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,hasan,purukulk,krithi}@cs.umass.edu

Abstract

In this paper, we examine two architectural alternatives—native OS support versus a 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. 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 by 5-400%, middleware resource management mechanisms are, nevertheless, just as effective as native OS mechanisms for many applications. For certain data- and compute-intensive applications, however, we find a middleware to be a less effective choice than a OS-based approach. Finally, we find OS kernel-based mechanisms to be more effective at providing application isolation and at preventing applications from interfering with one another.

Keywords: distributed multimedia systems, middleware, operating systems

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

*This research was supported in part by a NSF Career award CCR-9984030, NSF grants CCR-0098060, EIA-0080119, Intel, IBM, Sprint, and the University of Massachusetts.

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

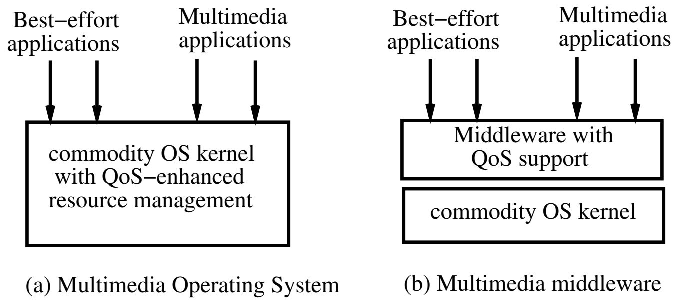


Figure 1: Two canonical approaches for supporting multimedia applications. A multimedia operating system employs enhanced resource management mechanisms within the kernel, whereas a multimedia middleware employs QoS mechanisms at the user-level to support soft real-time multimedia applications. Both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

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 such as word processors and compilers, 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 [24], the real-time scheduling class in Solaris [27] and the reservation and proportional-share schedulers developed for Linux [26] and FreeBSD [2].

- *Use of a middleware:* 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 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 can provide other useful services, such as naming, not provided by the operating system. Examples of this approach include real-time CORBA (TAO) [16, 20] and MidArt [17, 21].

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 a middleware is appealing since no modifications to the OS kernel are necessary, making the approach feasible on any COTS system. 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.

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.

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-based approach yield performance that is comparable to an OS-based approach despite its run-time overheads?

To address these questions, we carried out a detailed experimental evaluation of the two approaches. We use real-time CORBA (TAO) as a representative example of a multimedia middleware and QLinux as a representative example of a multimedia operating system. Our experimental evaluation of these platforms uses a mix of best-effort and multimedia applications ranging from web servers, streaming media servers, audio file download applications, and industrial control applications. Our results show that the run-time overheads of TAO impact application performance by 5% to 400% when compared to QLinux. Specifically, we see the TAO yields (i) file download times that are 8% slower, (ii) HTTP response times that are comparable at low loads and marginally worse at heavy loads, (iii) response times that are 18% larger when servicing dynamic HTTP requests, (iv) streaming performance that is comparable at low loads and 10% worse at heavy loads, and (v) comparable performance for industrial control applications. Surprisingly, we find that untuned applications sometimes yield *better* performance on TAO than on QLinux. A closer examination revealed that this was 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, our experiments indicate that these benefits are outweighed by the higher run-time overheads of a middleware at moderate to heavy loads, resulting in a degradation in application performance. Our experiments also indicate that, for certain applications such as industrial control systems, middleware resource management mechanisms are just as effective as native OS mechanisms. For data- and compute-intensive applications, however, the run-time overheads of middleware make it less effective than a QoS-enhanced kernel. Finally, we also find that a middleware is less effective at providing application isolation (i.e., preventing interference when multiple competing applications run concurrently).

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. Many of the differences between the two approaches are well-known, although not necessarily quantified; however, we repeat them for the benefit of the reader.

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. Consequently, such an approach has low overheads. Furthermore, the benefits of providing service differentiation to applications typically outweigh the run-time overheads imposed by the approach [22].

- *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 a familiar API, resulting in low complexity for the application developer (while not easily quantifiable, the ability to program to a familiar, well-known API is usually considered to be an advantage, especially since a new API typically requires a substantial learning effort). 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]

- *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 (e.g., the BeOS multimedia kernel [1]) or make radical changes to an existing OS kernel. Whereas such an approach eliminates the hurdles faced in incrementally extending OS functionality, a new or radically redesigned kernel can result in compatibility problems with existing applications.

2.2 Multimedia Middleware

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

[1] To illustrate, Ensim ServerXchange, a commercial product, employs this approach—OS functionality is extended via dynamically loadable modules that provide QoS support [8]. HP's Linux CPU scheduler interface also embraces this philosophy by allowing new CPU schedulers to be written as dynamically loadable modules [18].

[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.

- *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 [19]. Consequently, applications 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 and run legacy applications directly on the OS. However, this can interfere with the QoS guarantees provided by the middleware 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 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, designing a "heavyweight" middleware system such as real-time CORBA can have complexity comparable to or greater than OS kernels.[3]

- *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 a middleware—in principle, a middleware 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. 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

[3]As anecdotal evidence, consider the QLinux kernel versus the TAO real-time CORBA middleware. Compiling the QLinux kernel takes a few minutes on our machines, while compiling RT-CORBA takes several hours, indicating that the latter is significantly more complex software system than the former.

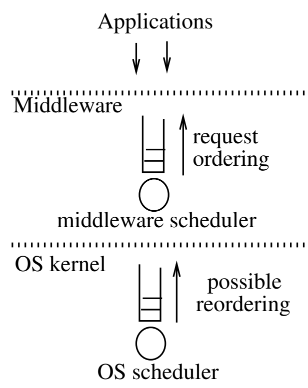


Figure 2: Possible interactions between the middleware scheduler and the OS scheduler. The two effectively form a two-level scheduler.

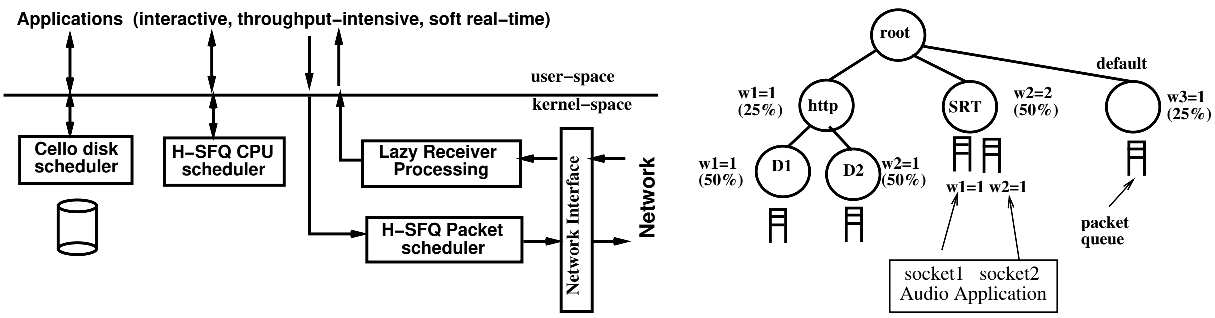
request). Whereas this approach suffices for certain OS resources, it can reduce the throughput and utilization of resources where concurrency is important. For instance, in the case of disks, the presence of multiple outstanding requests allows the disk scheduler to perform seek optimizations and reduce 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 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 the middleware 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. 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 [3]. Hence, a performance study of network interface bandwidth management using the two approaches provides a good overview of their tradeoffs. Further, there are publicly-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 experimental study.



(a) QLinux architecture

(b) H-SFQ packet scheduler

Figure 3: The QLinux architecture and the H-SFQ packet scheduler. 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.

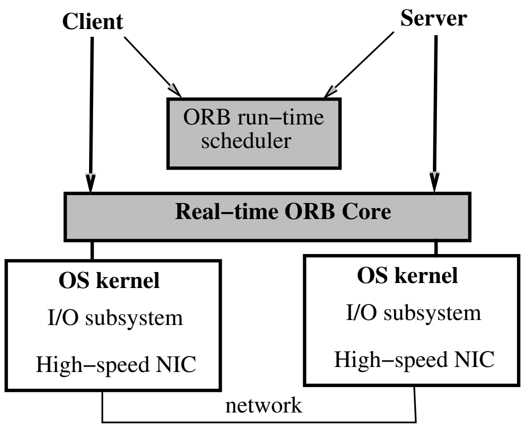
2.3.1 QLinux

QLinux is a QoS-enhanced kernel based on the popular Linux operating system [26]. 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 [26]. Figure 3(a) illustrates these components. 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.

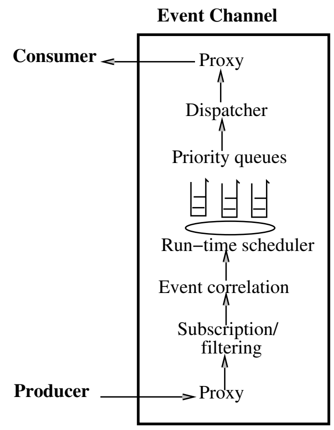
An OS kernel employs a packet scheduler at each network interface to determine the order in which outgoing packets are transmitted. Traditionally, operating systems have employed the FIFO scheduler to schedule outgoing packets. To better meet the needs of multimedia applications, QLinux employs H-SFQ 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(b)). 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.

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



(a) TAO architecture



(b) The TAO event channel

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

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 [20]. TAO provides these features and optimizations while conforming to the CORBA 2.0 standard. Moreover, TAO is backward compatible with conventional CORBA and hence is able to support all existing (best-effort) CORBA applications.

The key feature of TAO relevant to our study is the CORBA event 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.

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. We use QLinux and TAO as representative examples of a multimedia operating system and a multimedia middleware, respectively. Using these platforms, we attempt to answer the following question: how effective are middleware-based resource management mechanisms when compared to OS mechanisms? More specifically, (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?

In what follows, we first 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 Pentium III workstations 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.

The workload for our experiments consists of the following applications: (i) *null:* an application that measures the baseline overhead of network communication in the two platforms, (ii) *HTTP:* a multi-threaded web server that services HTTP requests, (iii) *download:* a simple file download application that mimics download of audio files, (iv) *streaming:* a streaming video server and client, (v) *industrial control:* a real-time application that emulates an industrial control console system and a data sensor and monitoring system.

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 Communication overheads of an Echo/Null RPC application

We first measure the baseline overhead of network communication. To do so, we design an application that measures the overhead of a single-unit of client-server communication in the two platforms. In the QLinux version of the application, the client sends a null string to the application over a TCP socket. The server responds by sending this string back to the client; a thread per request model is used to service client requests. The TAO version of the application uses a distributed Echo object. The server instantiates an instance of this object and the client communicates with the server by invoking a remote method on the object. Specifically, the client invokes a null (empty) method to mimic the sending of a null string, and the server responds by sending a null response. The underlying network communication in TAO uses remote procedure calls (RPC). The actual details of setting up the RPC and the network communication are handled by the the TAO object request broker (ORB) and are completely transparent to the application. Like the QLinux application, the TAO server is configured to employ a thread-per-request model.

Table 1: Response time of null communication with 95% confidence intervals.

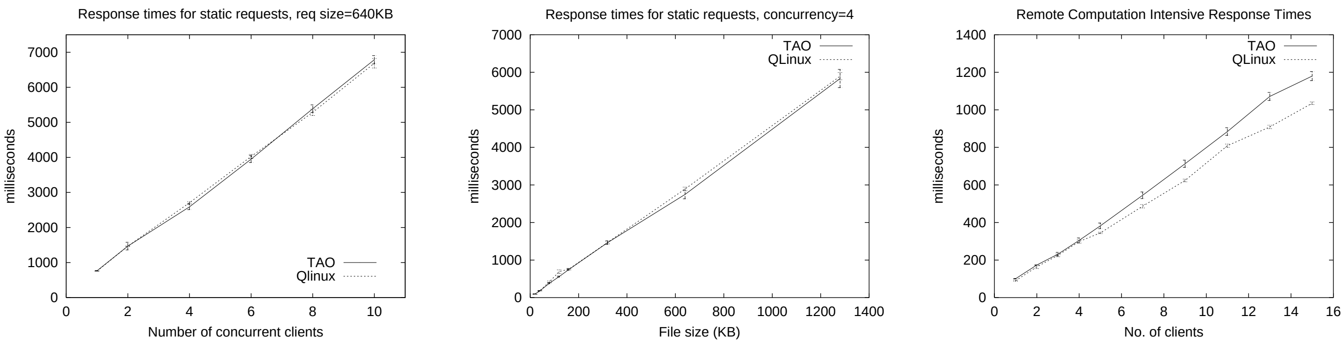
Server	QLinux (μs)	TAO (μs)
Local	1277.2 ± 49.5	4805.6 ± 398
Remote	1766.2 ± 364.7	5435.7 ± 1180.5

We ran both instances of the application, first with the client and server on the same machine and then on different machines. In each case, we measured the end-to-end response time for a large number of runs. Table 1 reports our results with 95% confidence intervals. Not surprisingly, our results show a higher overhead for TAO as compared to QLinux. This is because the RPC communication in TAO involves overheads such as marshalling and demarshalling of arguments, header processing, etc., that are not incurred in vanilla socket communication. As shown in the table, TAO overheads are a factor of 3.8 and 3.1 larger for local and remote communication, respectively, when compared to QLinux.

3.3 Effect of Concurrent HTTP Requests

Next, we compare the efficacy of QLinux and TAO in servicing web requests. Our application consists of a multi-threaded web server that services multiple web clients. Upon receiving each HTTP request, the server spawns a new thread that parses the request, retrieves the requested data, performs any computations if necessary (e.g., for cgi-bin requests) and sends back a HTTP response. Both the QLinux and TAO versions of the application employ a thread-per-request model to service requests. We populate the server with file sizes ranging from 2KB to 1.3MB and measure the response times for different request sizes.

We first study the performance for static web page requests. To do so, we vary the request size and the degree of concurrency (i.e., the number of simultaneous client requests) and measure their impact on client response times. Figure 5 depicts our results. As shown in Figure 5(a), the response times increases with increasing degree of concurrency (i.e., load) for both platforms. Surprisingly, TAO yields response times comparable to QLinux at low loads (the response times are even marginally better in some cases, although not definitively better due to the overlap in the 95% confidence intervals). An increase in load, however, causes an increase in the middleware overheads, resulting in marginally worse response times than the QLinux server. To further understand why TAO can yield comparable performance at low loads despite the middleware overheads, we profiled the two applications using the `strace` utility in Linux (`strace` prints all system calls invoked by an executing program). We found that the communication routines in TAO are highly optimized and employ efficient mechanisms such as `select` that are typically used for high-performance asynchronous network I/O. The QLinux server, on the other hand, uses the "standard" synchronous (blocking) mechanisms for socket communication (which are easier to use but less efficient than asynchronous I/O). To understand why the TAO server yields comparable performance at low loads but worse performance at high loads, consider the two key overheads involved in network communication: (i) message setup overhead and (ii) send and receive overheads (other components such as transmission and propagation delays are identical in the two cases). As explained in Section 3.2, the RPC mechanisms in TAO impose a higher message



(a) Effect of concurrency (load) (b) Impact of request size (c) Impact of dynamic requests

Figure 5: Response time of web requests in the two platforms.

setup and processing overhead than vanilla sockets. In contrast, the asynchronous I/O mechanisms employed by TAO results in more efficient data reception. The overall response time depends on which of the two components dominates. At low loads and for large data transfers, the benefits of the latter appear to outweigh the higher message setup overheads, resulting in better performance. With increase in load, however, the CPU overheads of the former component appear to outweigh the efficiencies in network communication (causing TAO to perform worse than QLinux). This is also evident from Figures 5(a) and (b). The figures show that the response time of the TAO server improves with increasing request sizes but degrades with increasing concurrency.

At a first glance, it might appear that the above comparison is not a “fair” one, since the two applications employ different mechanisms, namely synchronous and asynchronous I/O, for network communication. However, we note that the two applications are representative of what a knowledgeable programmer would have produced as an initial, untuned version. In fact, like any programmer, we used sample code and examples from standard reference texts to guide our application design. For instance, the client-server TCP communication code in Linux web server is based on examples from a widely-used reference on TCP/IP, while the code for the TAO server is based on examples provided in the TAO documentation [16]. Since the inter-process communication in TAO is transparently handled by the middleware, application programmers benefit from the highly optimized libraries written by TAO developers. In case the QLinux, the socket communication code is written by the application programmer and the efficiency of this code hinges on the expertise of the programmer (e.g., in using the more efficient features such as asynchronous I/O).

Next, we compare the performance of the two platforms in servicing dynamic HTTP requests. Each such request requires the server to perform some computation on the retrieved data before responding to the client. Like our previous experiment, we increase the degree of concurrency and measure its impact on the client response times. Figure 5(c) plots our results. Our experiments indicate that the CPU rather than the network is the bottleneck for dynamic web requests (prior studies have also observed a similar behavior [13]). In addition to request-specific computations, the TAO middleware imposes additional CPU overheads due to the RPC message setup processing. This further increases the demand on the bottleneck resource, resulting in worse response times then the QLinux server. The difference in the response time increases with load and is around 18% at a concurrency of 15 (see Figure 5(c)).

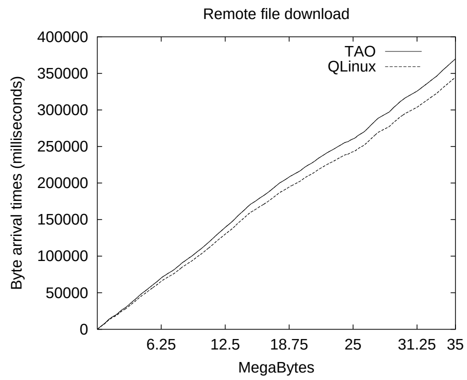
3.4 Performance of a File Download Application

File download applications such as ftp have been in use for many decades. More recently, a new generation of file download applications based on peer-to-peer file sharing have become popular. Such applications are typically used to share and download audio and video files. In this experiment, we examine the performance of such an application built using QLinux and TAO (our application only emulates the file download aspect and does not implement the search phase that precedes the file download in peer-to-peer applications). The QLinux version of our application involves a server that, upon receiving a request, sends the requested file over a TCP connection (the sending rate for large file is limited only by TCP's congestion control mechanism). Unlike the TAO web server, the TAO version of the file download application does not use synchronous RPCs for communication (since RPCs are not suitable for large file downloads). Instead, the application employs the TAO event service. The event service acts as a communication channel between the client and the server—the server sends blocks of the file as events to the event channel, which are then delivered to the requesting client. To permit a fair comparison, the amount of data sent in each event in TAO is identical to that sent in each socket call in the QLinux version. Since the TAO server could be servicing multiple file download requests concurrently via the event channel mechanism, each client employs event filters at the event channel to selectively receive data blocks belonging to the requested file. TAO allows the event channel service to be run on a different machine from the client and the server; however, we co-locate the server application and the event channel service on the same machine for a better comparison.

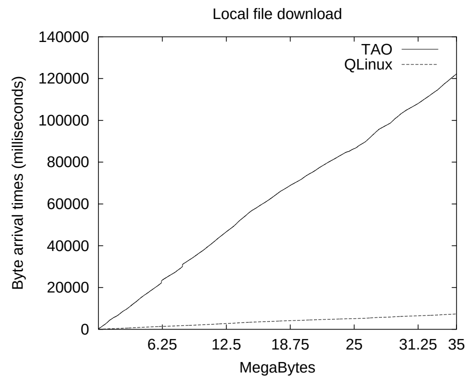
We use throughput as the metric to compare the performance of file downloads on the two platforms. To do so, we measure the total time to download a 35MB file from the server. Figure 6 plots a time series of the amount of data received by the client over time. As shown in Figure 6(a), the QLinux version of the application yields 8% faster downloads as compared to the TAO version. TAO's worse performance is primarily due to the overheads imposed by the event service—the event channel adds one level of indirection to data transfers, since the server must first send data to the event channel, which then sends it to the client. To further isolate the overhead of using the event service, we measured the overhead of sending 1KB of data over the event channel and compared it to the overhead of sending data over vanilla sockets. To completely isolate this overhead, we eliminated other network overheads such as transmission and propagation delays by running the client, the server and the event service on the same machine. Table 2 depicts our results. The table shows that, in the absence of network transmission and propagation delays, sending data over an event channel incurs a factor of 2.45 times larger overhead than vanilla sockets. Figure 6(b) further illustrates the impact of this overhead by plotting the download time of a 35MB file from a local server. The figure shows that the event channel becomes the bottleneck, causing the throughput of local downloads to be significantly worse than socket-based downloads. In contrast, the network, rather than the event channel, is the bottleneck in remote file downloads, resulting in a smaller degradation in throughput.

3.5 Performance of Video Streaming

Next, 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



(a) Remote file download



(b) Local file download

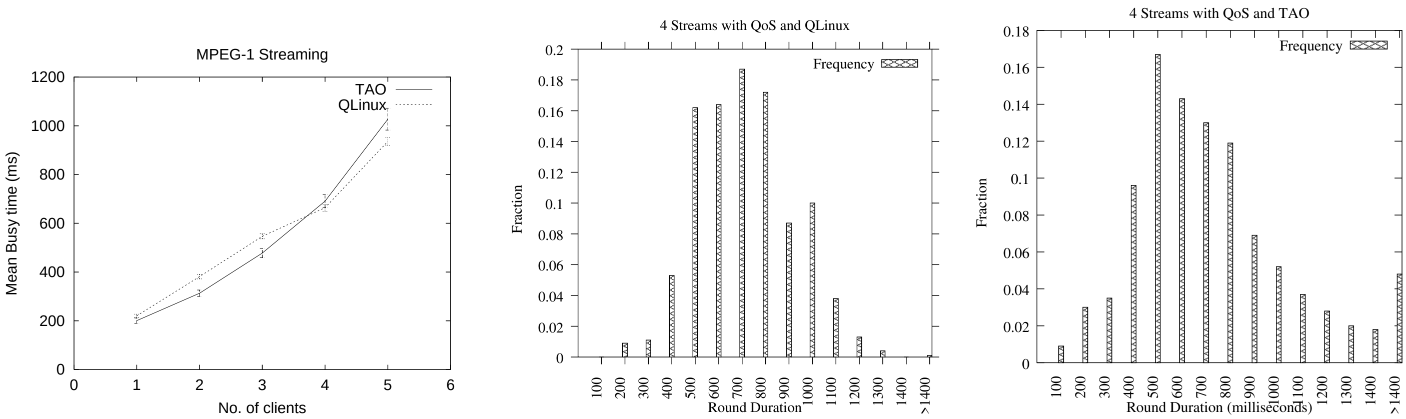
Figure 6: Performance of file downloads.

Table 2: End-to-end delay for sending 1KB data on a local machine (with 95% confidence intervals).

Platform	Delay (μs)
TAO Event Channel	3134.22 $\pm$ 17.1
QLinux sockets	1279.3 $\pm$ 203.3

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 (thus, the transmission deadline for frames is 1 sec). 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 real-time event service and the scheduling service for real-time streaming. 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. To do so, the server creates an event set for each client consisting of 30 frames and pushes them to the client via the event channel. The deadline on each event is set to the end of the round (this is done by specifying the worst case execution time and the period of the task, both of the which are set to the round duration). The real-time scheduler then uses these QoS specifications to schedule and deliver events (frames) by their deadlines.

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 7 plots our results. Figure 7(a) shows that both QLinux and TAO are able to meet the real-time requirements imposed by streaming (by virtue for delivering frames before the end of each round). The figure also shows that TAO outperforms QLinux at low loads, while the opposite is true at moderate to heavy loads. We conjecture that this behavior is due to the idiosyncrasies of specific schedulers employed by the two platforms. Recall that the real-time scheduler in TAO allows deadlines to be associated with event sets, while QLinux employs a fair share scheduler that only supports bandwidth allocation to streams and does not support the notion of deadlines. Consequently, the TAO scheduler can provide better QoS at low loads. Increasing the number of



(a) Busy time per round (b) Frame arrival times (QLinux) (c) Frame arrival time (TAO)

Figure 7: Performance of video streaming.

concurrent clients imposes additional CPU overheads on the event service in TAO—since multiple streams share the event channel, the event service needs to process each event using client-instantiated filters in order to deliver it to the appropriate client. The increased computational overhead outweighs the benefits of a better scheduler, resulting in worse performance than the QLinux server.

Fig 7(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. Less than 6% of the rounds have frames missing their deadlines in QLinux, where there are deadlines misses in 14% of the rounds in TAO.

As a final caveat, observe that the event service employs TCP as its underlying transport protocol. It is well known that TCP is a less suitable transport protocol for streaming data, and UDP is the protocol of choice for streaming. TAO does not currently support UDP-based event delivery, necessitating the use of TCP (the TAO documentation indicates that a different transport protocol can be employed for TAO services; however, this requires the application programmer to first implement such a protocol). No such restrictions exist when using vanilla sockets, since the OS supports both TCP and UDP. To permit a comparison between the two platforms, the QLinux server uses TCP for streaming data despite the availability of UDP. Our experiments with UDP (not reported here) have indicated that the performance of the QLinux server improves further when UDP is used.

3.6 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 [17] 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

Table 3: Response times and jitter for the command console application.

	QLinux (μs)	TAO (μs)
Without data sensor	67617.6 $\pm$ 12.97	67974.3 $\pm$ 96.09
With data sensor	67853.9 $\pm$ 11.53	70960.69 $\pm$ 443.86

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. 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 3 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 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 4 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.

3.7 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

Table 4: Inter-arrival times of updates and jitter for the data sensor.

	QLinux (μs)	TAO (μs)
Sender	29997.65 $\pm$ 4.63	29996.37 $\pm$ 9.21
Receiver	29997.63 $\pm$ 5.70	29992.54 $\pm$ 7.80

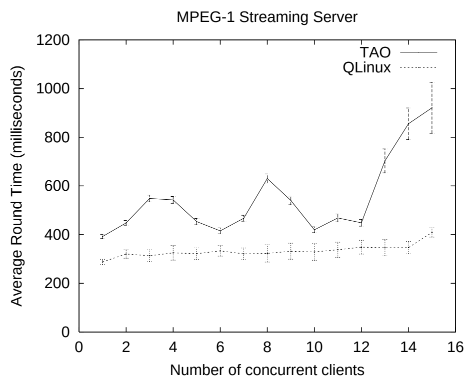


Figure 8: Effect of the Apache web server on a streaming media server.

guarantees. We use an unmodified Apache web server as an example of a legacy application and use the streaming server described in Section 3.5 as an example of a multimedia application. The streaming server is configured with the same QoS parameters discussed in Section 3.5, whereas the Apache web server does not use any QoS mechanisms. 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 8 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 erratic behavior and 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 degrade 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. To illustrate, many of our experiments clearly demonstrated that the overheads of a middleware can degrade application performance from 5% to 400%, depending on the scenario. However, 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 3.5. For this application, the TAO mechanisms were advantageous since they allowed deadlines to be associated with video frames. In contrast, QLinux mechanisms only support bandwidth allocation and do not support the notion of deadlines, making them a less desirable choice for such an application.

- *Functionality:* The middleware can constrain the application developer to the functionality supported by the platform, even when the underlying OS supports additional functionality. Our experience with the video streaming application indicated that the streaming server was constrained to using TCP, since UDP-based event delivery was not supported by the middleware (even though the underlying OS supported both transport protocols). It should be noted that TAO allows new protocols to be “plugged” into the architecture; however the complexity of implementing a new protocol is substantial. As an aside, we also explored the possibility of using the audio-video service supported by TAO; however, this service only provides support for signaling and does not provide any functionality for streaming.

- *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). The advantage though was that we could program using an already familiar OS interface.

- *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 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 takes about 10 minutes, while compiling a simple application takes only a few seconds. Clearly, these overheads are an artifact of choosing a heavy-weight middleware platform such as CORBA for our study—a lightweight middleware designed specifically for resource management may be significantly easier to use. Nevertheless, we are surprised by these overheads since TAO is claimed to be a highly efficient and optimized system.

5 Related Work

Several research efforts have investigated the two approaches for managing system resources such as CPU, disk and network interface bandwidth. Much of the research on enhancing OS functionality has been carried out in the multimedia and the real-time systems community. The research on middleware has had two distinct foci. Some researchers have focused on middleware 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. The other avenue of research in the area is on the design of middleware systems such as CORBA [5] and DCE [6] that provide a platform for running distributed applications. In the rest of this section, we provide a brief overview of these efforts from the perspective of three resources, namely CPU, network interface and disk.

Scheduling of threads in a threads library is the most prevalent example of managing CPU resources at the user-level [14]. 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 [15]; the approach exploits kernel scheduling policies to implement various policies at the user level. OS enhancements to the CPU scheduler have been implemented in numerous operating systems such as Linux, FreeBSD, Solaris and Windows. Whereas Linux [26], FreeBSD [2] and Solaris [25] implement a fair, hierarchical, proportional-share scheduler to support multimedia applications, the Windows operating system does so using real-time priority class [24].

Middleware approaches to manage network interface bandwidth include MidART [21], CREMES [4], and TAO [16]. 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 [7, 10, 23]; 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 [28], while file systems such as SGI XFS [12] and IBM TigerShark [11] implement schedulers that support guaranteed rate I/O.

As explained in Section 1, despite these numerous research efforts, there has been no systematic study of the tradeoffs of the two approaches; our current work is a step in this direction.

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 in a middleware. To answer these questions, we used QLinux and TAO our experimental platforms and examined their efficacy in supporting distributed applications. Our results showed that although the run-time overheads of a middleware can impact application performance by 5-400%, middleware resource management mechanisms are, nevertheless, just as effective as native OS mechanisms for certain applications. For data- and compute-intensive applications, however, the TAO middleware was found to be less effective than a OS-based approach. Finally, we found kernel-based mechanisms to be more effective at isolating applications from one another (particularly legacy applications from multimedia applications).

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. A particular 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, rather than any fundamental limitations of the two approaches. A better approach might have been to implement identical resource management mechanisms into a commodity OS kernel and a lightweight middleware and then compare the effectiveness of the two systems. Such a study is the subject of ongoing research.

References

[1] BeOS: The Media OS. Be, Inc. Whitepaper, http://www.be.com/products/freebeos/beoswhitepaper.html, 1999.

[2] 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.

[3] M. Bradshaw, B. Wang, S. Sen, L. Gao, J. Kurose, P. Shenoy, and D. Towsley. Periodic Broadcast and Patching Services: Implementation, Measurement and Analysis in an Internet Streaming Media Testbed. In *Proceedings of the Ninth ACM Conference on Multimedia, Ottawa, Canada (to appear)*, October 2001.

[4] 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.

[5] Corba Documentation. Available from http://www.omg.org.

[6] Distributed Computing Environment Documentation. Available from http://www.opengroup.org.

[7] A. Demers, S. Keshav, and S. Shenker. Analysis and Simulation of a Fair Queueing Algorithm. In *Proceedings of ACM SIGCOMM*, pages 1–12, September 1989.

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

[9] C. Gill, T. Harrison, and C O'Ryan. Using the Real-time Event Service. TAO Documentation, http://www.cs.wustl.edu/ schmidt/events_tutorial.html, December 1998.

[10] P. Goyal, H. M. Vin, and H. Cheng. Start-time Fair Queuing: A Scheduling Algorithm for Integrated Services Packet Switching Networks. In *Proceedings of ACM SIGCOMM'96*, pages 157–168, August 1996.

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

[12] 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.

[13] A. Iyengar and J. Challenger. Improving Web Server Performance by Caching Dynamic Data. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems, Montrey, CA*, December 1997.

[14] S. Kleiman, D. Shah, and B. Smaalders. *Programming with Threads*. Prentice Hall, Inc, 1996.

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

[16] Inc. Object Computing. *TAO Developer's Guide, Version 1.0*. Object Computing, Inc., 1999.

[17] 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.

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

[19] 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.

[20] D C. Schmidt. TAO Overview. TAO Documentation, http://www.cs.wustl.edu/ schmidt/TAO-intro.html, January 2001.

[21] 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.

[22] P. Shenoy, P. Goyal, and H M. Vin. Architectural Considerations for Next Generation File Systems. In *Proceedings of the Seventh ACM Multimedia Conference, Orlando, FL*, November 1999.

[23] M. Shreedhar and G. Varghese. Efficient Fair Queuing Using Deficit Round Robin. In *Proceedings of ACM SIGCOMM'95*, pages 231–242, 1995.

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

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

[26] 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.

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

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