Modular Real-Time Resource Management in the Rialto Operating System

Michael B. Jones, Paul J. Leach,
Richard P. Draves, Joseph S. Barrera, III

May, 1995

Technical Report
MSR-TR-95-16

Microsoft Research
Advanced Technology Division
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Paper presented at the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V), May, 1995.

Modular Real-Time Resource Management in the Rialto Operating System

Michael B. Jones, Paul J. Leach, Richard P. Draves, Joseph S. Barrera, III

Microsoft Research, Microsoft Corporation
One Microsoft Way, Building 9S/1
Redmond, WA 98052

mbj@microsoft.com, paulle@microsoft.com, richdr@microsoft.com, joebar@microsoft.com

Abstract

This paper describes ongoing investigations into algorithms for modular distributed real-time resource management. These investigations are being conducted in the context of the Rialto operating system – an object-based real-time kernel and programming environment currently being developed within Microsoft Research.

Some of the goals of this research include developing appropriate real-time programming abstractions to allow multiple independent real-time programs to dynamically coexist and share resources on the same hardware platforms. Use of these abstractions is intended both to allow individual applications to reason about their own resource requirements and for per-machine system resource planner applications to reason about and control resource allocations between potentially competing applications. The set of resources being managed is dynamically extensible, and may include remote resources in distributed environments. The local planner conducts resource negotiations with individual applications on behalf of the user, with the goal of maximizing the user's perceived utility of the set of running applications with respect to resource allocations for those applications.

1. The Need for Modular Real-Time Resource Management

One of our major research goals for the Rialto operating system is to investigate programming abstractions that make it possible for multiple independent real-time applications to dynamically coexist and share resources on the same hardware platforms. In particular, just as it is possible to today to purchase or write time-sharing applications that successfully coexist with other time-sharing applications, we are researching a real-time software architecture that is intended to make it possible to purchase or write real-time applications that can successfully coexist both with other real-time applications and time-sharing applications. Furthermore, the techniques we are developing are designed to be applicable not just for single-machine applications, but also to distributed applications that make use of remote resources through remote object invocations.

To be usable in a tractable fashion, it is our belief that resource management must be *modular*. By this, we mean that it should be possible to write and use software components (classes, modules, libraries, etc.) that have real-time resource requirements as components of larger real-time modules or programs without having to understand their implementations in order to reason about their real-time resource requirements. This allows the traditional benefits of modularity (abstract interfaces, information hiding, the ability to reimplement, etc.) to be carried forward into real-time programming.

As well as applying to software components, we believe that this kind of modularity of resource requirements must also extend to entire real-time applications. This allows a system *resource planner* application to reason about and participate in overall resource allocations between applications, just as an application can reason about its own internal resource requirements. The planner makes resource allocation decisions between applications on behalf of the user.

2. The Problem Being Solved

This paper focuses on one aspect of the real-time programming model provided by and used within the Rialto operating system. This aspect is:

- An extensible modular distributed real-time resource-management scheme with which programs can reason about their own real-time resource needs and negotiate for resource reservations based on those needs.

2.1 Research Context

This research is being conducted in a larger context of real-time systems work. While not the focus of this paper, it is nonetheless useful to have an overview of this larger systems context so as be able to better understand our resource management strategy. Other features integral to the programming model are:

- A constraint-based real-time scheduler. Time constraints contain a deadline, a time estimate, and an

earliest start time. The scheduler notifies the application if a constraint is unlikely to be met, providing for proactive load shedding in cases of transient overload. Actual time taken is reported back to the application, providing the basis for a realistic real-time feedback mechanism. This is a simplification of the mechanism described in [Jones 93].

- An object invocation mechanism that propagates a thread's real-time scheduling constraints to remote object invocations, both to remote processes and to remote machines. (The object invocation mechanism is a real-time implementation of the Component Object Model (COM), the invocation mechanism used by OLE2 [Microsoft 94].) Taken together, these three facilities enable a consistent end-to-end treatment to be applied to real-time scheduling decisions.
- An I/O system that also schedules I/O operations using the same real-time scheduling constraints.
- A set of I/O and RPC abstractions designed to avoid data copies when transmitting and operating on large quantities of data. This mechanism is derived from Fbufs [Druschel & Peterson 93].

2.2 The Rialto Approach

Resource management can be viewed as a generalization of admission control. Unlike CPU or I/O scheduling decisions, resource management decisions occur infrequently – typically at program startup, exit, or mode change. Programs negotiate for the resources needed to operate on an ongoing basis in a given mode, and then operate in that mode until either exiting or changing modes.

The Rialto programming model is designed to permit incremental development and refinement of the resource management code used by real-time programs. First, the application can be developed (or ported) and its gross real-time resource needs determined. Next, resource management calls can be added to cover the gross real-time resource requirements of the application, which is still a relatively non-invasive change. Finally, real-time scheduling constraints can be added to fine-tune the behavior of critical sections of code in the application.

The model carefully separates mechanisms and policies. This allows varied or dynamic resource management policies to be used without modifying applications.

We intend to use this flexibility to implement user-centric, rather than application-centric, resource management policies. By user-centric, we mean that they attempt to dynamically maximize the user's perceived utility of the entire system, rather than the performance of any particular application. We expect this to lead to policies which focus on maximizing expected normal-case resource utilization, rather than always limiting resource allocations to account for worst-case behavior.

3. Resource Management Design

This section describes our approach to modular real-time resource management, giving examples to help clarify how it would be used in practice.

3.1 Resource Management Abstractions

The following abstractions are used by our approach to real-time resource management:

- **Resource:** A limited hardware or software quantity provided by a specific machine. Individual resources might include CPU time, memory, I/O bus bandwidth, network bandwidth, devices such as video frame buffers and sound cards, or higher-level software-defined resources, which may themselves manage or use other resources.
- **Resource Amount:** An abstraction representing a quantity of a specific resource. This is represented by a number between zero and one, with one representing 100% of a particular resource. Resource amounts are derived by resource providers (see below) from interface-specific quality-of-service specifications supplied by interface clients.
- **Resource Set:** A set of (resource, resource amount) pairs.
- **Activity:** The abstraction to which resources are allocated and against which resources usage is charged. Normally each distinct executing program or application would be associated with a separate activity. Activities may span address spaces and machines and may have multiple threads of control associated with them. Threads execute with rights granted by secured user credentials associated with their activity. Examples of tasks that might be executed as distinct activities are playing a studio-quality video stream, recording and transmitting video for a video-conferencing application, and recording voice input for a speech recognition system.
- **Resource Provider:** The object that manages a particular resource. Operations include allocating amounts of the resource to activities, performing resource accounting as the resource is used by activities, and notifying the resource planner of activities exceeding their resource allocations. The resource provider object would typically be implemented by the device driver that manages the physical resource, by the scheduler (which manages the CPU resource), by other parts of the system software (which manages other physical resources, such as memory), or by the module that implements software-defined resources.

- **Resource Planner:** A server that arbitrates access to the resources of a machine among different activities. Rather than reserving resources directly from the specific resource providers, applications negotiate for resources with the resource planner. The planner, at the conclusion of a successful resource negotiation, in turn contacts the resource providers to grant specific resource amounts to the requested activity, which can then use up to that amount.

  The planner's job is to implement the resource arbitration policy between competing activities. The expected policy goal is to maximize the user's perceived utility of the system as a whole – the policy is user-centric rather than application-centric. The planner makes all resource allocation policy decisions between activities (on behalf of the user); this allows for a clean separation between mechanism and policy.
- **Preferences:** Input from the user to the resource planner as to the desired behavior of the system and of particular applications. Preferences may be either retrieved from a database, or in extraordinary cases, obtained by directly querying the user. Example preferences include statements that a video-phone call should pause a movie unless it's being recorded and that video should be degraded before audio when all desired resources are not available.

These abstractions are designed to make it possible to reason about application and overall system behavior.

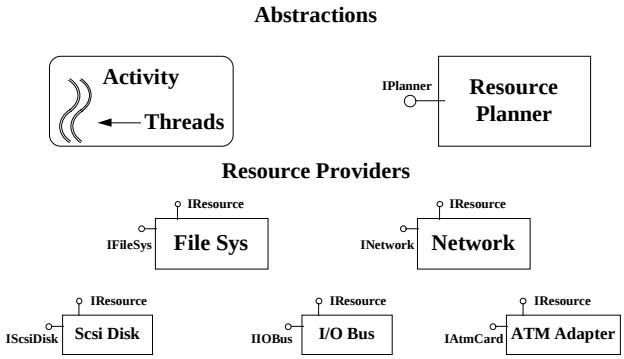

Figure 3-1: Resource Management Scenario

Figure 3-1 shows an activity that uses a set of abstract resources (file system, network) which themselves use other physical resources (SCSI Disk, I/O Bus, ATM Adapter). Exported interfaces are depicted as labeled circles. Note that software modules export both their usual functional interfaces, as well as resource manager interfaces. Also shown is the resource planner service. This scenario will be used to illustrate several aspects of resource management in later figures.

3.2 Modules and Resource Interfaces

Within our resource management framework, software components (classes, modules, libraries, etc.) that have real-time resource requirements provide interfaces exposing those requirements to clients of those components. This allows clients to query the module about the resources needed to perform the operations they will use, so that the client modules can, in turn, make their resource needs known to their clients.

For example, consider a module *M* which implements a network read operation. As well as exporting the network read operation, *M* would also export an operation for determining the resources needed to perform the network read on an ongoing basis. In particular, it would allow client modules *C* to ask *M* questions of the form: "What resources are needed to read *N* bytes every *T* time units" with the response being a resource set *S* enumerating the needed resources. In this instance, *S* might indicate an amount of CPU time, an amount of network bandwidth, an amount of bus bandwidth, and an amount of memory.

Note that resource queries are in terms of operations exported by the modules, and may contain whatever qualifications are necessary to sufficiently specify the operations being asked about. For instance, to accurately quantify the resources needed for a series of network reads, *M* might also need to know the source address from which the data would be read and might need to know acceptable jitter bounds. If so, the corresponding resource query operation would accept this qualifying information as parameters.

Also, note that *C*, in general, does not (and need not) understand the contents of the resource set *S*. To determine their own resource requirements, client modules just add together the resources required by each of the modules (such as *M*) that they use, and add in any resources required for operations directly implemented in the client. Indeed, it is the fact that clients do not need to understand what resources are in a set returned from a resource query that makes the resource management scheme modular. Implementations may change to use different resources without requiring changes in clients.

This gives us a modular algebra for reasoning within a program about the program's resource requirements. As a starting point, resource providers are aware of and understand the resources needed to do their jobs. Client modules subsequently determine their own resource requirements in terms of those of the modules they use. Finally, this permits a program to determine its own resource requirements for the various modes of behavior which the program might choose to exhibit.

This ability for a program to reason about its own resource requirements forms a basis for it to negotiate for the these resources.

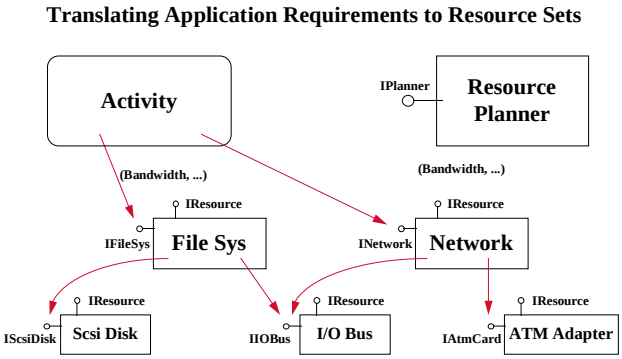

Figure 3-2: Resource Queries

Figure 3-2 shows resource queries being made by the program to its resource providers. Note that some of the providers themselves make queries to other resource providers as part of this process. Resource sets are returned in response to these queries.

3.3 Resource Negotiation

Once an application has determined what resources it will need (either through resource queries, as described above, or by consulting a database of cached resource requirements taken from past runs) it negotiates for those resources with the local resource planner. If the requested resource reservation is granted by the planner, the planner in turn contacts the resource providers on behalf of the program's activity and makes the actual resource reservations. At this point, the application is free to use at least the reserved amounts of the requested resources until such point as it is notified to the contrary by the resource planner.

If, however, the requested resource reservation cannot be granted, either due to conflicts with other programs or because of insufficient capacity, the planner will notify the application of this fact, telling it what quantities of the requested resources the program could successfully acquire. Then, the program either makes a modified resource request (probably based on reasoning about its own resource requirements for running in a different mode than originally negotiated for) or it may decide that there are insufficient resources to function in any mode, and will shut itself down.

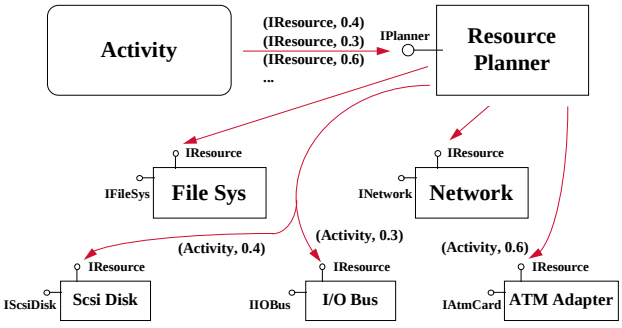

Figure 3-3: Resource Reservation

Figure 3-3 shows a program requesting a resource reservation from the resource planner and the planner in turn contacting the individual resources to make the actual resource reservations.

Unlike simple first-come first-served admission control schemes, our scheme does not have the property that once a resource is reserved for an application that the application is guaranteed at least that resource amount until it explicitly relinquishes it. We view this as an application-centric policy. Instead, we have opted for a user-centric policy – ideally the resource planner allocates resources among the competing applications in the way that provides the most perceived value to the user. (Of course, the planner *can* implement irrevocable reservation for some resources or some applications if it is deemed appropriate, but this is merely a special case of more flexible policies.) Design and implementation of these policies is an important area of future work.

Under our scheme, there are several different scenarios where resource re-negotiation may occur. First, a program may modify its own behavior or enter a new mode, causing its resource needs to change. In this case, the program contacts the resource planner to request that its resource reservations be revised.

Second, another program may have been started, may have exited, or may have changed its resource usage pattern. In this case, the planner may contact running programs, requesting that they modify their resource usage in specific ways (or notifying them that they may request more resources if they choose to do so).

Third, a resource provider may detect a persistent overload condition, at which point the resource provider would contact the resource planner making it aware of the activities that are exceeding their reservations.

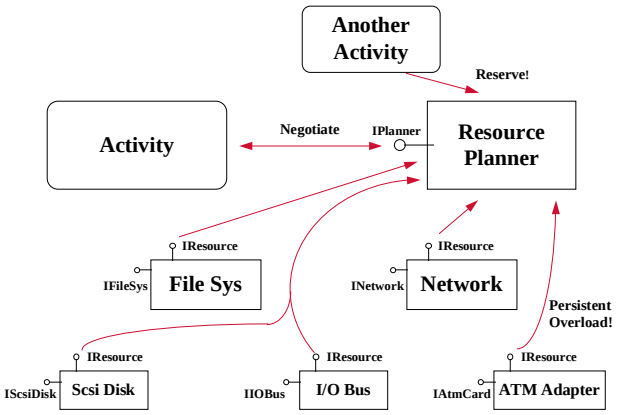

Figure 3-4: Resource Negotiation

Figure 3-4 depicts several scenarios under which resource re-negotiation may occur. First, a program may modify its own behavior or enter a new mode, causing its resource needs to change. Second, another program may have been started, at which point resources may be

reallocated by the planner among existing activities. Third, a resource provider may detect a persistent overload condition, at which point it would contact the resource planner making it aware of the activities that are exceeding their reservations.

3.4 Distributed Resource Management

In our scheme, each resource is represented by a resource object that is registered with a resource planner that is (typically) running on the machine where the resource resides. Resource queries for locally implemented objects return references to local resource objects and resource reservation is done via the local resource planner. Thus, most resource management decisions require only local object invocations.

However, resource queries to remotely implemented objects will cause remote object invocations and will consequently return references to the remote resource objects needed to implement the requested service. Applications in general are not aware of which resources are local or remote, but the local resource planner is. If a reservation request is made to the local planner for remote resources, the planner forwards this portion of the request to the remote planner. Because the planners cooperate to transparently manage remote resource reservations, application resource management code is resource-location-independent.

3.5 Simplifying Assumptions

A number of simplifying assumptions underlie our resource management model. This section describes and motivates these assumptions.

- Linearity of resource amounts – For most resources this should be a reasonable approximation to reality when not close to overload. This assumption permits the resource planner to manage resource allocations without deep understanding of individual resources.
- Independence of resources – Like linearity, we believe this to be reasonably true for many resources. Where not true (for instance, reading from disk causes DMA, which can reduce effective processor speed) we may need to handle this at the resource provider level by explicitly modeling interdependencies (for instance, by also reserving some "CPU" time for DMA transfers). This assumption permits the resource planner to manage allocations of different resource independently (even though resource providers and consumers may be aware of the interdependencies).
- Application resource self-awareness – We believe that cost in complexity of having applications be aware of their own resource requirements and usage is reasonable in comparison to the benefits gained. This self-awareness permits applications to reason about their own behavior in the presence of different resource allocations. Note also that an incremental approach can be taken, adding refinements of resource awareness to a program on an as-needed basis.

One of the research goals of this work is identifying which simplifying assumptions yield reasonable results, and under what circumstances they hold.

4. Related Work

This section examines the relationships between this work and other related work.

Mercer [Mercer et al. 94] has advocated a "temporal protection" scheme in which enforcement of CPU and possibly other resource reservations is provided between competing programs. Our resource management strategy is largely independent of whether hard enforcement of resource usage is provided, but is compatible with it. Indeed, if both are present, resource amounts derived from resource negotiation would be used to choose the values used for resource enforcement.

Unlike Mercer, Compton and Tennenhouse believe that resource protection is inappropriate and that applications should dynamically and cooperatively shed load when necessary [Compton & Tennenhouse 93], but they bemoan the crude measures available for deciding when to shed load. Rather than shedding load reactively, our work provides a means for programs to cooperatively reason about their resources in advance, proactively avoiding most dynamic load shedding situations.

A number of mechanisms are currently being proposed for reserving network bandwidth and related resources such as RSVP [Zhang et al. 93] and a number of ATM-specific schemes. This work is complementary to such mechanisms. Indeed, one result of distributed resource negotiation can be using these mechanisms to allocate any network resources needed by the activity.

Anderson described a system for trading off buffer space and variabilities in network latency when delivering continuous media streams [Anderson 93]. The application resource self-awareness needed to analyze these tradeoffs is an example of the kind of self-awareness needed to be able to negotiate for and make tradeoffs among resources in our more general setting.

One important aspect of this work is that it provides a more flexible admission control scheme than the first-come first-served or priority schemes that are common today. Admission policy is controlled by the resource planner, which is able to redistribute resources among both existing applications and new applications in a user-centric rather than application-centric manner.

Finally, it should be stated that this work is intended to be complimentary to, and not a replacement for, algorithms which provide fine-grained CPU scheduling, whether classical priority-based schemes or more flexible schemes, such as those employed by Northcutt [Northcutt

et al. 90, Wall et al. 92]. Even given sufficient resources, fine-grained scheduling decisions still must be made correctly to ensure that application deadlines are met.

5. Status

The Rialto operating system kernel, including its real-time constraint-based scheduler, has been implemented and is in use as a research testbed for a number of kinds of real-time applications. The implementation of resource management is under way. We expect to report initial results at the workshop.

6. Conclusions

This paper presents a design for modular resource management within and between applications. The set of resources managed is dynamically extensible and may include remote resources in distributed environments. The design carefully separates mechanisms and policies, allowing varied or dynamic resource management policies to be used without modifying applications. We intend to use this flexibility to implement user-centric, rather than application-centric, resource management policies.

While ambitious, we believe that the goals of this research are both attainable and practical. We believe that dynamic resource management will allow combinations of independently authored real-time applications to nonetheless coexist and be concurrently executed on the same platform. Resource management can be an enabling technology for a free market in independently authored real-time components and applications for widely available home multi-media information platforms.

References

[Anderson 93] D. P. Anderson. Metascheduling for Continuous Media. In *ACM Transactions on Computer Systems*, 11(3):226-252, August, 1993.

[Compton & Tennenhouse 93] Charles L. Compton and David L. Tennenhouse. Collaborative Load Shedding. In *Proceedings of the Workshop on the Role of Real-Time in Multimedia/Interactive Computing Systems.* IEEE Computer Society, Raleigh-Durham, NC, November 1993.

[Druschel & Peterson 93] Peter Druschel and Larry L. Peterson. Fbufs: A High-Bandwidth Cross-Domain Transfer Facility. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles.* December, 1993.

[Jones 93] Michael B. Jones. Adaptive Real-Time Resource Management Supporting Composition of Independently Authored Time-Critical Services. In *Proceedings of the Fourth Workshop on Workstation Operating Systems*, pages 135-139. IEEE Computer Society, Napa, CA, October, 1993.

[Mercer et al. 94] Clifford W. Mercer, Stefan Savage, Hideyuki Tokuda. Processor Capacity Reserves: Operating System Support for Multimedia Applications. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems (ICMCS)*, May 1994.

[Microsoft 94] *OLE2 Programmer's Reference, Volume One.* Microsoft Press, 1994.

[Northcutt et al. 90] J. D. Northcutt, R. K. Clark, D. P. Maynard, and J. E. Trull. *Decentralized Real-Time Scheduling.* Final Technical Report to RADC, RADC-TR-90-182, School of Computer Science, Carnegie-Mellon University, August, 1990.

[Wall et al. 92] Gerald A. Wall, James G. Hanko, and J. Duane Northcutt. Bus Bandwidth Management in a High Resolution Video Workstation. In *Proceedings of the Third International Workshop on Network and Operating System Support for Digital Audio and Video*, pages 236-250. IEEE Computer Society, San Diego, CA, November, 1992.

[Zhang et al. 93] Lixia Zhang, Steve Deering, Deborah Estrin, Scott Shenker, and Daniel Zappala. RSVP: A New Resource ReSerVation Protocol. IEEE Network 7(5), Sept., 1993.