A Linux-Based Implementation of a Middleware Model Supporting Time-Triggered Message-Triggered Objects

Stephen F. Jenks, Kane Kim, Emmanuel Henrich, Yuqing Li, Liangchen Zheng
Univ. of California, Irvine
{sjenks, khkim, ehenrich, yuqingl, lzheng}@uci.edu

Moon H. Kim
KonKuk University, Korea
mhkim@konkuk.ac.kr

Hee-Yong Youn
SungKyunKwan Univ., Korea
youn@ece.skku.ac.kr

Kyung Hee Lee, Dong-Myung Seol
ETRI, Korea
{kyunghee, dmsul}@etri.re.kr

Abstract

Programming and composing deterministic distributed real-time systems is becoming increasingly important, yet remains difficult and error-prone. An innovative approach to such systems is the general-form timeliness-guaranteed design paradigm, which is the basis for the Time-triggered Message-triggered Object (TMO) programming and system specification scheme. This approach was originally developed for Windows programming environments and operating systems. This paper describes the techniques needed to make TMO support the Linux operating system and reports the resulting performance characteristics.

1. Introduction

Distributed real-time (RT) applications have become common and essential recently, yet the state of the art in engineering large-scale distributed RT systems remains inadequate. One of the major challenges is the difficulty of programming such applications while maintaining timeliness guarantees.

Since 1992, co-author Kane Kim and his research collaborators have been establishing an RT distributed computing (DC) object programming model. The project started with the skeleton of a concrete syntactic structure and execution semantics of a *high-level* RT DC object named the *Time-triggered Message-triggered Object* (TMO) [1-4]. The TMO programming and specification scheme has been enhanced in several steps along with supporting tools (kernel, middleware, API, specification, etc) since then.

To enable programming and execution of TMOs, a middleware model/architecture called the TMOSM (*TMO Support Middleware*) that provides execution support mechanisms and can be easily adapted to a variety of commercial, industry standard kernel+hardware platforms was established [3, 5].

Prototype implementations of TMOSM were first built on the Windows XP/2000/NT family of (OS) system kernel platforms [5]. TMOES/AnyORB/NT [6] is another prototype implementation realized in the form of a CORBA service that runs on platforms equipped with Windows NT and an ORB (object request broker) and supports CORBA-compliant application TMOs. A Windows CE based prototype of TMOSM has also been developed. To enhance the appeal of TMO to many segments of the RT computing system engineering community, an early effort to support TMOs on Linux platforms was undertaken by several researchers [7]. The current Linux adaptation, described here, is significantly different from that earlier prototype.

This paper describes the techniques used to enhance the portability of the well-established real-time middleware model, TMOSM, as well as the techniques for adapting TMOSM to the Linux OS kernel platform. The paper starts with an overview of TMOSM in Section 2. Section 3 introduces the enhanced TMOSM, which consists of two layers, the TMOSM Main Layer and the Kernel Adaptation Layer, unlike the previous monolithic-structured TMOSM. The two-layer structure eases porting of TMOSM to different OS kernel platforms. Section 4 discusses the boundaries between the two layers that have evolved to keep one common Main Layer in the course of developing the Linux-based prototype. Windows provides a somewhat richer set of kernel services. Specific differences in provided services were encountered in the areas of thread management, memory management, and scheduling. That section outlines how these differences were addressed and what performance impacts they have on the performance of TMOSM which facilitates

the powerful approach to RT object-oriented DC programming.

Section 5 reports the success of this conversion effort and some performance behavior observed from the Linux-based prototype of TMOSM. With the availability of TMOSM on Windows XP, Window CE, and Linux, a capable, cross-platform RT DC programming environment is available to support advanced distributed RT system engineering (http://dream.eng.uci.edu/TMOdownload/).

2. TMOSM description and requirements

In the TMO programming model, RT systems are structured as networks of distributed computing (DC) objects. These DC objects specify their timing requirements through mechanisms which are intuitive and simple to use and yet possess strong expressive power. These timing specifications are interpreted by the DC resource management component of the TMOSM. The TMO timing specification model allows programmers to specify *start-time-windows* and *completion deadlines* of RT computation-segments, rather than simplistically specifying priority values. After all, it is natural for RT application programmers to think about start-time-windows and completion deadlines rather than fragile priority schemes [8]. In addition, programmers specify *deadlines for result arrival* in remote method calls, as depicted in Figure 1. This enables systematic composition of higher-level services with timeliness assurances from lower-level services.

TMO programming does not require a new language or compiler. Instead, an execution engine such as TMOSM provides execution support services which can be invoked directly or indirectly via application programming interfaces (APIs). TMOSM uses well-established services of commodity OS kernels, e.g., process and thread support services, short-term scheduling services, and low-level communication protocols, in a manner transparent to the application programmer [5, 9]. The TMOSM architecture was devised to contribute to simplifying the analysis of the execution time behavior of application TMOs running on TMOSM. As mentioned earlier, TMOSM has been found to be easily adaptable to commercial hardware + kernel platforms, e.g., PCs or similar hardware with Windows XP, Windows CE, or Linux. A prototype implementation on Windows XP/2000/NT, TMOSM/XP, was developed [5]. Our experiences indicate that even this middleware extension of a general-purpose OS (Windows XP) can support application actions with the 10ms-level timing accuracy. TMOES/AnyORB/NT

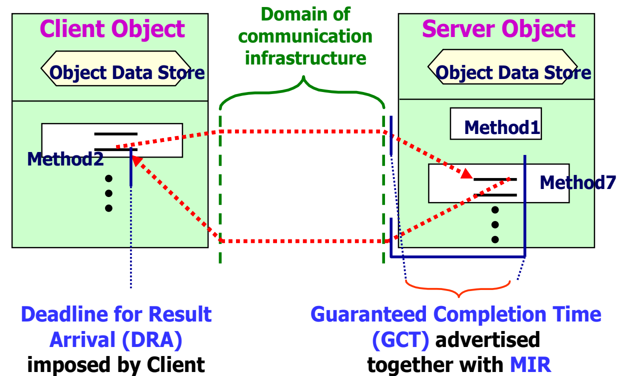

Figure 1. Client deadline vs. server GCT (adapted from [1])

[9] is another prototype implementation realized in the form of a CORBA service that runs on platforms equipped with Windows NT and an ORB (object request broker) and supports CORBA-compliant application TMOs. A Windows CE based prototype of TMOSM has also been developed and under continuous optimization with the goal of supporting application actions with better-than-10ms-level timing accuracy.

A friendly programming interface wrapping the execution support services of TMOSM has also been developed and named the *TMO Support Library* (TMOSL) [1, 5]. It consists of a number of C++ classes and approximates a programming language directly supporting TMO as a basic building block. The programming scheme and supporting tools have been used in a broad range of basic research and application prototyping projects in a number of research organizations and also used in an undergraduate course on RT DC programming at UCI for about three years (http://dream.eng.uci.edu/eecs123/SERIOUS.HTM). TMO facilitates a highly abstract programming style without compromising the degree of control over timing precisions of important actions.

2.1. TMO structure and semantics

TMO is a natural, syntactically minor, and semantically powerful extension of conventional object structure. As depicted in Figure 2, the basic TMO structure consists of four parts:

ODS-sec: Object-data-store section. This section contains the data-container variables shared between methods of a TMO. Variables are grouped into *ODS segments* (ODSSs) which are the units that can be locked for exclusive use by a TMO method in execution. Scheduling and access control protect mutual exclusion.

EAC-sec: Environment access capability section. These "gate objects" provide efficient call-paths to remote object methods, real-time multicast and memory replication channels (RMMCs), and I/O device interfaces.

SpM-sec: Spontaneous method section. These are *time-triggered methods* that become alive at specified times.

SvM-sec: Service method section. These provide service methods which can be called by other TMOs.

Major features are summarized below.

(1) Distributed computing component: The TMO is a distributed computing component and thus TMOs distributed over multiple nodes may interact via remote method calls. To maximize the concurrency in execution of client methods in one node and server methods in the same node or different nodes, client methods are allowed to make non-blocking service requests to service methods. In addition, TMOs can interact by exchange of messages over RMMCs.

(2) Clear separation between two types of methods: The TMO may contain two types of methods, time triggered (TT) methods (*spontaneous methods* or SpMs), which are clearly separated from the conventional *service methods* (SvMs). The SpM executions are triggered when the RT clock reaches time values determined at the design time. On the contrary, SvM executions are triggered by calls from clients that are transmitted by the execution engine in the form of service request messages. Moreover, actions to be taken at real times, which can be determined at the design time, can appear only in SpMs.

Triggering times for SpMs must be fully specified as constants during the design time. Those RT constants as well as related *guaranteed completion times* (GCTs) of the SpM appear in the first clause of an SpM specification called the *autonomous activation condition* (AAC) section. An example of an AAC is "for t = from 10am to 10:50am every 30min start-during (t, t+5min) finish-by t+10min" which has the same effect as {"start-during (10am, 10:05am) finish-by 10:10am", "start-during (10:30am, 10:35am) finish-by 10:40am"}.

(3) *Basic concurrency constraint* (BCC): This rule prevents potential conflicts between SpMs and SvMs and reduces the designer's efforts in guaranteeing timely service capabilities of TMOs. Basically, activation of an SvM triggered by a message from an external client is allowed only when potentially conflicting SpM executions are not active. An SvM is allowed to execute only when an execution time-window big enough for the SvM exists and does not overlap with the execution time-window of any SpM that accesses the same ODSS data as the SvM. However, the BCC does not stand in the way of either concurrent SpM executions or concurrent SvM executions.

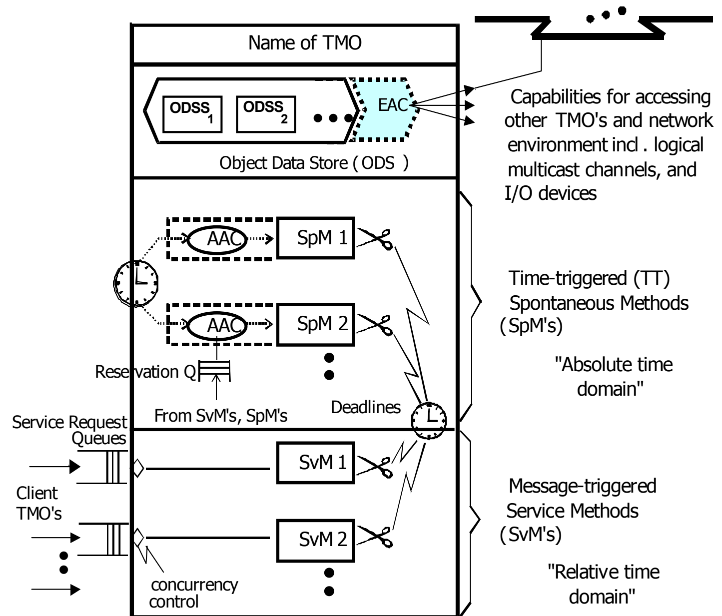

Figure 2. Basic TMO structure (from [2])

(4) *Guaranteed completion time* (GCT) of the server (i.e., an SvM of a server TMO) and the *result return deadline* imposed by the client: The TMO incorporates deadlines in the most general form. Basically, for output actions and method completions of a TMO, the designer guarantees and advertises execution time-windows bounded by start times and completion times. In addition, deadlines can be specified in the client's calls for service methods for the return of the service results.

2.2. TMOSM Architecture

The TMO Support Middleware (TMOSM) consists of a number of virtual machines (VMs), each managing a set of threads and using them to perform certain specialized functions as parts of executing TMOs. See Figure 3. To make VMs co-exist on top of a commodity kernel, TMOSM contains one more component, which can be viewed as the innermost core and is a *super-micro* thread called the WTST (*Watchdog Timer & Scheduler Thread*). It is a "super-thread" in that it runs at the highest possible priority level. It is also a "micro-thread" in that it manages the scheduling / activation of all VMs which in turn operate other threads in TMOSM. Even those threads created by the node OS before TMOSM starts are executed only if WTST allocates some time-slices to them. Therefore,

WTST is in control of the processor and memory resources with the cooperation of the node OS kernel.

WTST leases processor and memory resources to three VMs in a time-sliced and periodic manner. Each VM can be viewed conceptually as being periodically activated to run for a time-slice. Each VM is responsible for a major part of the functionality of TMOSM. Each VM maintains a number of application threads. In fact, whenever WTST assigns a time-slice to a VM, the VM in turn passes the time-slice onto one of the application threads that belong to it. The component in each VM that handles this "time-slice relay" is the application thread scheduler. For example, VM-A has the application-thread-scheduler VM-A-Scheduler. The application thread scheduler is actually executed by WTST. To be more precise, at the beginning of each time-slice, a timer-interrupt results in WTST being awakened. WTST then determines which VM should get this new time-slice. If VM-A is chosen, WTST executes VM-A-Scheduler and as a result, an application thread belonging to VM-A runs for a time-slice as WTST enters into the event-waiting mode.

The set of VMs is fixed at the TMOSM start time. One iteration of the execution of a specified set of VMs is called a *TMOSM cycle*. For example, one TMOSM cycle may be: VCT VMAT VAT VMAT. The following three VMs handle the core functions:

(1) VCT (*VM for Communication Threads*): The application threads maintained by this VM are those dedicated to handling the sending and receiving of middleware messages. Middleware messages are exchanged through the communication network among the middleware instantiations running on different DC nodes to support interaction among TMOs. Therefore, these application threads are called communication threads and denoted as CTs in Figure 3. A communication thread also distributes middleware messages coming through the network to their destination threads, typically belonging to another VM discussed below.

(2) VMAT (*VM for Main Application Threads*): The application threads maintained by this VM are those dedicated to executing methods of TMOs with maximal exploitation of concurrency. Those application threads are called main application threads and denoted as MATs in Figure 3. Normally to each execution of a method of an application TMO is dedicated a main application thread. In principle, TMO method executions may proceed concurrently whenever there are no data conflicts among the method executions. Every time-slice not used by the other VMs is normally given to this VM. In every one of our prototype implementations of TMOSM, the application thread scheduler in VMAT uses a kind of a deadline-driven policy for choosing a main application thread to receive the next time-slice.

(3) VAT (*VM for Auxiliary Threads*): This VM maintains a pool of threads which are called auxiliary threads and denoted as ATs in Figure 3. Some auxiliary threads are designed to be devoted to controlling certain peripherals under orders from TMO methods (executed by main application threads). Others wait for orders for executing certain application program-segments and such orders come from main application threads in execution of TMO methods. Use of this VAT has been motivated partly by the consideration that it should be easier to analyze the temporal predictability of the application computations handled by each VM, i.e., those handled by VMAT and those by VAT, than to analyze the temporal predictability of the application computations when there is no VAT and thus VMAT alone handles the combined set of application computations.

Also, WTST provides the services of checking for any deadline violations and if a violation is found, it provides an exception signal to the user.

We believe that structuring of VMs as periodic VMs is a fundamentally sound approach which leads to easier analysis of the worst-cast time behavior of the middleware without incurring any significant performance drawback.

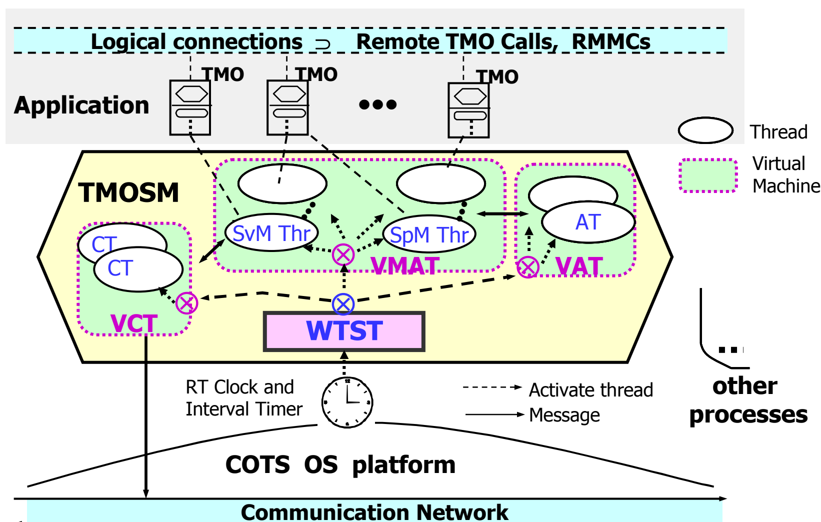

Figure 3. TMOSM architecture

3. Kernel Adaptation Layer

The *Kernel Adaptation Layer* (KAL) in TMOSM provides to the rest of TMOSM, the TMOSM *Main Layer*, an abstract interface that wraps the services of the underlying OS kernel in a form convenient for use by the Main Layer and the associated libraries. See Figure 4. The code of TMOSM is thus divided into two parts, TMOSM-KAL module and TMOSM-Main module. The KAL module contains all OS-dependent and language-dependent code. TMOSM-Main does not directly make system calls to the OS, instead it invokes corresponding KAL operations.

KAL provides APIs for:

(1) thread management,
(2) event handling,
(3) communications,
(4) heap memory acquisition,
(5) timer management,
(6) global time operations, and
(7) error reporting.

These abstractions are then implemented using OS kernel services, as described in Section 4. Communication services are a wrapper around sockets, while error reporting simply returns the last error. The other APIs are examined below.

Thread management: KAL provides a thread abstraction and operations to create, terminate, suspend, and resume threads, as well as manipulate a thread's priority. The suspend and resume operations are used by WTST (running at the highest priority) to manage the timing of application and other threads by enabling them during their permitted execution time, and suspending them when their time expires. The priority interface allows thread priority to be set and queried. Rather than an arbitrary range of priorities, the API defines *highest*, *high*, *medium*, and *low* settings. Since only the WTST runs at the highest priority, it preempts all others in order to manage execution of the latter.

Events: KAL provides events, which are used to synchronize threads. Events can be created, deleted, set (signaled), and reset. Threads can wait for one or more events to be set. The wait operations involve timeout value parameters and thus the amount of blocking that threads may experience can be controlled by design. Events may be "automatic," which means they are automatically reset when they activate a thread, or they may be manually reset.

Timers: Timer objects are programmable timers that signal an event when their specified interval expires. Timers may be programmed to signal once or periodically. Currently TMOSM uses a timer to awaken the WTST every three milliseconds.

Time service: This interface provides the global time reference and a sleep function that suspends the calling thread for a specified duration. The precision of the global time base offered can also be queried.

Heap memory: The memory pool interface allows TMOSM to create a heap region and destroy the allocated heap region.

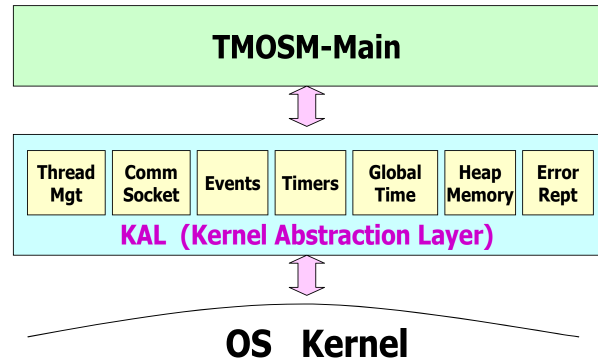

Figure 4. The Kernel Adaptation Layer

4. TMOSM/Linux implementation details

Development of a KAL/Linux that bridges the TMOSM Main Layer and the Linux kernel platform involved resolving a few non-trivial issues. Beyond the OS differences, the KAL also hides language and compiler differences. For example, a 64-bit integer is of type *int64* in Microsoft Visual C++, but it is a *long long* in the Gnu C Compiler (gcc). The KAL provides abstract types that hide these details.

4.1. Threads

On the basis of portability and programmability considerations, the POSIX thread (pthread) Library was used in the implementation of the thread management services of the KAL/Linux. The pthread Library provides APIs for thread creation and deletion, and some APIs for scheduling settings, e.g. APIs for changing the priority of a thread. However, it does not provide APIs for suspending and resuming of one thread by another thread as required by KAL and provided by the Windows APIs. It has turned out that the *signal* mechanism provides a way to implement the needed APIs with the pthread Library. The signal provides a way for one thread or process to notify a thread or process (possibly itself) of an event. The signal does not carry information other than which signal was sent. Threads expecting signals can designate handler functions that are called when a signal is sent to them. Upon arrival of a signal at a thread, the

normal flow of the thread is interrupted and the control jumps to the signal handler immediately. When the handler finishes, the execution control of the thread normally resumes at the point where it left off before the signal.

If there is a blocking call inside the signal handler, the thread will then be blocked and can not continue. The suspend/resume functionality in KAL/Linux is implemented by making use of this feature. For this purpose, two user signals, SIGUSR1 and SIGUSR2 are used. A signal handler for SIGUSR1 is attached to each thread. Inside this signal handler, the thread makes a blocking system call *sigwait (SIGUSR2)* to suspend itself until another signal SIGUSR2 arrives. Hence, if a signal SIGUSR1 is sent to one thread, the thread will be blocked inside the SIGUSR1 handler. When a signal SIGUSR2 is sent to the same thread, the thread is awakened and resumes its execution. These procedures for suspending and resuming a thread are depicted in Figure 5.

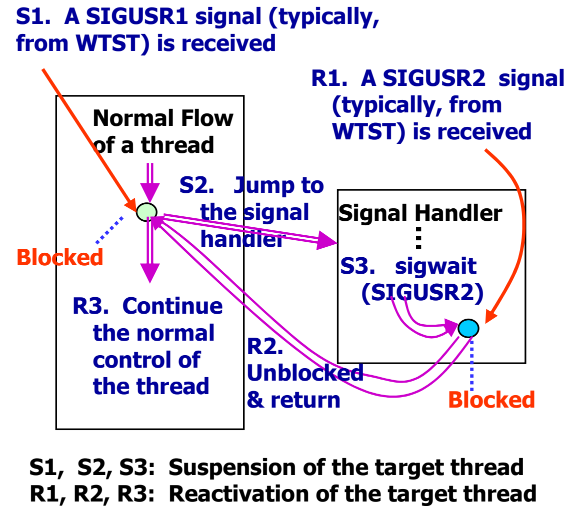

Figure 5. Suspending and reactivating a thread

4.2. Scheduling

TMOSM is a middleware system that manages thread scheduling and communications scheduling, some parts in indirect manners. Since it runs above the OS kernel, it relies on kernel services to achieve context switching among the threads and also the services of directly interfacing with hardware resources, but otherwise needs the OS to stay out of the way.

Linux provides basic disciplines for thread scheduling. There are three *preemptive* scheduling policies for threads: *SCHED_FIFO, SCHED_RR* and *SCHED_OTHER*. The *SCHED_FIFO* and *SCHED_RR* are priority-based scheduling policies. As implied by the names, the difference between *SCHED_FIFO* and *SCHED_RR* is in that for the threads with the same priority the former uses the first-in first-out ordering while the latter uses the round robin ordering. The thread priority range for these two policies is from 0 to 99. If the scheduling policy of one thread is set to be one of these two policies, the priority of the thread remains to be fixed unless it is changed by use of the APIs for priority setting. *SCHED_OTHER* is the default time-sharing scheduler policy used by most processes. The priority of one thread with this scheduling policy is dynamically adjusted based on program behaviors. In principle, the initial priority for a thread with *SCHED_OTHER* policy can be set between 100 and 140.

In order to ensure timely executions of TMOSM threads, the scheduling policy for each of them is set to SCHED_RR. Again, the priority of the super-thread, WTST is set to the highest one to avoid disturbances.

4.3. Events and the WOMS mechanism

TMOSM requires a mechanism which can be viewed as an extension of *sigwait()* illustrated in Figure 4. The needed mechanism can be called the *wait-for-one-of-multiple-signals* (WOMS) mechanism.
The *sigwait()* in Linux is inadequate in several respects. First, only two signals are available for use by applications. Second, when two signals arrive at a thread at nearly the same time, the earlier signal gets overwritten by the later. Real-time signals were introduced in recent versions of the Linux kernel, but they appeared to be immature for critical usage such as this.

The WOMS mechanism was simulated by use of *pipes* and the *select()* system call. We call the signals handled by the WOMS mechanism *events* to distinguish them from the signals handled by *sigwait()*. When a thread needs to wait for an event, it calls *select()* to wait for an event notice from a pipe. When another thread needs to "set the event", i.e., send the signal, it writes an event notice into the pipe. This approach ensures that the event notices are not lost even if multiple events are set nearly at the same time.

The procedures of *WaitForEvent()* and *SetEvent()* of WOMS are shown in Figure 6.

4.4. Heap memory management

Windows provides mechanisms by which threads can allocate memory from private heaps, thus eliminating the locking and delays encountered when using a shared heap. The KAL was originally designed with this capability in mind. Linux does not inherently provide such capabilities and thus in our adaptation ap-

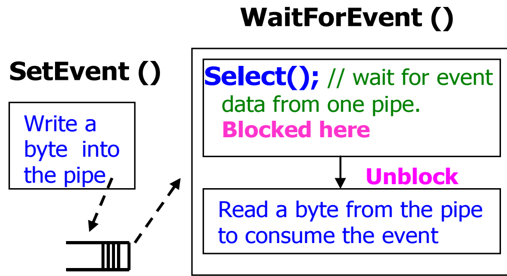

Figure 6. WaitForEvent() and SetEvent() of WOMS

proach, the global heap is acquired by the TMOSM Main Layer through the KAL from the Linux kernel and then the private heap behavior is emulated by a memory manager in the Main Layer. Specialized heaps are allocated from the global heap and then objects (data created to support execution of TMOs) are allocated from those specialized heaps by the memory manager in the Main Layer. This means that much of the contention can be avoided and good performance is maintained.

5. Performance analysis of TMOSM/Linux

One of the most timing-critical distributed activities in a TMO system is the synchronization of the global clock via network messages over the LAN [10]. In this activity, TMOSM instances on different nodes exchange time synchronization messages in a manner similar to the Network Time Protocol (NTP) [11]. TMOSM achieves higher precision than NTP's normal accuracy (1-2 milliseconds) because it is tailored for LAN environments and does not need to handle the long, unpredictable latencies of the Internet and also because it periodically prepares the system in a special mode for minimal-interference exchanges of clock-resynchronization messages. Since the clock synchronization operation involves most aspects of TMOSM, from message passing to scheduling to event handling, it is an illuminating benchmark of TMOSM/Linux performance.

Another important performance measurement is the *application thread switch time*. TMOSM uses a separate thread for each TMO method execution. TMOSM/Linux requires the use of signals and thread self-suspending rather than the external thread suspension and resumption provided by Windows. Therefore, the expectation is that the Linux version will have slower application thread switch times because of the larger number of kernel/user mode transitions and their associated performance penalties.

5.1. Experimental Setup

The global time synchronization experiments used a network of three machines connected with 100 Mbit/second switched Ethernet. The machines' CPU characteristics are listed in Table 1, and each has 512MB of RAM. The context switch time measurements were performed on Machine 1.

Table 1. Experimental system configuration

Machine	Processor Type	CPU Speed
1	Pentium 4	1.8 GHz
2	Pentium III	600 MHz
3	Pentium III	667 MHz

5.2. Results

Table 2 shows the global clock sync results when Machines 2 and 3 synchronize with Machine 1. The worst-case clock deviation of about 300 microseconds is easily sufficient for real-time systems that require millisecond precision. The average case from more than 1000 measurements is close to 100 microseconds, thus the clocks are well synchronized for a millisecond-precision system. Figure 7 shows the variance in measurements for Machine 2, and while there are some outliers, the bulk of the measurements are very near the average case.

Table 2. Clock sync results

Machine	Average Sync	Max Sync
2	110 μsecs	253 μsecs
3	128 μsecs	307 μsecs

TMOSM/XP shows similar results for the clock synchronization measurements. The average clock deviation is 109 microseconds with the worst case of 326 microseconds as shown in Figure 8. Therefore, the magnitude of clock deviation is roughly the same for the Linux and Windows versions of TMOSM. In both cases the vast majority of the synchronization measurements are near the average.

The slight performance difference between two platforms can be attributed to several factors, e.g., different running processes / threads tied to the kernels and different networking architectures (INET vs. NDIS), which introduce different communication jitters at the software level and affect the clock synchronization precision.

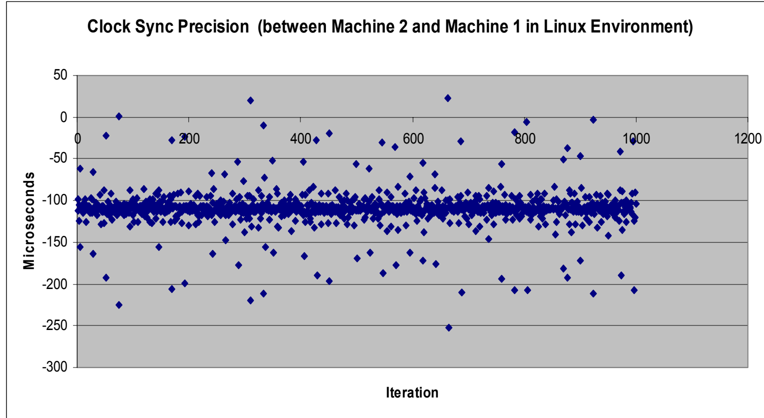

Figure 7. Clock sync precision in Linux environment

The context switch measurements in Table 3 show that the TMOSM/Linux signal-based approach takes somewhat more time to switch between application threads than the Windows version does. This is expected because of the sequence of actions required for the thread switch. First, WTST sends a signal to the running thread to cause it to suspend itself, and then WTST sends a signal to the next thread to cause it to resume. Each signaling call for suspending an application thread requires a switch from user mode to kernel mode and back, which is expensive on modern processors with deep pipelines such as Pentium 4. The average of more than 11,000 measurements is 51 microseconds for Linux, while the TMOSM/XP takes only 30 microseconds. Even the worst-case time of roughly 100 microseconds easily allows application thread management with timing precision far better than the millisecond precision. Figure 9 shows that the vast majority of thread switch times are close to the average, while very few are significantly worse.

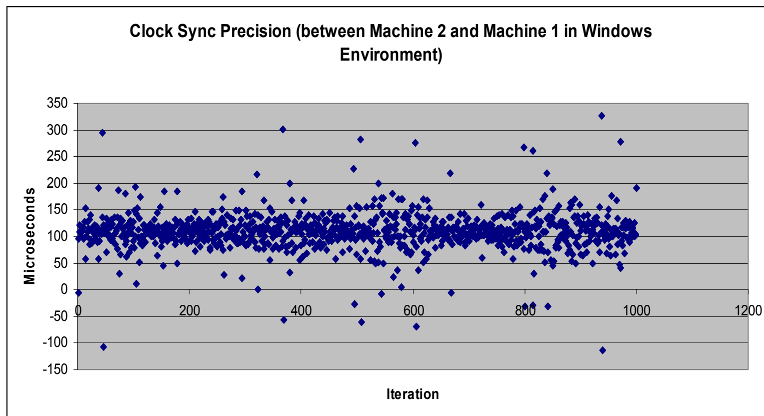

Figure 8. Clock sync precision in Windows environment

Table 3. Thread switch time

TMOSM/KAL version	Average (μsecs)	Max (μsecs)
Linux	51	104
Windows	30	86

6. Conclusion

This paper shows that the powerful TMO Support Middleware, which provides execution support for to distributed RT application systems composed as networks of high-level RT DC objects, was designed to possess a high degree of portability in that porting between Linux platforms and Windows platforms has become a relatively easy task. TMOSM provides precise user-level scheduling, so it essentially takes over the target machine and schedules the execution of user threads and system operations so that critical deadlines may be met. Because of this, TMOSM, more specifically, its KAL, requires significant and detailed interaction and support from the underlying OS kernel.

Windows and Linux provide significantly different thread support models, so much of the porting effort involved adapting the KAL to the Linux threading approach. Because TMOSM requires a mechanism to allow one thread to suspend and resume another thread, nested signal operations are used to emulate this functionality. Linux pipes are used to implement reliable ordered event channels with powerful reception/waiting options. Other KAL features required similar unconventional adaptation techniques.

The performance results show that the Linux version of TMOSM performs well and will easily support millisecond-precision RT computing application systems on moderate commodity hardware. This will allow much wider distribution options and flexibility for TMO and its associated middleware to the distributed and embedded computing system communities. It will also provide an initial step towards running TMOs on many current-generation cluster computers with high performance interfaces.

Acknowledgment: The work reported here was supported in part by the NSF under Grant Numbers 02-04050 (NGS) and 03-26606 (ITR), in part by SKKU under Contract Number M2004A−SKKU−UCI, and in part by ETRI. No part of this paper represents the views and opinions of any of the sponsors mentioned above.

References

[1] K. H. Kim, "APIs for Real-Time Distributed Object Programming," *IEEE Computer*, pp. 72-80, June 2000.

[2] K. H. Kim, "Object Structures for Real-Time Systems and Simulators," *IEEE Computer*, vol. 30, no. 8, pp. 62-70, August 1997.

[3] K. H. Kim, "Commanding and Reactive Control of Peripherals in the TMO Programming Scheme," in Proceedings of 5th IEEE CS Int'l Symp. on OO Real-time distributed Computing (ISORC 2002), Crystal City, VA 2002.

[4] K. H. Kim, "Basic Program Structures for Avoiding Priority Inversions," in Proceedings of IEEE CS 6th Int'l Symposium on Object-oriented Real-time distributed Computing (ISORC 2003), Hakodate, Japan, 2003.

[5] K. H. Kim, M. Ishida, and J. Liu, "An Efficient Middleware Architecture Supporting Time-Triggered Message-Triggered Objects and an NT-Based Implementation," in Proceedings of 2nd IEEE CS Int'l Symp. on Object-Oriented Real-time Distributed Computing (ISORC *99), St. Malo, France 1999.

[6] K. H. Kim, J. Q. Liu, H. Miyazaki, and E. H. Shokri, "CORBA Service Middleware Enabling High-Level High-Precision Real-Time Distributed Object Programming," *Computer System Science & Engineering*, vol. 17, no. 2, pp. 77-84, March 2002.

[7] H. J. Kim, S. H. Park, and M. H. Kim, "TMO-Linux: A Linux-Based Real-Time Operating System Supporting Execution of TMOs," in Proceedings of 5th IEEE CS Int'l Symp. on OO Real-time distributed Computing (ISORC 2002), Washington, DC, 2002.

[8] K. H. Kim and J. Q. Liu, "Going Beyond Deadline-Driven Low-Level Scheduling in Distributed Real-Time Computing Systems," *Design and Analysis of Distributed Embedded Systems (Proc. IFIP 17th World Computer Congress, TC10 Stream, Montreal, Can., Aug 2002), B. Kleinjohann et al. eds.*, pp. 205-215, 2002.

[9] K. H. Kim, J. Q. Liu, M. H., and E. H. Shokri, "TMOES: A CORBA Service Middleware Enabling High-Level Real-Time Object Programming," in Proceedings of IEEE CS 5th Int'l Symp. on Autonomous Decentralized Systems (ISADS 2001), Dallas, TX, 2001.

[10] K. H. Kim, C. S. Im, and P. Athreya, "Realization of a Distributed OS Component for Internal Clock Synchronization in a LAN Environment," in Proceedings of 5th IEEE CS Int'l Symp. on OO Real-time Distributed Computing (ISORC 2002), Crystal City, VA, 2002.

[11] D. L. Mills, "Internet Time Synchronization: The Network Time Protocol," *IEEE Transactions on Communications*, vol. 39, no. 10, pp. 1482-1493, Oct. 1991.

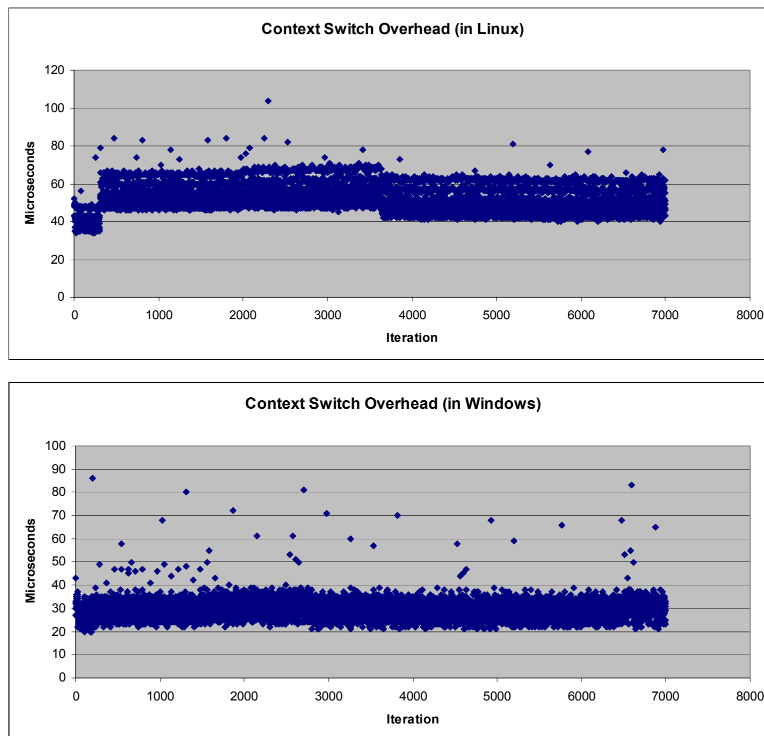

Figure 9. Application thread switch time