exercised uniformly across all resources, then it can be exploited to satisfy other needs such as security in the context of untrusted applications and requests.
- The admission control mechanism used was based on relatively simple assumptions which forced the timing behavior of a thread to be possibly dependent upon the blocking behavior of a different thread with a higher priority reservation. It can couple the timing semantics of a reserve to the coding structure of a thread, when the two would be orthogonal in an ideal setting. When this coupling happens, the semantics of the reservation can lead to unpredictable timing behavior.
- When a reserved client thread invokes a service offered by a server (such as X11), the server must either employ its own reservation (analogous to the priority ceiling protocol) or inherit the priority of the reservation of the highest priority client waiting for its service. This approach addresses the priority inversion problem. However, cleaner mechanisms are needed to make such "reservation inheritance" be done more naturally and easily for arbitrary services.
- Flexible options are needed when CPU reservations expire. Currently, a thread whose reservation expires will always be inserted back into the time-sharing queue and continue to consume some cycles. Also, the thread will receive no notification when its reservation has expired.
- The constant setting and re-setting of timers can be inefficient if access to the hardware timer itself can be relatively expensive. This was indeed the case in our implementation which uses an high-resolution timer card on the ISA bus.

We are currently actively addressing these limitations of the processor reserve mechanism. As an illustration, we next present new kernel primitives that we are in the process of adding to RT-Mach to address some of the problems described above.

4. New Kernel Primitives for Resource Reservation and Management

4.1. Resource Set

A *resource set* is a kernel entity representing a set of resources, where resources include CPU time, physical memory pages, bandwidth of devices (network, disk, and so on). Since a resource set is a first-class kernel entity, it can be passed around in the kernel. While user-space programs can send a reserve set to one another, a resource set cannot be counterfeited and will be enforced by the kernel. Usually, a virtual processor is an abstraction of a CPU. A resource set is an abstraction which represents all the system resources that are accessible to the program(s) bound to the resource set. Assurance and security can be enforced, for example, by making available only a subset of resources for some applications.

Threads on a resource set are scheduled and dispatched by it. The relationship between a resource set and threads is like the one between a virtual processor and user-level threads. As a result, different resource scheduling policies can be applied to the same physical resource used in two different resource sets. For instance, a set of threads bound to one resource set can be using a fixed priority CPU scheduling policy, while another set of threads bound to another resource set can be using a time-sharing CPU policy or the earliest deadline processor policy. Base-level schedulers will determine which resource set (and therefore which scheduling policy) will be active at any given time.

The typical usage of a resource set will be for an appropriately privileged application to create a resource set and use it globally across all threads/tasks in that application or created by that application. In its implementation, if the same resource set is being used by two threads in two different tasks comprising an activity, the context switch between the two tasks will be efficient.

In summary, a resource set is a specialized "virtual machine" which only contains a part of the resources that are actually provided by a hardware platform. If, for example, "disk bandwidth is not one of the resources in the resource set", this means that "the maximum amount of disk bandwidth which is available to applications using this resource set is 0%"'. If the CPU resource in the resource set uses the time-sharing policy, the maximum CPU cycles available to applications using the resource set is the maximum (of 100%) and the minimum guaranteed is 0%.

This notion of resource set as defined would be most useful in non-real-time systems as well as in real-time systems which do not provide any support for resource reservation. Since some resources can be selectively restricted from access, high assurance/security concerns can still be addressed and enforced.

4.2. Hard, Firm and Soft Resource Reservation

Many options are possible while allocating a resource between unreserved threads and threads with expired reservations. The RT-Mach CPU reservation mechanism currently allows a reserved thread to consume more cycles from a time-sharing queue after its reservation has expired. We refer to this as "*soft resource reservation"*. Its usage is rather flexible but uncontrolled once the reservation expires. At the other end of the spectrum lies the notion of "hard" resource reservation. Under *hard resource reservation*, a thread cannot obtain more than its specified reservation independent of whether the resource is fully used or not. As a result, even if the resource is idle, a ready-to-run thread whose reservation has expired will *not* be allocated the resource. An intermediate option is "firm resource reservation", under which a thread whose reservation has expired can use the resource *only if* no other unreserved thread is ready to use the resource. A reserve using hard, firm or soft resource reservation is called a *hard reserve*, a *firm reserve* and a *soft reserve* respectively.

The net result of the various "shades" of resource reservation is that

(a) threads with unexpired reservations have priority over all other threads in using a resource,
(b) if no threads with unexpired reservations are ready to use a resource, the resource can be used by threads with soft reserves or no reserves, and
(c) if no other eligible threads are ready to use a resource (as per the above two rules), the resource can be used by threads with expired firm reserves.

Tighter resource usage semantics such as hard reserves, available as an option on each reservation, can force an application to consume no more than its specified reservation. Such tight usage of resources, when applied globally across reservations of multiple resource types, can form the basis of *virtual firewalls* within which untrusted applications can be executed without fear of damage.

4.3. Reserve Set

A *reserve set* is a kernel entity representing a set of *reserved* resources, where resources include CPU time, physical memory pages, bandwidth of devices (network, disk, and so on). In other words, a reserve set is similar to the resource set except that some well-defined portion of each resource in the set is explicitly allocated and guaranteed to be available to the threads bound to the reserve set[2]. For example, a CPU reserve of 20ms CPU time every 100ms, a disk bandwidth reserve of 100KB every 500ms and a network bandwidth of 100Kb every 100 ms can be the reserved resources in a reserve set. Each reserve in a reserve set can be hard, firm or soft.

4.4. Reserve Ownership and Usage Rights

A reserve must be created by an appropriately privileged thread, and is then owned by the task containing that thread. In other words, the creating task retains *ownership* of the reserve. Reserve rights can be shared by all threads within a task and are also owned by one task. But rights of usage can be passed by threads in this task to other threads. Rights to use a reserve can be passed by the owner of a reserve to other threads. These rights are called "*reserve usage rights*".

5. Discussion

5.1. Binding Scope of Reserves

The "*binding scope"* of a reserve defines the kernel entities that can be bound to a reserve. A thread can bind itself to the CPU reserve such that its timing behavior is controlled by the parameters of the CPU reserve. Thus, the binding scope of the CPU reserve is said to be threads. However, a Mach (and RT-Mach) task owns the address space which is shared by all the threads in the task. Consider the reasonable proposition that a fixed number of physical pages be allocated to a *VM reserve*, which can then be assigned to threads such that they can control their own paging behavior. It is logical to expect that a real-time thread be bound to this VM reserve such that its paging behavior is strictly predictable and under its own control. However, since all the pages in the task are normally shared by all the threads in it, how would (or should) the behavior of other threads in the task affect the real-time thread? In other words, what is the binding scope of the VM reserve? There are two alternatives:

- Suppose that the binding scope of the VM reserve is considered to be tasks, so that all the threads within a task share the same VM reserve. This is advantageous in the sense that a task with multiple threads still controls its overall timing behavior independent of other tasks

[2] No such guarantees of resource availability is explicit in the resource set.

and their threads. However, this binding scope is different from that of CPU reserves and therefore can introduce other side-effects[3].

- Suppose that the binding scope of the VM reserve is considered to be threads such that there can be multiple threads from the same task which are running on different VM reserves. When a thread needs to allocate a memory region for its activity, physical memory pages reserved for the activity should be used for the region. Who supplies physical memory pages for memory regions of common program text and its local data becomes very tricky. Specifically, if a physical page from one VM reserve contains a function used by two threads bound to two different VM reserves, can one VM reserve page out the physical page containing that function because it no longer needs it? The pageout can unfortunately affect the other thread. Duplicating the code segment for either reserve (as read-only) may be feasible but results in wastage of memory resources. Also, this cannot be easily done for data segments with read-write semantics *and* the semantics of sharing.

Given the problems with the second approach of keeping the binding scope of a VM reserve to be threads, we prefer the first approach of making the binding scope of a VM reserve to be tasks. In general, it appears that the binding scope of a resource should remain the same as it was in the absence of resource reservation. In the case of VM reserves, the underlying address space is shared among all threads of a task, and we choose retain that model.

5.2. Enforcement of Reservation and Counting Resource Usage

Enforcing the reservation of resources requires the kernel to monitor and enforce resource usage. Since reserved resources are bound to a reserved set, and it directly represents the execution of an activity in our software architecture, counting resource usage can be performed efficiently. The kernel can charge the use of resources to the currently active reserve set. It is unnecessary to switch or pass reserved resources, for example, when an IPC occurs between two threads on different tasks using the same reserve set.

5.3. Optimization of IPC Path

A reserve set, in typical usage, will be bound to an activity[4], and IPC is a means to connect activity threads to servers which actually perform work for the activity. Thus, the IPC path between a client and its server becomes easy to infer. A client thread can simply imply/pass its resource context to the waiting thread of its server if both threads are to use the same reserve set. In this case, IPC may require no scheduling of threads. Since scheduling threads depends on the policy of the corresponding reserve set, this fast path can be used to obtain low latencies in terms of both lower context switching and IPC costs.

6. Concluding Remarks

Resources such as CPU, disk bandwidth, memory pages and network bandwidth represent the fundamental system entities used and scheduled by operating systems to meet the needs of real-time and multimedia applications. Real-time and multimedia operating systems must be able to

[3] One such side-effect is the need to think about the binding scope of a new reserved resource type!

[4] An activity comprises a set of multiple threads and tasks which coordinate to fulfill a common application goal.

support end-to-end resource management, as well as dynamic quality degradation/upgrading based on changing needs and resources. In order to meet these requirements, we argued in this paper that multimedia operating systems must provide a rich set of resource-centric primitives and abstractions that enable applications to specify, monitor and control their resource usage and thereby their timing behavior. In addition, multimedia operating systems must support enforced and secure resource usage, time-aware communication protocol processing and high-performance mechanisms for synchronization. The CPU reservation mechanism of RT-Mach [Mercer94] was a first significant step in this direction. However, we have also come to realize that it was also the first step in "learning to walk", and generalizations and extensions to the mechanism are sorely needed to make resource management schemes flexible and easy to use in practice. We have proposed new abstractions such as a resource set, a reserve set and resource usage that act as organizing principles for real-time and multimedia microkernels. Our new concepts of hard, firm and soft reserves as well as the binding scope of reserves offer new insights into how resource reservation in a microkernel should be structured in order to retain conceptual control, maximum flexibility, and efficient implementation of resource allocation and management.

References

[Bershad90] B. Bershad, T. Anderson, E. Lazowska, and H. Levy. Lightweight Remote Procedure Call. *ACM Transactions on Computer System*, Vol. 8, No. 1, February 1990.

[Ford94] B. Ford and J. Lepreau. Evolving Mach3.0 to a Migrating Thread Model. In *Proceedings of the Winter USENIX Conference*, January 1994.

[Lee96a] C. Lee, R. Rajkumar and C. W. Mercer, Experiences with Processor Reservation and Dynamic QoS in Real-Time Mach, In *Proceedings of Multimedia Japan*, March 1996.

[Lee96b] C. Lee, K. Yoshida, C. Mercer and R. Rajkumar, Predictable Communication Protocol Processing in Real-Time Mach, In *Proceedings of IEEE Real-time Technology and Applications Symposium*, June 1996.

[Liedtke93] J. Liedtke. Improving IPC by Kernel Design. In *Proceedings of 14th ACM Symposium on Operating System Principles*, December 1993.

[Liedtke95] J. Liedtke. On u-Kernel Construction. In *Proceedings of 15th ACM Symposium on Operating System Principles*, December 1995.

[Mercer93] C. Mercer, S. Savage, and H. Tokuda. Processor Capacity Reserves for Multimedia Operating Systems. Technical Report CMU-CS-93-157, School of Computer Science Carnegie Mellon University, May 1993.

[Nakajima93] T. Nakajima, T. Kitayama, H. Arakawa, and H. Tokuda. Integrated Management of Priority Inversion in Real-Time Mach. In *Proceedings of IEEE Real-Time Systems Symposium*, December 1993

[Tokuda90] H. Tokuda, T. Nakajima and P. Rao. Real-Time Mach: Towards a Predictable Real-Time System. In *Proceedings of USENIX Mach Workshop*, October 1990.