Proc. ISORC '99 (IEEE CS Int'l Symp. on Object-oriented Real-time distributed Computing), May 1999, St. Malo, France, pp.54-63.

An Efficient Middleware Architecture Supporting Time-Triggered Message-Triggered Objects and an NT-based Implementation

K. H. (Kane) Kim, Masaki Ishida, and Juqiang Liu
DREAM Lab., UCI
Irvine, CA 92697, USA
{kane, masaki, jqliu} @ece.uci.edu

Abstract: The time-triggered message-triggered object (TMO) structuring scheme has been established to remove the limitation of conventional object structuring techniques in developing applications containing real-time (RT) distributed computing components. It is a natural and syntactically small but semantically powerful extension of the object oriented (OO) design and implementation techniques which allows the system designer to abstractly and yet accurately specify timing characteristics of data and function components of high-level distributed computing objects. It is a unified approach for design and implementation of both RT and non-RT distributed applications. A cost-effective way to support TMO-structured distributed RT programming is to build a TMO execution engine as a middleware running on well established commercial software/hardware platforms. In this paper, we present an efficient middleware architecture named TMO Support Middleware (TMOSM) which can be easily adapted to many commercial-off-the-shelf (COTS) platforms. The performance of a prototype implementation of TMOSM running on Windows NT platforms is also discussed.

Keywords: time-triggered, message triggered, object, TMO, middleware, watchdog timer, scheduler, thread, TMOSM, atomic section, object data store, spontaneous method, NT.

1. Introduction

In recent years, there have been several efforts [Att91, Ish90, Tak92, Kim94] for extending conventional object structuring approaches into distributed software component structuring approaches that allow explicit specification of action timing requirements. One such sharable component structuring approach adopted by us is called the *Time-Triggered Message-Triggered Object* (TMO) structuring scheme [Kim94, Kim97b]. It is a natural and syntactically small but semantically powerful extension of the conventional object-oriented design and implementation techniques.

The TMO structuring scheme has the several unique features of fundamental nature. For example, in addition to traditional *service methods* (called by client objects), each TMO may also have *time-triggered (TT) methods* (activated when the real-time clock reaches specific points in time). Several experiments on the TMO structuring conducted in the context of a variety of applications ranging from military applications to factory automations [Kim96, Kim97b], validated the proposition that the TMO-structured design of RT systems offers a rigorous way to develop complex RT systems in easily understandable forms.

To facilitate TMO-based design of RT systems in the most cost-effective manner, it is essential to provide the execution support mechanisms on well-established commercial software/hardware platforms. The first TMO execution engine which was designed by the first co-author was named the *DREAM (Distributed Real-time Ever Available Micro-computing) kernel* [Kim96, Kim98]. The lower layers part of the DREAM kernel was also intended to be a model of an operating system (OS) kernel which can support RT processes with guaranteed timely kernel services. Several prototype implementations of the DREAM kernel with a user-friendly interface, called DREAM library, have been produced by the first co-author and his research collaborators but they were based on MS DOS platforms which are now on the demise.

More recently, middleware implementations of TMO support facilities based on commercial-off-the-shelf (COTS) software/hardware platforms in wider use today, e.g., Sun Solaris and Windows NT, have been produced [Sho98]. Partly taking advantage of these experiences, the authors formulated a refined middleware architecture that can be adapted to a variety of COTS platforms. This middleware architecture named the *TMO support middleware* (TMOSM) is more efficient than the architectures of the previously implemented middleware in terms of the *precision of action timings supported*. It also supports the current TMO programming model fully whereas previously implemented middleware do not support the programmable data-field-channel (DFC) [Kim97a], one of the significant features of the current TMO structure. Therefore, the main theme of this paper is to present this TMOSM and discuss its strengths and limitations.

Also, a prototype implementation of TMOSM on the Windows NT platform [Ric97] has been completed recently (available from the web http://dream.eng.uci.edu). The performance characteristics of this prototype are discussed.

Overall, the new middleware architecture TMOSM is closer to the layer L4 of the DREAM kernel

0-7695-0207-5/99 $10.00 © 1999 IEEE

architecture. Actually, the DREAM kernel can be viewed as a non-commercial kernel (layers L0–L3) plus a middleware supporting TMO's (layer L4), although the DREAM kernel is not referred by the term middleware in this paper. TMOSM can be as easily adapted to run on COTS operating systems (OS's) as previously experimented middleware architectures can be. The main differences between TMOSM and the previously experimented architectures can be summarized as follows:

i) The inter-node communication is based on UDP broadcasts, neither object request broker (ORB) services [OMG95] nor point-to-point protocols;

ii) Concurrency control inside the middleware is done via *thread-to-thread atomic section* (TTAS) and *application-to-application atomic section* (AAAS) (which will be explained in Section 3), not via semaphores as was done previously;

iii) Executions of computation segments are scheduled at two levels, segments of *middleware- threads* at one level and segments of *application-method-threads* at another level, whereas one-level scheduling was used in previous middleware architectures;

iv) Programmable DFC support is added, which was missing previously.

This paper starts with a brief overview of the backgrounds on the TMO structuring scheme and the essential features of the OS kernel used for supporting TMOSM. The TMOSM architecture is discussed in Section 3. Section 4 then presents our TMOSM implementation on the Windows NT platform and its performance. The paper concludes in Section 5.

2. Backgrounds

2.1 An overview of the TMO structuring scheme

The *Time-triggered Message-triggered Object (TMO)* structuring scheme was established a few years ago [Kim94, Kim97b] with a concrete syntactic structure and execution semantics for economical reliable design and implementation of real-time systems. TMO is depicted in Figure 1 and some of its major characteristics are as follows:

(a) *Spontaneous method*: The TMO contains new types of methods, *time-triggered (TT) methods*, also called the *spontaneous methods* (SpM's), which are clearly separated from the conventional *service methods* (SvM's). The SpM executions are triggered when the real-time clock reaches specific values determined at design time, whereas the SvM executions are triggered by service request messages from clients. Moreover, actions to be taken at real times that can be determined at the design time can appear only in SpM's.

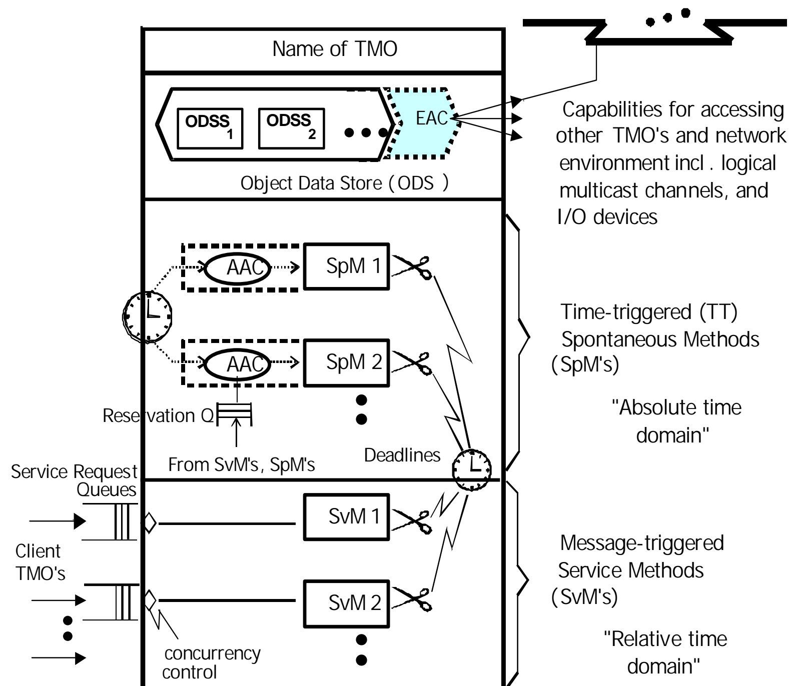


Figure 1. The basic structure of TMO (Adapted from [Kim97b])

(b) *Basic concurrency constraint (BCC)*: Under this rule, SvM's cannot disturb the executions of SpM's and the designer's efforts in guaranteeing timely service capabilities of TMO's are greatly simplified. Basically, *activation of an SvM triggered by a message from an external client is allowed only when potentially conflicting SpM executions are not in place.* An SvM is allowed to execute only if no SpM that accesses the same portion of the Object Data Store (ODS) to be accessed by this SvM has an execution time window that will overlap with the execution time window of this SvM. However, the BCC does not stand in the way of either concurrent SpM executions or concurrent SvM executions.

(c) A *time window* is imposed for each output action and completion of a method of a TMO.

Triggering times for SpM's must be fully specified as constants during the design time. Those real-time constants appear in the first clause of an SpM specification called the *autonomous activation condition* (AAC) section. An example of an AAC is

"<u>for</u> t = from 10am <u>to</u> 10:50am every 30min
<u>start-during</u> (t, t+5min) <u>finish-by</u> 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" }

A provision is also made for making the AAC section of an SpM contain only candidate triggering times, not actual triggering times, so that a subset of the candidate triggering times indicated in the AAC section may be dynamically chosen for actual triggering. Such a dynamic selection occurs when an SvM within the same TMO object requests future executions of a specific SpM. The AAC specifying candidate triggering times rather than actual triggering times starts with a declaration "if-demanded".

Extensions (a) and (b) are unique to the TMO structure in comparison with other proposed object extensions [Att91, Ish90, Tak92]. Client methods (SpM's or SvM's) may request the services of SvM's in other TMO's. To maximize concurrency in the execution of client and server methods, client methods are allowed to make non-blocking (sometimes called asynchronous) types of service requests to SvM's.

The designer of each TMO indicates the *time-window for every output* produced by each SvM (and by each SpM which may be executed on requests from SvM's) in the specification of the SvM (and some relevant SpM's) and advertises this to the designers of potential client objects. The designer of the server object thus guarantees the timely services of the object. Before determining the time-window specification, the server object designer must make sure that with the available *object execution engine* (hardware plus operating system) the server object can be implemented such that the output actions are performed within the specified time-windows. Also, the designer of TMO object can view SpM's as internal capabilities and SvM's as services advertised to all potential clients.

2.2 Essential features of a candidate operating system kernel

To support a TMOSM implementation which in turn supports application TMO's, the OS kernel should possess the following capabilities:

(a) High-resolution timer interrupt

The OS kernel on a node should offer a RT clock synchronized with the clocks on other cooperating nodes. This is necessary to enable a TMO to obtain a reasonably accurate reading of the current time-of-the-day whenever it needs regardless of which node it is running on. It should also provide some degree of *alarm-clock services*, at least a simple watch-dog timer, which can be used for implementation of *time-sliced thread scheduling*. TMOSM must effect *TT* actions of TMO's. Therefore, the timing of the triggering signal generation must be accurate and the gap between the timer's generation of a signal and the activation of a relevant action must be within an acceptably small bound. The granularity of the base clock supporting the globally synchronized timer, which differs in different OS/hardware platforms, is a major factor to consider in selecting a platform for a specific application domain.

(b) Top-priority real-time thread support

"RT thread" is a generic construct which has the following characteristics:
(i) the delay it experiences in accessing a resource should not exceed a predetermined tight bound,
(ii) its priority is as high as that of any other thread and thus its execution can not be interrupted just for the sake of executing another thread, and
(iii) when a system service call is issued by an RT thread, the system call inherits the RT characteristics of its issuer thread.
Such an RT thread can be used to realize a high-level timer which drives all other middleware threads of TMOSM.

In addition, it is desirable to use an OS kernel that enables calculation of a tight bound on the amount of time spent handling interrupts during any time-slice or any interval of a few time-slices. In this respect, current COTS OS kernels have some shortcomings but are expected to improve in coming years. DREAM kernel is a demonstration of the feasibility of such evolution.

The advantages of implementing the TMO execution support facility as middleware are:
(i) Economy of implementation and maintenance: No changes are made to the OS kernel;
(ii) Better portability: The TMO execution engine can be ported with small effort onto any OS which has the basic features mentioned in this section.

3. TMOSM: A middleware architecture supporting TMO executions

Major features of the TMOSM architecture are now discussed.

3.1 Threads in TMOSM

Figure 2 shows the internal thread structure of TMOSM. There are two types of threads in TMOSM, the *application thread* and the *middleware thread* (also called system thread). An application thread executes a method (SpM or SvM) of an application TMO as assigned by TMOSM. Middleware threads are *periodic threads* (periodically activated by high-precision timer interrupts), each responsible for a major part of the functions of TMOSM. Contrasted with the application thread, the set of middleware threads is fixed at the TMOSM loading time. There are four essential middleware threads.

(1) WTST (Watchdog Timer & Scheduler Thread): This periodic thread manages the scheduling / activation of all other threads in TMOSM and

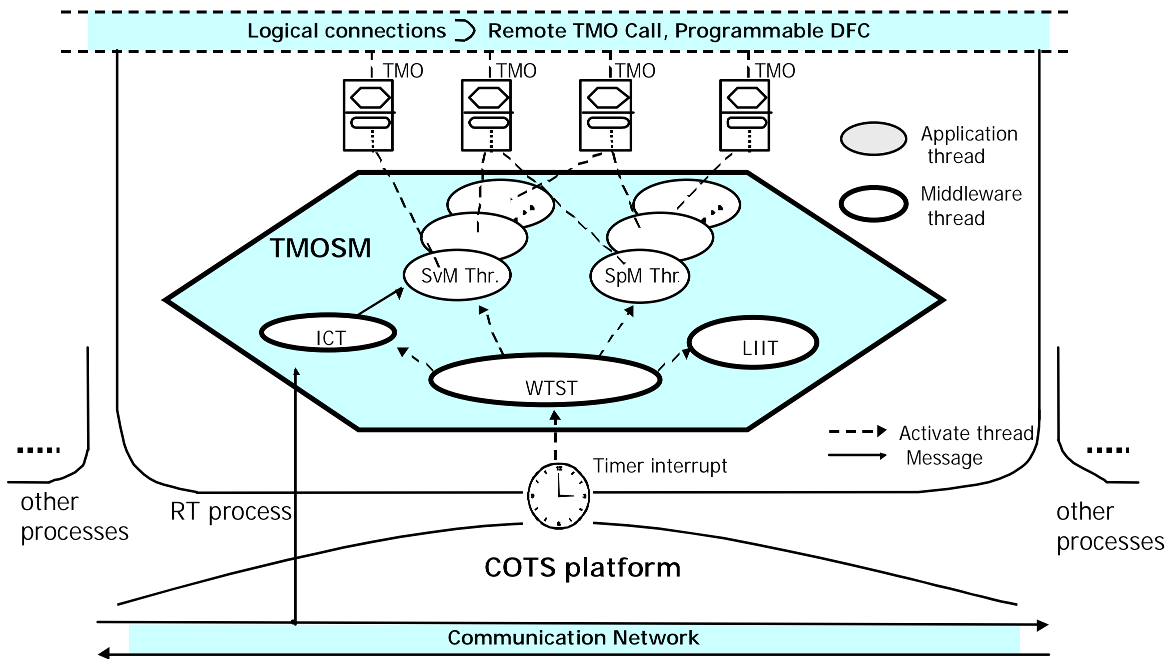


Figure 2. The Basic Internal Thread Structure of TMOSM

checks if there are deadline violations;

(2) ICT (Incoming Communication Thread): This periodic thread manages the distribution of messages coming through the communication network to the destination threads;

(3) LIIT (Local I/O Interface Thread): This periodic thread manages local I/O activities such as serial character I/O and disk I/O; and

(4) VMST (Virtual Main System Thread): Every time-slice not used by the above three middleware threads is conceptually given to this virtual thread which merely represents all application threads. The actual time-slice allocation actions are taken by WTST which executes the application scheduler function and every time-slice conceptually belonging to VMST is allocated to a fairly selected application thread. Therefore, VMST is a virtual thread representing all application threads.

3.2 Two-level scheduling

Figure 3 shows how the machine time is divided into time-slices which are grouped into TMOSM cycles. TMOSM adopts the two-level scheduling policy under which middleware threads are scheduled at the base level and then the time-slices allocated to VMST are distributed to application threads at another level. The amount of machine time spent by WTST is not shown explicitly in the figure. In fact, at the beginning of every time-slice, a small portion of the time-slice is spent by WTST.

WTST first executes the middleware thread scheduler function to select the next middleware thread – one among ICT, LIIT, and VMST. From the middleware thread scheduler point of view, all application threads in TMOSM are viewed as one virtual middleware thread, VMST. When the middleware thread scheduler selects VMST as the next thread to be executed, the application thread scheduler function is invoked to select the next application thread to be executed according to the application scheduling policy.

If there are just two application threads which are ready and if they are allocated the time-slices given to VMST in an alternating fashion, each application thread will be given at least one time-slice in every two TMOSM cycles. Often ICT and LIIT do not use up their time-slices fully because there is no work to be done and in such a case, the unused remainder of the time-slice may be given to VMST. In our prototype implementation, such unused remainder of a time-slice is given to other threads belonging to basic Windows NT system processes (e.g., threads running the memory manager, the cache manager, some device drivers, etc.).

There are advantages in this two-level scheduling over the one-level scheduling, which treats both middleware threads and application threads together as one group of competing threads. The most important advantage is the ease of analyzing the timing behavior of

various threads. Middleware threads are periodic threads and thus their timing behavior is highly predictable. The minimum number of time-slices given to an application thread during a certain interval is also relatively easy to determine under this two-level scheduling.

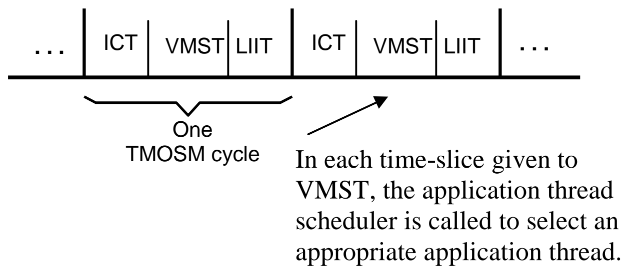


Figure 3. TMOSM scheduling cycle

3.3 Thread-to-Thread Atomic Section (TTAS) and Application-to-Application Atomic Section (AAAS)

The TTAS mechanism is used to avoid conflicts among middleware threads and between middleware threads and application threads in accessing shared data without using locks. A thread needing to access a shared data structure orders WTST not to disturb it, i.e., orders WTST to disable the thread switch so as not to let the time-slice be taken away from the ordering thread until the thread notifies WTST that the *disturbance prohibition period* is over. So even if a thread-time-slice expiration occurs during the disturbance prohibition period (i.e., while the thread is accessing the shared data), WTST does not change the running thread. Such a code-segment during the execution of which a thread accesses shared data structures is called a *thread-to-thread atomic section* (TTAS). The sending of a no-disturbance request to WTST is called an "Enter-TTAS" operation, and the cancellation of a no-disturbance request is called an "Exit-TTAS" operation.

The AAAS mechanism can be used in a similar manner. So, if an AAAS is in execution, WTST does not change the user of the time-slices allocated to VMST, i.e., a change from the current application thread to another application thread is prohibited. Independent of this, the middleware thread scheduling will proceed.

The other way to manage access to shared data is to use data locks, such as semaphore, provided by the OS. However, compared to the TTAS-AAAS approach, such an approach is much less efficient when used inside middleware such as TMOSM. To be more specific, a thread Thr1 can be preempted in the middle of its atomic section if the lock approach is used. Then the other thread Thr2 which needs to access the shared data will get a time-slice some time later but it will have to yield the time-slice to another thread without succeeding in entering its atomic section since the shared data which it needs to access is locked. Later when Thr1 gets another time-slice, it can finish its atomic section and release the lock. Then Thr2 can enter its atomic section. Therefore, it is difficult to determine the worst-case execution time for the atomic section of Thr2 and moreover, the worst-case execution time is generally much greater than that under the TTAS-AAAS approach. Also, under the TTAS-AAAS approach, handling of potential data conflicts among middleware threads is clearly separated from the handling of potential data conflicts among application threads.

3.4 Object Data Space Segment (ODSS)

ODSS is a storage unit for which an object method in the host TMO obtains a read-only or read-write access right. Since the access rights possessed by object methods for ODSS's determine the possibilities of concurrent execution of the object methods, the segmentation of the object data store into ODSS's directly impact the degree of parallelism realizable in object executions. To facilitate dynamic decisions on the concurrent execution possibilities, each ODSS as well as the set of ODSS's that may be accessed by each TMO method should be registered with TMOSM.

An ODSS in a TMO may be concurrently accessed by multiple methods of the same TMO as long as the methods need to access the ODSS for the read-only purpose. The CREW(Concurrent-Read-and-Exclusive-Write) monitor mechanism [Kim96] is used inside TMOSM in supporting ODSS's.

3.5 Incoming Communication Thread (ICT)

ICT manages the distribution of messages coming through the communication network to the destination threads. The computational capacity of ICT is one factor determining the maximum incoming bandwidth that the host node can handle.

3.6 Local I/O Interface Thread (LIIT)

LIIT manages local I/O activities such as serial character I/O and disk I/O according to the requests from application threads or other middleware threads. Invoking local I/O actions from each application thread by directly calling underlying OS services, e.g., printf() or fopen(), is prohibited. This is because I/O operations are the most time-consuming and unpredictable operations and the timing analysis is difficult if all I/O operations are scattered in many threads. Thus making LIIT to be the sole authorized periodic thread for managing local I/O eases the analysis of the TMOSM's timing performance (especially in supporting more time-critical parts of the applications) significantly. In principle, we can make LIIT to accept a request message from a TMO method for any kind of I/O service offered by the underlying OS. However, the designer of the TMO method sending an I/O service request message to LIIT should have the understanding of the timing

behavior of the specific I/O service requested to be carried out by the OS.

3.7 Method execution management

Figure 4 shows the internal structure of TMOSM. At the method registration time, SpM and SvM are associated with threads created by TMOSM in advance and are stored in the Method Control Block (MCB) along with their execution parameters such as AAC (Autonomous activation conditions) for SpM's. Although both SpM and SvM are registered in similar fashions, TMOSM handles the two types of methods differently during run-time.

3.7.1 SpM activation

WTST periodically examines the registered SpM's in MCB and identifies the SpM's to be executed in the near future. The identified methods are placed in the "SpM-Reservation-Queue" for further analysis. Each SpM in SpM-Reservation-Queue is moved into "Ready-Application-Thread-Queue" later as the time-window for starting the SpM execution arrives. The application scheduler (executed by WTST) selects a thread in Ready-Application-Thread-Queue according to the scheduling policy each time a new time-slice (belonging to VMST) opens up.

3.7.2 SvM activation

In contrast to the SpM activation, which is totally handled by WTST, the SvM activation procedure is mainly executed by ICT. When receiving one service request message, ICT identifies the requested SvM in MCB, binds it to a thread if not done already, and places it in "Waiting-SvM-Thread-Queue". As each SvM in Waiting SvM-Thread-Queue passes the basic concurrency control (BCC) check, it is moved into Ready-Application-Thread-Queue.

3.7.3 Static creation and dynamic binding of threads

In our TMOSM architecture, application threads are created at the middleware initialization time and put into the thread pool. Later on those application threads are bound to or unbound from specific SpM's or SvM's as needed. In case TMOSM cannot find an unbound thread when a new method execution needs to be launched, TMOSM steals a thread from the method which is not currently activated and then binds it to the new method. By using this approach, we can avoid dynamic thread creation at run time as long as the adequate number of application threads are created at the initialization time.

3.8 Programmable Data Field Channel (DFC)

Programmable DFC's as well as point-to-point communications are implemented by using the Connectionless UDP (User Datagram Protocol) / IP broadcast protocol. The main reasons are.

(i) The UDP/IP protocol is supported by almost all kernels through the common *socket* mechanism.

(ii) A connectionless UDP protocol is preferred over the connection oriented TCP protocol because a connectionless protocol eliminates the overhead

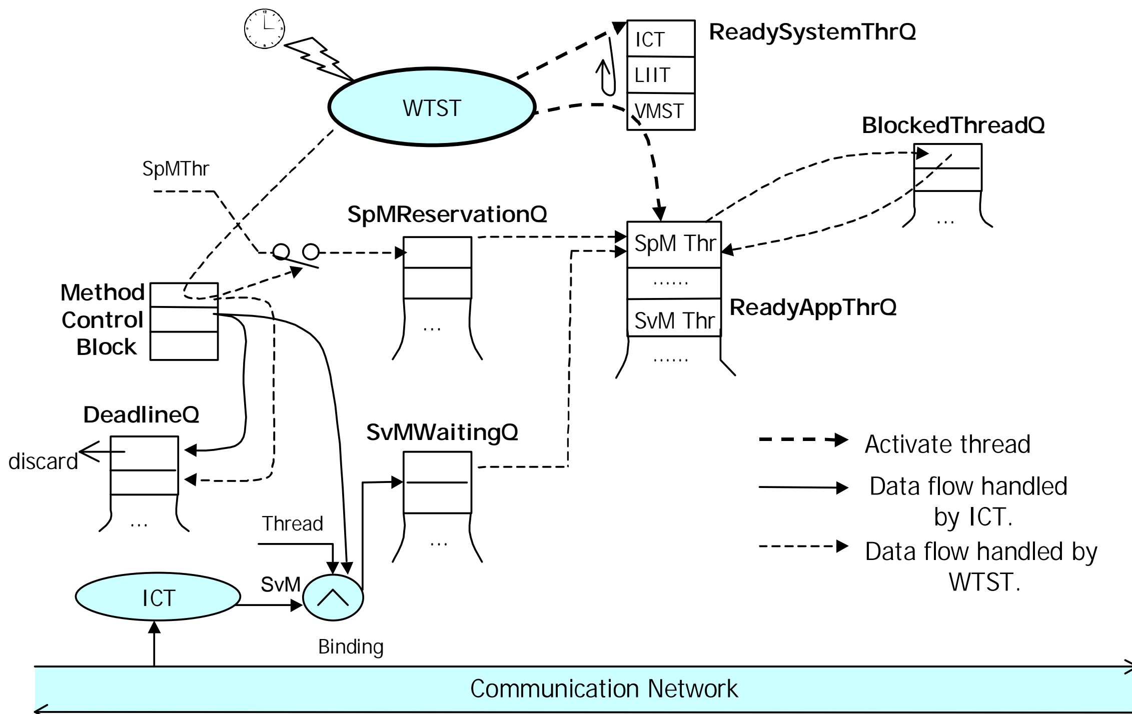


Figure 4. Internal structure of TMOSM

involved in creating and destroying a connection.

The programmable DFC scheme is the extension of the conventional data-field (DF) scheme [Mor93, Kim95]. The original DF scheme developed in [Mor86, Mor93] supports establishment of "logical multicast channels" called data field channels (DFC's) shared among the concurrent distributed processes for their interaction in such a way that the idiosyncrasies of the physical communication networks are transparent to the process designer. Each DFC was associated with a specific service process. The HU-DF scheme [Kim95] added the functionality of dynamic connection and disconnection between processes and DFC to the original DF scheme.

The programmable DFC is the logical multicast channel facilitating communication among the distributed TMO's. Thus it supports the object-level network transparency.

Unlike the channel in the original DF scheme, a programmable DFC can carry two types of messages, *event messages* and *state messages* [Kop89, Kim95]. An event message carries information about an event occurrence and every event message should be read by the intended TMO. An event message is not supposed to be overridden by another event message. On the other hand, a state message carries information to be stored in a fixed memory location in each receiver TMO. Each receiver TMO will read the content of its state message memory at its convenient time. This means that the producer object of a state message may update the contents of the state message memory units at a higher frequency than at which a certain receiver TMO object reads the content of its state message memory. A state message is thus typically used to share the periodically observed information about a dynamic entity, e.g., temperature of an oven. Programmable DFC's declared by a TMO network programmer must thus be facilitated by TMOSM.

4. A prototype implementation of TMOSM on Windows NT

The TMOSM architecture described in Section 3 can be easily implemented on tope of any OS which has the essential features discussed in Section 2.2. In this section, our recent prototype implementation of TMOSM on the Windows NT [Ric97] platform is briefly discussed.

4.1 Windows NT's features needed by TMOSM

As mentioned before, the main features needed by TMOSM are a high-precision timer and the RT thread mechanism. Windows NT is a preemptive multi-threaded OS that allows multiple processes/threads with different priority-levels to run concurrently. An application is constructed as a set of processes each of which consists of one or more threads. Windows NT uses a priority-based time-sliced scheduling policy and supports eight discrete thread priority levels.

Preemption in NT is based solely on the thread priority and threads in the same level run in the round-robin fashion with a time-slice equal to 25 milliseconds. Since a thread with the highest priority-level (THREAD_PRIORITY_TIME_CRITICAL) gets selected each time a new time-slice opens up, it releases the execution resources either when its execution is completed or when it voluntarily suspends itself to allow other threads to be executed. Obviously Windows NT's default scheduling time-slice (25ms) is too long for many TMO-structured applications.

The timer in Windows NT that can generate periodic interrupt signals at the highest frequency is the "Waitable Timer" [Ric97] construct. This timer can be used to "announce" the opening of each time-slice. With this facility, the thread-time-slice can be set as small as one millisecond. In our prototype implementation denoted by TMOSM/NT, it was set to be 3 milliseconds.

WTST reduces the role of the NT's scheduler to the minimum and works as the scheduler of all other middleware threads and application threads involved in execution of application TMO's. Every time a scheduling decision needs to be made, WTST selects one middleware or application thread and suspends all other threads and itself. The priority of WTST is the highest and all other threads in TMOSM are the second highest to ensure that they will not be interrupted by other NT threads. When the selected thread's time-slice expires, WTST will be waken up to select another thread.

To be more specific, the priority of the TMOSM process is REALTIME_PRIORITY_CLASS, the priority of WTST is THREAD_PRIORITY_TIME_ CRITICAL, and the priorities of other threads are THREAD_PRIORITY_HIGHEST.

TMOSM/NT is initialized to operate a certain number, say X, of threads. This pool of threads is utilized as follows;

(1) During the registration of each SpM, an available thread is bounded to the SpM If all X threads have been exhausted, more thread are created, 5 at a time.

(2) During the registration of each SvM, if there is an unbound thread (in X's), an available thread is bound to the SvM. If there is no unbound thread, then the SvM is not bound to any thread. at this time.

(3) When an SvM is called, it may or may not be already bound to a thread. If the SvM called is already bound to a thread and idling, then the thread can start the execution of the SvM. If the SvM called is already bound to a thread but the thread is in active execution of an earlier invocation, TMOSM/NT tries to find a thread which has been bound to SvM that is idling. If such a thread is found, then the existing binding between the thread and the idling SvM is removed and the thread is

rebound to the newly called SvM. If such a thread is not found, then TMOSM/NT checks if new threads can be created. If so, new thread are created, 5 at time, and a new thread is bound to the newly called SvM. Otherwise, TMOSM/NT waits until a thread bound to an SvM becomes idling. When an SvM called turns out to be unbound, TMOSM/NT again tries to either find an available thread or create new threads as in the case mentioned above where the SvM called is already bound to a thread but the thread is in active execution of an earlier invocation.

(4) When an SpM needs to be initiated while the thread bound to the SpM is still active, the TMOSM/NT tries to find a thread which has been bound to an SvM that is idling. If such a thread is found, then the existing binding between the thread and the latter SvM is removed and the thread is rebound to the new invocation of the SpM. If such a thread is not found, then new threads are created, 5 at a time, and a new thread is bound to the new SpM invocation.

4.2 Performance of the prototype implementation, TMOSM/NT

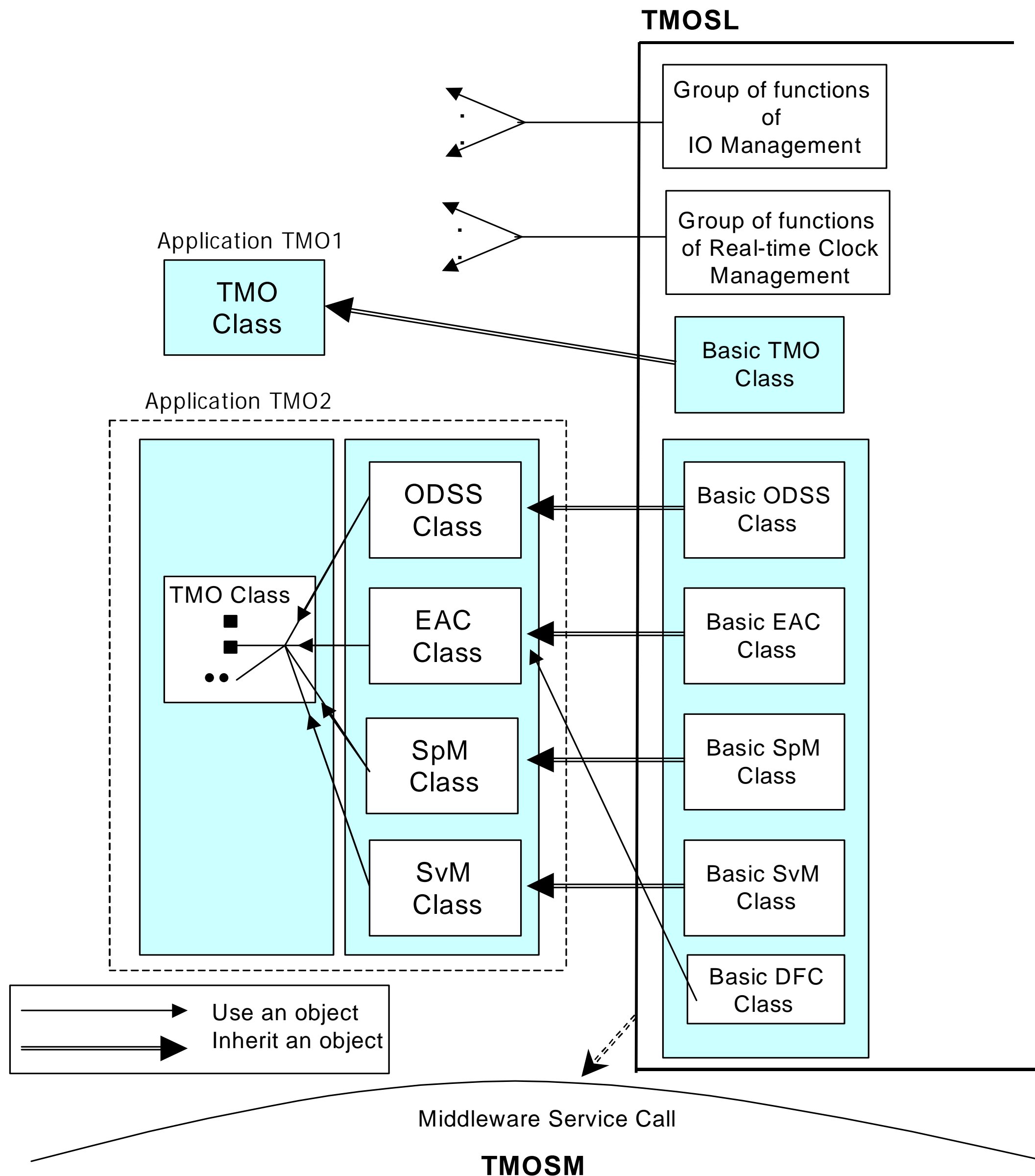


Figure 5 TMOSL structure

The basic time-slice for and middleware threads is 3 milliseconds. Switching the user of the time-slices allocated to VMST among application threads occurs each time after a user thread consumes four time-slices allocated to VMST, i.e., after a user thread is executed for 12 ms. Through construction of one non-trivial defense application prototype named CAMIN (Coordinated Anti-Missile Interception Network) which was originally implemented on DREAM kernel platforms [Kim98], we found that our TMOSM/NT can accurately support the time-window for activating a method as small as 10ms and the deadline as small as 20ms for more than 99.9% of the time.

The reason why 100% guarantee is not achieved for these high-precision action timings is due to some limitations of Windows NT. In order to prevent undesirable competition from non-RT processes in using execution resources during the execution of application TMO's, it is desirable to turn off all processes other than the TMOSM process. However, there is at least one process which actively uses execution resources and cannot be turned off. It is the basic NT system process composed of threads running the memory manager, the cache manager, etc. A good understanding of the timing behavior of these essential NT threads is important to realize the 100% guarantee on the timeliness of TMOSM. In addition, the timing behavior of some essential device drivers must be well understood since TMOSM threads can be interrupted by such device drivers. Also, each time WTST is invoked and selects a thread to be executed next, WTST suspends itself and all other threads except the selected thread and then the NT scheduler must dispatch the selected thread. Therefore, the dispatch latency is an interval from the instant at which WTST suspends itself to the instant the selected thread starts its execution. Measurement of this dispatch latency appears necessary to obtain good understanding of the NT's limitation.

4.3 TMOSL

To facilitate a user-friendly API for TMO programmers, a library named TMOSL (TMO Support Library) has been created. It consists of a collection of C++ classes and functions, and provides a user-friendly interface for TMOSM services for RT application programmers. The basic structure of TMOSL is shown in Figure 5.

5. Conclusion

In this paper, we presented a generic architecture for TMO Support Middleware which can be easily adapted to COTS platforms. We developed a prototype implementation of TMOSM on Windows NT to validate this proposition. The goal of the TMO structuring scheme is to realize significant reduction in costs of development and maintenance of large-scale RT systems as well as substantial increases in the overall robustness and dependability of application systems. An application development experiment has enhanced our conviction that the TMO structuring approach supported by facilities such as TMOSM/NT can support highly efficient and economic development of complex distributed RT applications with action timings of the precision in the range of around ten milliseconds.

Acknowledgments: The research work reported here was supported in part by the US Defense Advanced Research Project Agency under Contract N66001-97-C-8516 monitored by SPAWAR, and in part by Hitachi, Ltd.

Reference

[Att91] Attour, A. "An Oriented Model for Parallel and Reactive Systems", *Proc. IEEE CS 12th Real-Time Systms Symp.,* 1991, pp.84-93.

[Ish92] Ishikawa, Y. and Mercer, C.W., " An Object-Oriented Real-Time Programming Language", *IEEE Computer*, October 1992, pp. 66-73.

[Kim94] Kim, K.H. et al., "Distinguishing Features and Potential Roles of the RTO.k Object Model", *Proc 1994 IEEE CS Workshop on Object-oriented Real-time Dependable Systems (WORDS),* Oct. 94, Dana Point, pp. 36-45.

[Kim95] Kim, K.H., Mori, K., and Nakanishi, H., "Realization of Autonomous Decentralized Computing with the RTO.k Object Structuring Scheme and the HU-DF Inter-Process-Group Communication Scheme", *Proc. 1995 IEEE CS Int'l Symp. on Autonomous Decentralized Systems (ISADS)*, April 1995, Phoenix, pp.305-312.

[Kim96] Kim, K.H., et al, "The DREAM Library Support for PCD and RTO.k programming in C++", *Proc 2nd workshop on Object-Oriented Real-Time Dependable Systems*, February 1996, Laguna Beach, CA, pp. 59-68.

[Kim97a] Kim, K.H., Subbaraman, C., and Bacellar, L., "Support for RTO.k Object Structured Programming in C++", *Control Engineering Practices*, and IFAC journal, Vol. 5, No.7, 1997, pp. 983-991.

[Kim97b] Kim, K.H., "Object Structures for Real-Time Systems and Simulators", *IEEE Computer*, Vol. 30, No.8, August 1997, pp. 62-70.

[Kim98] Kim, K.H., Subbaraman, C., "Principles of Constructing a Timeliness-Guaranteed Kernel and Time-triggered Message-triggered Object Support Mechanisms", *Proc First International Symposium on Object-Oriented Real-time Distributed Computing (ISORC '98)*, Kyoto, Japan, April, 98, pp.80-89.

[Kop89] Kopetz, H., Damm, A., "Distributed Falut-Tolerant Real-Time Systems: The MARS Approach", *IEEE Micro*. Vol. 9. pp. 25-40.

[Mor86] Mori, K., et. al., "Autonomous Decentralized Software Structure and Its Application.", *Proc. Fall Joint Computer Conference*, Dallas, Texas, November 1986, pp. 1056-1063.

[Mor93] Mori, K, "Autonomous Decentralized System; Concept, Data Field Architecture, and Future Trends", *Proc. IEEE CS 9th Symp. on Reliable Distributed Systems, Huntsvile*, AL., Oct. 1990, pp.165-174.

[OMG95] Object Management Group, The Common Object Request Broker: Architecture and Specification, Revision 2.0, 1995.

[Ric97] Richter, J., *Advanced Windows*, 3rd Edition, Microsoft Press 1997.

[Sho98] Shokri, E., Crane. P., and Kim, K.H., "An Implementation Model for Time-Triggered Message-Triggered Object Support Mechanisms in CORBA-Compliant COTS Platforms", *Proc First International Symposium on Object-Oriented Real-time Distributed Computing (ISORC '98)*, Kyoto, Japan, April, 98, pp. 12-21.

[Tak92] Takashio, K., and Tokoro M., "DROL, An Object-Oriented Programming Language for Distributed Real-Time Systems", *Proc. OOPLA*, 1992, pp. 276-294.

[Tim97] Timmerman. M., Monfret. J.C., "Windows NT as Real-Time OS?", *Real-Time Magazine*, Issue 97/2.

Proceedings

2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'99)

2–5 May 1999

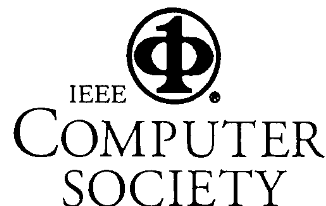
Saint-Malo, France

Sponsored by

IEEE Computer Society Technical Committee on Distributed Processing

In cooperation with

IFIP Working Group 10.4
OMG
INRIA



Los Alamitos, California
Washington • Brussels • Tokyo