Middleware versus Native OS Support: Architectural Considerations for Supporting Multimedia Applications*

Prashant Shenoy, Saif Hasan†, Purushottam Kulkarni, Krithi Ramamritham‡

Department of Computer Science
University of Massachusetts,
Amherst, MA 01003
{shenoy,purukulk,krithi}@cs.umass.edu

†Microsoft Corporation
One Microsoft Way,
Redmond, WA, 98052
shasan@windows.microsoft.com

Abstract

In this paper, we examine two architectural alternatives—native OS support versus middleware—for supporting multimedia applications. Specifically, we examine whether extensions to OS functionality are necessary for supporting multimedia applications, or whether much of these benefits can be accrued by implementing resource management mechanisms in a middleware system. To answer these questions, we use QLinux and TAO as representative examples of a multimedia operating system and a multimedia middleware, respectively, and examine their effectiveness in supporting distributed applications. Our results show that although the run-time overheads of a middleware can impact application performance, middleware resource management mechanisms can, nevertheless, be as effective as native OS mechanisms for many applications. We also find OS kernel-based mechanisms to be more effective then middleware systems at providing application isolation and at preventing applications from interfering with one another.

1 Introduction

1.1 Motivation

Since the emergence of multimedia applications more than a decade ago, applications such as streaming media players, distributed games, and online virtual worlds have become commonplace today. Multimedia applications access a combination of audio, video, images and textual data and have timeliness constraints. Until recently, the demanding computing and storage requirements of these applications as well as their soft real-time nature necessitated the use of specialized hardware and software. For instance, continuous media servers were used to stream audio and video instead of general-purpose file servers, while audio-video playback required the use of specialized hardware decoders. Due to the rapid improvements in computing and communication technologies as dictated by Moore's Law, it is now feasible to employ general-purpose, commodity hardware, software and operating systems to run such applications. For instance, today's processors can easily decode full-motion DVD-quality (MPEG-2) video; in fact, popular streaming players, such as Real and WindowsMedia, employ a software-only architecture and need no special hardware. Since commodity (COTS) operating systems such as Linux and Windows were originally designed for traditional best-effort applications, an important issue is how to enhance them to meet the needs of multimedia applications.

In the simplest case, the soft real-time needs of multimedia applications can be met by running such applications at low utilization levels—the absence of resource contention from other applications allows a COTS operating system to easily meet all the needs of a multimedia application and no enhancements to the OS are necessary. However, running multimedia applications in the presence of traditional best-effort tasks (e.g., DVD playback in the presence of compute-intensive background tasks) causes resource contention and jitter, resulting in unsatisfactory performance. Two fundamentally different approaches can be employed to address this problem (see Figure 1).

- *Native operating system support:* In this approach, resource management mechanisms in existing operating systems are augmented to support multimedia applications. The augmented mechanisms employ service differentiation to provide a "better than best-effort" service or explicit QoS guarantees to soft real-time applications. Typically this is done in an incremental manner so as to preserve the semantics and the API provided to best-effort applications, ensuring backward compatibility. Examples of this approach include the real-time priority class employed by Windows 2000 [16], the real-time scheduling class in Solaris [19] and the reservation and proportional-share schedulers developed for Linux [18] and FreeBSD [1].

*This research was supported in part by a NSF Career award CCR-9984030, NSF grants CCR-0098060, EIA-0080119

†This research was carried out when Saif Hasan was a graduate student at the University of Massachusetts.

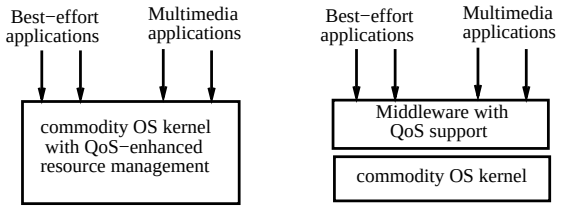
‡Also with the Indian Institute of Technology Bombay, Mumbai, India.

- *Use of a middleware system:* In this approach, a middleware layer between the application and the operating system arbitrates access to system resources (see Figure 1(b)). Since all requests for system resources are made via the middleware, the middleware has complete control over how to allocate resources to various applications. Such a middleware system can employ sophisticated resource management mechanisms to provide QoS guarantees to soft real-time applications, while continuing to provide a best-effort service to other applications. In addition, the middleware system can provide other useful services, such as naming, not provided by the operating system. Examples of this approach include real-time CORBA (TAO) [13] and MidArt [10, 14].

These two approaches represent fundamentally different philosophies for supporting multimedia applications, namely "change the OS" versus "leave the OS unchanged and use mechanisms that build on top of the OS instead". The approaches also represent two viewpoints in a broader philosophical debate—are additional extensions to OS functionality warranted, or can most of these benefits be accrued by implementing resource management mechanisms in a middleware without modifying the operating system? This is a non-trivial question to address since both approaches have several advantages and disadvantages.

Typically, OS extensions are feasible only when the system designer has access to the kernel source code—although this is less of an issue for open-source operating systems, source code access to proprietary or commercial operating systems is often difficult (the only option in such a scenario is to enhance OS functionality *without* modifying the kernel). OS extensions are, nevertheless, efficient by virtue of entrusting all resource management to the OS kernel. The need to seamlessly coexist (i.e., be backward compatible) with kernel management mechanisms can, however, limit the choice of feasible OS enhancements. In contrast, the use of middleware systems is appealing since no modifications to the OS kernel are necessary, making the approach feasible on any COTS system with appropriate resource management mechanisms. However, the presence of a middleware layer imposes an additional overhead, which is turn degrades application performance.

In the recent past, both approaches have been investigated in detail, resulting in numerous commercial products and research prototypes. Surprisingly, however, there has been relatively little effort at a systematic study that quantifies the tradeoffs of the two approaches—existing efforts have implicitly assumed that one or the other approach is necessary without first considering the merits and demerits of both approaches. This paper attempts to address this issue by presenting a detailed comparative study of the two approaches. The goal of our work is not to recommend one approach over the other; rather it is to articulate and quantify the tradeoffs of the two approaches so as to provide guidelines to future system designers.



(a) Multimedia Operating System (b) Multimedia middleware

Figure 1. Two canonical approaches for supporting multimedia applications. A multimedia operating system employs enhanced resource management mechanisms within the kernel, whereas a multimedia middleware employs QoS mechanisms at the user-level to support soft real-time multimedia applications. Both approaches also support traditional best-effort applications.

1.2 Research Contributions

In this paper, we compare the two different approaches for supporting multimedia applications—native OS support and the use of a middleware—along dimensions such as portability, complexity, performance, ease of use, and backward compatibility. Since metrics such as ease of use and compatibility are difficult to quantify, we provide *qualitative* arguments for these metrics and quantify the tradeoffs of the two approaches with respect to performance. Performance, however, is not the sole criterion for choosing between the two approaches and many other factors influence this decision. Consequently, our goal is to articulate the tradeoffs of the two approaches along various dimensions and quantify them wherever possible so as to enable system designers to make good engineering tradeoffs. It should be noted that the two approaches are at the two ends of a spectrum; hybrid approaches are also possible where some of the enhancements are implemented within the OS kernel and the rest are implemented at the user-level [7].

Specifically, this paper attempts to answer the following questions:

- Under what scenarios and for what kind of applications is one approach more suitable then the other?
- What are the run-time overheads of the middleware approach and what are their implications on application performance? Specifically, can a middleware system yield performance that is comparable to an OS-based approach despite its run-time overheads?

One approach to address these issues is to develop new resource control mechanisms for OS kernels and middleware and compare their performance. However, the goal of

our work is not to develop new mechanisms, rather it is to compare existing systems and mechanisms that belong to the two categories and evaluate their tradeoffs. To do so, we use TAO as a representative example of a multimedia middleware and QLinux as a representative example of a multimedia operating system and conduct an experimental evaluation of these systems. Our evaluation of these systems uses a mix of best-effort and multimedia applications ranging from streaming media servers, industrial control applications and web servers. Our results show that although the run-time overheads of TAO can impact application performance when compared to QLinux, TAO can, neverthless, provide performance comparable to QLinux for many applications. Specifically, we see the TAO yields (i) streaming performance that is comparable at low loads but slightly worse at heavy loads and (ii) comparable performance for industrial control applications. A closer examination of why TAO can provide comparable performance, despite middleware overheads, reveals the following. We find that some of these these benefits are due to the higher level abstractions provided by a middleware. An untuned application using these high-level abstractions benefits from the optimizations performed on these abstractions by middleware developers, whereas the same untuned application running on a commodity OS kernel needs to implement this functionality (and does so without any tuning). However, some of these benefits are outweighed by the higher run-time overheads of a middleware at moderate to heavy loads, which can result in a degradation in application performance.

Our experiments also indicate that, for certain applications such as industrial control systems, middleware resource management mechanisms can be as effective as native OS mechanisms. We also find that TAO is less effective at providing application isolation (i.e., preventing interference when multiple competing applications access OS resources concurrently). Since our study is specific to the two systems, QLinux and TAO, we note that our results should be interpreted accordingly—our quantitative results should not be treated as broadly applicable to other systems, although many of our observations may apply in qualitative terms.

The rest of this paper is structured as follows. Section 2 discusses the qualitative differences between the two approaches. We present the results of our experimental evaluation in Section 3. Section 4 discusses our experiences with the two platforms used in our experimental study and the lessons learned in the process. Section 5 discusses related work, and finally, Section 6 presents some concluding remarks.

2 Qualitative Considerations

In this section, we articulate the qualitative differences between the two approaches and provide several examples of existing systems that employ these approaches.

2.1 OS Extensions for Multimedia

OS kernel enhancements to support multimedia applications have the following advantages and disadvantages.

- *Efficiency and performance:* Handling all resource management mechanisms within the kernel allows this approach to efficiently arbitrate resources among contending applications, resulting in low overheads. Furthermore, the benefits of providing service differentiation to applications typically outweigh the run-time overheads imposed by the approach [18].

- *Complexity for the application developer:* Observe that this approach requires incremental enhancements to the OS system call interface to expose the new resource management mechanisms to applications. Since the interface is enhanced in an incremental manner, applications continue to deal with standard OS interfaces (API), resulting in lower complexity for the application developer. An added benefit of the approach is that all existing and legacy applications continue to run on the OS without any modifications.

- *Complexity for the system designer:* In certain instances, it may be possible to enhance OS functionality via dynamically loadable modules *without* modifying the kernel source.[1] In general, however, OS enhancements are feasible only when one has access to the kernel source code. In either case, OS modifications require a solid understanding of the intricacies of the kernel and of the possible interactions between the new and the existing mechanisms. Since OS kernels tend to be complex software systems, this imposes a significant challenge on the system designer.

- *Choice and effectiveness of resource management mechanisms:* One of the main challenges of this approach is that new OS enhancements need to coexist with existing mechanisms. The need for coexistence and backward compatibility can limit the choice of feasible OS enhancements, making the task of the system designer more complex. Moreover, these compatibility limitations can reduce the overall effectiveness of the approach. For instance, depending on the kernel architecture, only mechanisms that provide a "better than best effort" service may be feasible instead of those that provide explicit QoS guarantees.[2]

[1] To illustrate, Ensim ServerXchange, a commercial product, employs this approach—OS functionality is extended via dynamically loadable modules that provide QoS support [3]. HP's Linux CPU scheduler interface also embraces this philosophy by allowing new CPU schedulers to be written as dynamically loadable modules [11].

[2] A better-than-best-effort-service is one where applications receive qualitatively better service than vanilla best-effort service, although no quantitative guarantees are provided.

- *Portability and Reuse:* Since OS kernels have different architectures, enhancements made to one kernel are not directly applicable to another (the basic concepts may apply but the implementation is not portable). Thus, the approach does not permit reuse and is not portable across OS kernels. Portability can be problematic even across different versions of the same kernel.

Finally, we note that our study focuses on approaches that incrementally extend the functionality of existing OS kernels. It is also possible to design a completely new OS kernel or make radical changes to an existing OS kernel. Whereas such an approach eliminates the hurdles faced in incrementally extending OS functionality, designing a new or radically redesigned kernel is very expensive (in terms of programming effort) and can result in compatibility problems with existing applications.

2.2 Multimedia Middleware

The use of a middleware system to support multimedia applications has the following advantages and disadvantages:

- *Efficiency and performance:* The need to use an additional software layer to access system resources imposes a run-time overhead, which in turn lowers application performance. One of the goals of this paper is to quantify the impact of this overhead on application performance.

- *Complexity for the application developer:* Typically each middleware layer exports an API for accessing system resources and middleware services. Although the API may be similar in design to commonly used application libraries and OS system call interfaces, there are often subtle differences in the syntax and semantics of the API [12]. Consequently, applications may need to be programmed to a new API, which increases complexity for application developers. Furthermore, existing and legacy applications need to be modified to use this API if they are to run on the middleware, which results in compatibility problems. An alternate approach is to bypass the middleware system and run legacy applications directly on the OS. However, this can interfere with the QoS guarantees provided by the middleware system to soft real-time applications (since the middleware no longer controls access to system resources for all applications).

- *Complexity for the system designer:* Since a middleware system is a separate software component from the OS kernel, there are fewer interdependences between the middleware and the kernel, reducing complexity for the middleware designer. However, depending on the functionality and services provided, a sophisticated middleware system can be a complex software system.

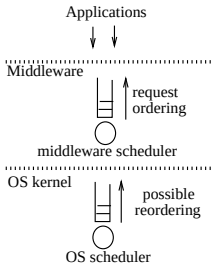


Figure 2. Possible interactions between the middleware scheduler and the OS scheduler. The two effectively form a two-level scheduler.

- *Choice and effectiveness of resource management mechanisms:* Whereas the choice of feasible OS kernel enhancements can be limited by compatibility issues, no such limitations apply to middleware system—in principle, the middleware system can employ any resource management mechanism to provide QoS support for multimedia applications. In reality though, implementing a particular resource management mechanism in the middleware is complex, since it depends on the scheduling policy implemented by the underlying OS kernel. As shown in Figure 2, the middleware scheduler and the OS scheduler effectively form a two-level scheduler; the overall order in which requests get scheduled depends on the combined effect of the two schedulers. In general, requests ordered by the middleware scheduler may be *reordered* by the OS scheduler, reducing the overall effectiveness of the middleware system.

  Certain OS scheduling policies can make the task of middleware resource management easier. For instance, if the OS scheduler is FIFO, it is possible to implement any arbitrary scheduling policy in the middleware—once requests are ordered by the middleware scheduler, they get serviced in the same order inside the kernel due to the FIFO policy. Similarly, if the OS scheduler is a strict priority scheduler, it is possible to implement any arbitrary scheduling policy in the middleware—based on the middleware scheduling policy, the next request to be scheduled is elevated to the highest priority level, causing the OS to schedule this request next. In general, however, an arbitrary OS scheduling policy makes the task of the middleware resource manager more complex.

  One approach to prevent reordering of requests within the kernel is to simply issue one request at a time to the OS, based on the order determined by the middleware—the presence of a single outstanding request eliminates the possibility of request reordering (all scheduling policies reduce to FIFO in the presence

of a single request). Whereas this approach suffices for certain OS resources, it can reduce the throughput and utilization of resources where concurrency is important. For instance, in the case of disks, the presence of multiple outstanding requests allows the disk scheduler to perform seek optimizations and reduce the seek overhead incurred per request (e.g., the SCAN scheduling policy). For such resources, there is a conflict between the need to improve throughput and the effectiveness of the middleware scheduler. Thus, managing resources using a middleware is a complex issue and requires an intimate knowledge of OS scheduling mechanisms.

Finally, observe that the middleware resource manager is effective only if all requests for system services are made via the middleware. If some applications bypass the middleware and request OS resources directly, this interferes with QoS guarantees provided by the middleware to soft real-time applications. Such interference may be inevitable if the machine runs legacy applications.

- *Portability and reuse:* Since a middleware system does not require any modifications to the underlying OS, the approach is portable and can be reused on different COTS systems. Similarly, applications designed to run on middleware system are portable to any platform supported by the middleware.

2.3 Examples

In the recent past, several systems—both from the commercial and research domains—have been developed to manage CPU, network interface bandwidth and disk bandwidth based on the two approaches (see Section 5 for examples of these systems). In this paper, we restrict our focus to only one of these three resources, namely network interface bandwidth and examine its impact on application performance. We choose network bandwidth over other resources since studies have shown that the network is typically a bottleneck resource for many distributed multimedia applications such as network servers. Hence, a study of network interface bandwidth management using the two approaches provides a good overview of their tradeoffs. Further, there are freely-available systems such as QLinux and TAO (real-time CORBA) that belong to the two approaches and make our study feasible. Next, we provide a brief overview of the two systems used in our study.

2.3.1 QLinux

QLinux is a QoS-enhanced kernel based on the popular Linux operating system [18]. QLinux replaces the standard CPU, disk and network interface schedulers within Linux with QoS-aware schedulers. Specifically, QLinux employs four key components: (i) hierarchical start-time fair queuing (H-SFQ) CPU scheduler that allocates CPU bandwidth fairly among application classes, (ii) hierarchical start-time fair queuing (H-SFQ) packet scheduler that can fairly allocate network interface bandwidth to various applications, (iii) Cello disk scheduler that can support disk requests with diverse performance requirements, and (iv) lazy receiver processing for appropriate accounting of protocol processing overheads [18]. Since the focus of our work is on managing network interface bandwidth, in this paper, we are only concerned with the H-SFQ packet scheduler, which we describe next.

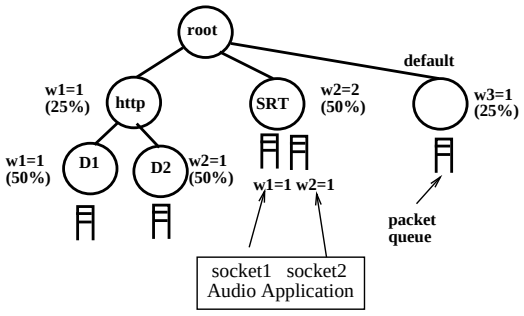
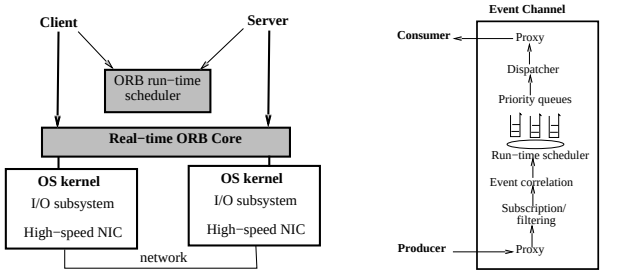


Figure 3. The H-SFQ CPU scheduler in QLinux. The packet scheduler shows a sample scheduling hierarchy with three classes, namely HTTP, soft real-time and default, and two sub-classes within HTTP. The weights indicate the partitioning of bandwidth among classes.

An OS kernel employs a packet scheduler at each network interface to determine the order in which outgoing packets are transmitted. Traditionally, operating systems have employed the FIFO scheduler to schedule outgoing packets. To meet the needs of multimedia applications, QLinux employs H-SFQ to schedule outgoing packets. H-SFQ is a fair, proportional-share scheduler based on generalized processor sharing (GPS). H-SFQ allows a weight to be assigned to each outgoing flow (more specifically, a socket) and allocates bandwidth to flows in proportion to their weights. Hence, a socket with a weight w_i is allocated $\frac{w_i}{\sum_j w_j}$ fraction of the interface bandwidth. The scheduler is hierarchical in that sockets can be hierarchically grouped into classes that are allocated an aggregate weight (see Figure 3). Bandwidth unused by a class or a flow is reallocated to other classes to improve network utilization. QLinux ensures backward compatibility by instantiating a FIFO class in the H-SFQ hierarchy—outgoing packets are queued up at the FIFO scheduler by default. An application requiring QoS guarantees needs to associate its sockets to a different application class and assign them an appropriate weight.



(a) TAO architecture (b) The TAO event channel

Figure 4. The TAO architecture and the real-time event channel.

2.3.2 TAO/real-time CORBA

TAO is a freely available, open-source, real-time implementation of CORBA that provides predictable quality of service to applications (see Figure 4(a)). Conventional CORBA implementations have provided only a best-effort service to applications. The barrier to real-time support within CORBA has been the challenge of providing real-time guarantees that transcend the layering boundaries within CORBA. TAO overcomes this drawback by integrating the network interface, the I/O subsystem, the ORB and the middleware services so as to provide QoS guarantees that span all layers from the application to the operating system. The salient features of TAO include: (i) the use of active demultiplexing and perfect hashing to dispatch requests to objects in constant time, (ii) a concurrency model that minimizes context switching and locking (the model can be configured to incur only a single context switch and no locking or memory allocation on the fast path), (iii) the use of a non-multiplexed connection model that avoids priority inversion and behaves predictably when used with real-time applications [13]. TAO provides these features and optimizations while conforming to the CORBA 2.3 standard. Moreover, TAO is backward compatible with conventional CORBA and hence is able to support all existing (best-effort) CORBA applications.

Although TAO implements a rich set of features and optimizations, the two particular features of TAO relevant to the present study are the event service [4] and the audio-video streaming service [9]. The event service enables client-server communication based on a publish-subscribe paradigm—producers publish events and consumers receive events to which they have subscribed (see Figure 4(b)). An *event channel* provides a mechanism that decouples consumers from producers; by using an event channel as an intermediary, neither producers nor consumers need to be aware of one another. TAO uses a push model for the event channel—producers push events (or data) to the event channel, which pushes them to appropriate consumers. TAO enhances the standard CORBA event service with features such as real-time event dispatch and scheduling, event filtering, event correlation and periodic event processing. Applications using TAO can associate deadlines with events; the deadlines determine the scheduling and dispatch of events within the event channel. Further, consumers can instantiate filters to receive only those events that are of interest to them. Event correlation allows an event to be delivered depending on the occurrence of a related event. TAO's real-time event service can be employed to provide predictable service to distributed multimedia applications such as streaming servers and audio/video players.

The CORBA audio-video streaming service is designed to support audio/video streaming. The service integrates well-defined modules, interfaces, and semantics for stream establishment and control with efficient data transfer protocols for streaming data transmission [9]. It supports the notion of pluggable protocols to allow different streaming protocols to be supported by the service; the current implementation supports various transport protocols such as UDP, TCP and RTP on the data path. The control path supports client-server signaling for stream operations such as play, stop and rewind.

3 Experimental Evaluation

Having discussed the qualitative differences between the two approaches, in this section, we quantify the tradeoffs of the OS- and middleware-based approaches with respect to application performance. Using QLinux and TAO as representative examples of the two approaches, we attempt to answer the following questions: (i) what are the run-time overheads of middleware resource management mechanisms and what are their implications on application performance? and (ii) can a middleware-based approach yield performance that is comparable to a OS-based approach? If so, under what operating regions? Next, we present our experimental methodology and then our experimental results.

3.1 Experimental Methodology

The testbed for our experiments consists of a cluster of Linux-based 350 MHz Pentium III PCs, each with 96MB RAM and 512 MB swap space, interconnected by 100Mb/s switched ethernet. The version of QLinux used for our experiments is based on the 2.2.0 Linux kernel, while the version of TAO used is 1.1 (version 1.2 is used for experiments related to streaming, since the audio-video streaming service was first supported in version 1.2). Both versions are compiled with the GNU C/C++ compiler (version 2.95.2) with the highest level of optimizations.

The workload for our experiments consists of the following applications: (i) *streaming:* a streaming video server and client, (ii) *industrial control:* a real-time application that emulates an industrial control console system and a data sensor

and monitoring system and (iii) the *Apache* web server that is chosen a legacy application. Experiments with additional applications are reported in [15].

To ensure a fair comparison between QLinux and TAO, we ensured that applications developed for the two platforms were identical in all respects except for their communication routines (which were based on the specific functionality provided by QLinux and TAO).

Next, we present our experimental results.

3.2 Performance of Video Streaming

First, we examine the performance of video streaming on the two platforms. Our application consists of a video server that streams a 1.5 Mb/s, 30 frames/s MPEG-1 video to multiple clients. The QLinux version of the application consists of a multi-threaded server that services clients in periodic rounds. The duration of a round is set to 1 second in our experiments. In each round, the server retrieves the next 30 frames for each client from disk; frames retrieved in a given round are transmitted to clients in the following round. Due to the real-time nature of streaming, all frames need to be transmitted to clients by the end of each round. The QLinux server uses the H-SFQ packet scheduler to provide the desired QoS guarantees to each client. To do so, the server associates a weight with each socket over which video data is transmitted; the weight depends on the bit-rate of each stream and ensures that the stream is allocated the necessary transmission bandwidth. In contrast, the TAO version of the server employs the audio-video streaming service. Like in the QLinux version, the server proceeds in periodic rounds. In each round, the server retrieves frames from disk and transmits them to clients in the next round. Like in the QLinux server, the server transmits each set of 30 frames at their real-time rates; In both cases, the server was run at the highest priority level of the CPU scheduler.

We vary the number of concurrent clients accessing the server and measure the total time required to transmit and deliver data in each round. Figure 5 plots our results. Figure 5(a) shows that both QLinux and TAO are able to meet the real-time requirements imposed by streaming at low and moderate loads (by virtue of delivering frames before the end of each round). The figure also shows that TAO results in a larger fraction of deadline violatations at higher loads.

Fig 5(b) and (c) shows the histogram of frame arrivals for the QLinux and TAO, respectively, at a moderate load of 4 concurrent clients. These plots show that the increased computational overheads in TAO cause more frames to miss their deadlines than QLinux. About 3.7 % of the rounds have frames missing their deadlines in QLinux, whereas there are deadlines misses in 4.1% of the rounds in TAO. Thus, we conclude that TAO can provide streaming performance comparable to QLinux at low loads, but the middleware overheads degrade streaming performance at higher loads.

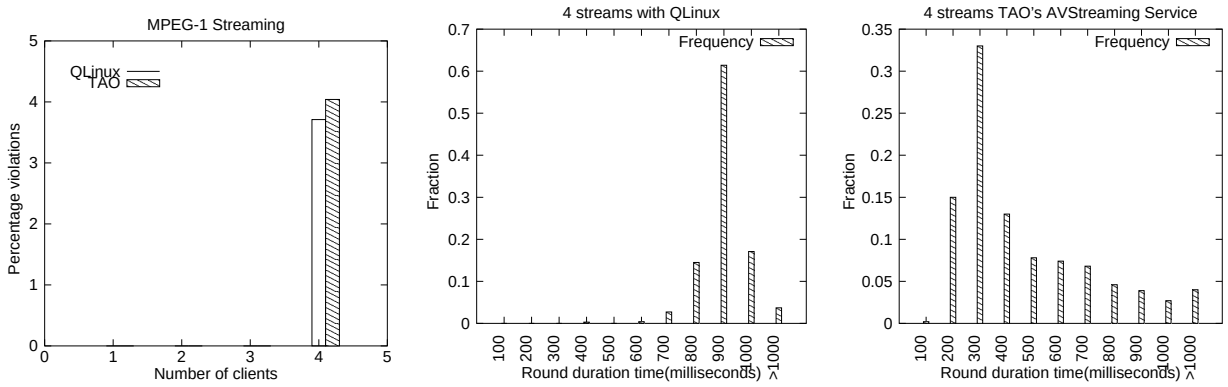
3.3 Performance of an Interactive Real-time Application

In this experiment, we study the performance of an interactive real-time application on the two platforms. The objective of this experiment is two-fold: (1) understand the efficacy of the two platforms in servicing interactive applications, and (2) study how the presence of background applications affects the performance of interactive applications. We use an industrial control console system as a representative application for our experiment (see [10] for a detailed description of this application). The applications consists of two distinct components. The first component is a supervisor's command console that periodically issues commands to a remote actuator device (emulated by a remote server). The actuator performs the operation requested by the client and then sends an acknowledgment back to the console. Commands generated by the console have inter-arrival times that are uniformly distributed between 100 and 300 ms. The second component consists of a data sensor (simulated by a producer) that periodically (every 30ms) sends an update to a remote monitor (simulated by a consumer). The two components serve as an interactive application and the background application, respectively. The components enable us to study the performance of interactive applications in the presence and absence of background load (by simply running the console component with and without the data sensor component).

The QLinux version of the application consists of two client-server pairs, one for each component, that communicate using TCP. As explained above, the client emulating the command console sends 1 byte commands at random intervals and receives a 1 byte response; the response time and the jitter are measured for each command. In the data sensor application, the server receives periodic 1 byte updates from the client; the arrival times of these updates and the jitter are recorded. The TAO version of the command console uses RPCs to issue commands, while the data sensing component is implemented using TAO's real-time event channel. Like in our previous experiment, the mechanisms supported by the H-SFQ packet scheduler and the real-time scheduler in the event channel are used to specify the desired QoS requirements in the two platforms.[3] As an aside, observe that these two applications have characteristics similar to web and video servers—the command console is a request-response application, while the sensor streams data to the monitor, albeit at lower data rates.

Table 1 depicts the response times and the jitter seen by the command console application with and without the data sensor application. The table shows that the response time degrades in the presence of the data sensor application for both platforms (due the increase in system load). The table also shows that QLinux yields smaller response times and

[3] An alternative to the real-time event service is to use the ORB's real-time CORBA features.



(a) Deadline violations (b) Frame arrival times (QLinux) (c) Frame arrival time (TAO)

Figure 5. Performance of video streaming.

Table 1. Response times and jitter for the command console application.

	QLinux (μs)	TAO (μs)
Without data sensor	67617.6 ± 12.97	67974.3 ± 96.09
With data sensor	67853.9 ± 11.53	70960.7 ± 443.9

Table 2. Inter-arrival times of updates and jitter for the data sensor.

	QLinux (μs)	TAO (μs)
Sender	29997.65 ± 4.63	29996.37 ± 9.21
Receiver	29997.63 ± 5.70	29992.54 ± 7.80

smaller jitter than TAO (the response times are 0.5% and 4.4% smaller in QLinux). Further, the increase in jitter due to background load is larger in TAO than QLinux, indicating that QLinux is able to better isolate applications from one another.

Table 2 depicts the performance of the data sensor application as measured by the inter-arrival times of updates at the server and the resulting jitter. As shown, the performance of the two platforms is comparable, with updates arriving every 30ms, indicating that both platforms are able to provide the desired service quality to this application.

The above results show that the effectiveness of middleware mechanisms is only marginally worse than OS kernel mechanisms, indicating that they may be acceptable for applications such as those considered in this experiment.

3.4 Effect of Running Legacy Applications

In this experiment, we examine the efficacy of the two platforms in isolating applications from one another. We consider a worst-case scenario where a legacy application is run concurrently with an application that needs QoS guarantees. We use an unmodified Apache web server as an example of a legacy application and use the streaming server described in Section 3.2 as an example of a multimedia application. The streaming server is configured using the using the audio-video streaming service discussed in Section 3.2. We increase the load on the Apache web server by increasing the number of concurrent requests and measure its impact on the streaming server (i.e., on the transmission of 30 frames in each round). Figure 6 depicts our results. The figure shows that increasing the load on the Apache web server degrades streaming performance by less than 30% in QLinux. In contrast, the web workload interferes with the streaming server on TAO, resulting in a significant degradation in performance. This is because the legacy Apache server bypasses the TAO middleware and interacts directly with the underlying operating system. Since TAO has no control over the behavior of Apache, it is unable to isolate the streaming server from the web requests. In contrast, by managing resources at the OS level, QLinux is able to provide better application isolation (even though Apache does not use any QoS mechanisms, network packets from *all* applications are processed by the QLinux packet scheduler, allowing it to reduce interference from legacy applications).

4 Experiences and Lessons Learned

In this section, we discuss our experiences in using the two platforms and present some of the lessons learned from our study.

- *Effectiveness of resource management mechanisms:* Although the run-time overheads of a middleware de-

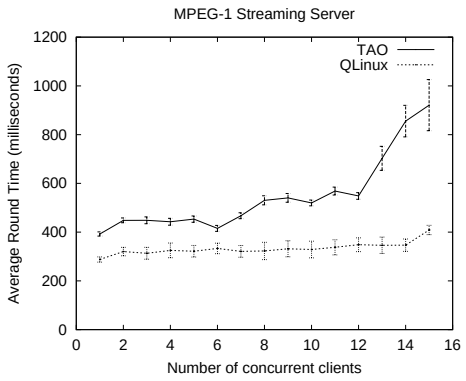


Figure 6. Effect of the Apache web server on a streaming media server.

grade application performance, we found that a middleware can indeed provide performance comparable to an OS-based approach for certain applications and in certain operating regions. Despite the overheads of a middleware system, our study also showed that middleware applications benefit from the optimizations and tuning in the middleware libraries and services, which can result in performance comparable to highly-tuned applications. In contrast, an OS-based approach requires application developers to write code dealing with the low-level OS interactions. Consequently, the efficiency of the application depends on the expertise of the programmer and the efforts employed in tuning the code.

- *Choice of resource management mechanisms:* The choice of the exact resource management mechanism is important, since it can greatly simplify or complicate application development. Consider the streaming media server experiment described in Section 3.2. In this case, QLinux mechanisms only support bandwidth allocation and do not support the notion of deadlines; consequently, the OS supports only throughput guarantees and not frame-specific deadline guarantees.

- *Complexity for application developers:* Designing a middleware application involves a substantial "learning curve". However, the subsequent programmer productivity is also higher due to the higher-level primitives supported by the middleware. Initially, we had to invest a significant effort in learning about TAO's API and functionality. Having done so, we found that TAO's high-level abstractions and services (e.g., event service, naming service) made the task of designing a distributed application easier. In contrast, designing QLinux applications required us to use lower-level primitives supported by the OS system call interface and libraries (which meant writing additional code).

- *System complexity:* Building (compilation) the TAO system and TAO-based applications was found to be surprisingly CPU and memory-intensive. A minimal build of TAO And TAO ORB services, around a million lines of C++ code, took several hours on a lightly loaded Pentium-III with 192MB RAM. Even the simplest TAO application took several minutes to build. In contrast, a build of the QLinux kernel, a few hundred thousand lines of C code, takes about 10 minutes, while compiling a simple application takes tens of seconds. Some of these overheads are an artifact of using C++ for TAO and C for QLinux—C++ compilers are generally slower than those for C. Further, portions of TAO depend on the ACE framework, which further increases its compile-time complexity. While a large compilation complexity may not necessarily translate to run-time complexity, we were nevertheless surprised by these differences.

5 Related Work

Several research efforts have investigated approaches for predictable allocation of system resources such as CPU, disk and network interface bandwidth. Recently, many researchers have devised middleware-based approaches that extend OS functionality in non-trivial ways—these efforts assume that kernel source code access is unavailable due to the proprietary nature of many operating systems, and consequently, enhancing OS functionality without modifying the kernel is the only feasible approach. For instance, scheduling of threads in a threads library is the most prevalent example of managing CPU resources at the user-level.The advantages and disadvantages of the approach are well known—since the kernel is unaware of the presence of user-level threads, the efficacy of the thread scheduler can be diminished by kernel scheduling decisions. The advantage though is that any scheduling policy can be implemented in the threads library. More recently, scheduling tasks at the user-level using a middleware has been studied in [8]; the approach exploits kernel scheduling policies to implement various policies at the user level. Similarly, (soft) real-time OS enhancements to the CPU scheduler have been implemented in numerous operating systems such as Linux [18], FreeBSD [1], Solaris [17] and Windows [16].

Middleware approaches to manage network interface bandwidth include MidART [14], CREMES [2], and TAO [13]. All of these approaches run on commodity operating systems such as Windows and Linux and provide QoS guarantees for inter-process communication. OS enhancements for managing network interface bandwidth includes a number of predictable packet scheduling algorithms;these schedulers provide bandwidth and/or delay guarantees to network flows. Many of these scheduling algorithms have also been implemented in commercial and open-source operating systems. Streaming media servers implemented at the user level

are an example of managing disk resources at the application level [20], while file systems such as SGI XFS [6] and IBM TigerShark [5] implement schedulers that support guaranteed rate I/O.

6 Concluding Remarks

In this paper, we examined two architectural alternatives, namely native OS support and a middleware, for supporting multimedia applications. Specifically, we examined whether extensions to OS functionality are necessary for supporting multimedia applications, or whether much of these benefits can be accrued by implementing resource management mechanisms at the user-level. Our results showed that although the run-time overheads of a middleware can impact application performance, user-level resource management can, nevertheless, be just as effective as native OS mechanisms for certain applications. We also found that kernel-based mechanisms can be more effective at providing application isolation than a middleware system.

We emphasize here that our study represents a first step in the debate between extending OS functionality versus the use of a middleware—our study has not answered all the questions that arise in this debate but has, nevertheless, provided valuable insights into the tradeoffs of the two approaches. Further, our results are specific to the two platforms we examined; while our quantitative results might not generalize to other platforms, many of our observations may apply in qualitative terms. One limitation of our study was that we choose readily available "off-the-shelf" systems for our experimental evaluation (since we believed they were representative examples of the two approaches). An artifact of this design decision was that some of our results were colored by the idiosyncrasies of the two systems. As part of future work, we plan to implement identical resource management mechanisms into a commodity OS kernel and a lightweight middleware and then compare the effectiveness of the two systems.

References

[1] J. Blanquer, J. Bruno, M. McShea, B. Ozden, A. Silberschatz, and A. Singh. Resource Management for QoS in Eclipse/BSD. In *Proceedings of the FreeBSD'99 Conference, Berkeley, CA*, October 1999.

[2] S Chung, O. Gonzalez, K Ramamritham, and C. Shen. CReMeS: A CORBA Compliant Reflective Memory based Real-time Communication Service. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 47–56, December 2000.

[3] Ensim ServerXchange Architecture. Ensim, Inc., http://www.ensim.com/solutions/sxc-arch.shtml, 2000.

[4] T. Harrison, D Levine, and D. Schmidt. The Design and Performance of a Real-time CORBA Event Service. In *Proceedings of OOPSLA '97, Atlanta, GA*, October 1997.

[5] R. Haskin. Tiger Shark–A Scalable File System for Multimedia. *IBM Journal of Research and Development*, 42(2):185–197, March 1998.

[6] M. Holton and R. Das. XFS: A Next Generation Journalled 64-bit File System with Guaranteed Rate I/O. Technical report, Silicon Graphics, Inc, Available online as http://www.sgi.com/Technology/xfs-whitepaper.html, 1996.

[7] D. Levine, S Flores-Gaitan, and D. Schmidt. Empirical Evaluation of OS Endsystem Support for Real-time CORBA Object Request Brokers. In *Proceedings on Multimedia Computing and Networking Conference*, pages 113–129, January 2000.

[8] C. Lin, H Chu, and K. Nahrstedt. A Soft Real-time Scheduling Server on the Windows NT. In *Proceedings of the 2nd USENIX Windows NT Symposium, Seattle, WA*, August 1998.

[9] S. Mungee, N. Sunderan, and D. Schmidt. The Design and Performance of a CORBA Audio/Video Streaming Service. In *Proceedings of the 32st Hawaii International Conference on System Systems (HICSS), Hawaii*, January 1999.

[10] K. Ramamritham, C. Shen, O. Gonzalez, S. Sen, and S. Shirgurkar. Using Windows NT for Real-Time Applications: Experimental Observations and Recommendations. In *Proceedings of the Fourth IEEE Real-Time Technology and Applications, Denver, CO*, June 1998.

[11] Scott Rhine. Loadable Scheduler Modules on Linux. HP Labs, http://resourcemanagement.unixsolutions.hp.com/-WaRM/docs/loadable_sched.html, September 2000.

[12] T. Roscoe and B. Lyles. Distributing Computing without DPEs: Design Considerations for Public Computing Platforms. In *Proceedings of the 9th ACM SIGOPS European Workshop, Kolding, Denmark*, September 2000.

[13] D. Schmidt, D. Levine, and S. Mungee. The Design and Performance of Real-time Object Request Brokers. *Computer Communications*, 21(4):294–324, April 1998.

[14] C. Shen, O. Gonzalez, K. Ramamritham, and I. Mizunuma. User Level Scheduling of Communicating Real-Time Tasks. In *Proceedings of the Fifth IEEE Real-Time Technology and Applications, Vancouver, Canada*, June 1999.

[15] P Shenoy, S. Hasan, P. Kulkarni, and K Ramamritham. Middleware versus Native OS Support: Architectural Considerations for Supporting Multimedia Applications. Technical Report TR01-54, Department of Computer Science, University of Massachusetts, October 2001.

[16] D. Solomon and M. Russinovich. *Inside Windows 2000, 3rd Ed.* Microsoft Press, 2000.

[17] Solaris Resource Manager 1.0: Controlling System Resources Effectively. Sun Microsystems, Inc., http://www.sun.com/software/white-papers/wp-srm/, 1998.

[18] V Sundaram, A. Chandra, P. Goyal, P. Shenoy, J Sahni, and H Vin. Application Performance in the QLinux Multimedia Operating System. In *Proceedings of the Eighth ACM Conference on Multimedia, Los Angeles, CA*, November 2000.

[19] U Vahalia. *UNIX Internals: The New Frontiers*. Prentice Hall, 1996.

[20] M. Vernick, C. Venkatramini, and T. Chiueh. Adventures in Building the Stony Brook Video Server. In *Proceedings of ACM Multimedia'96*, 1996.