

An Implementation Model for Time-Triggered Message-Triggered Object Support Mechanisms in CORBA-Compliant COTS Platforms

Eltefaat Shokri and Patrick Crane
SoHaR Inc.
Beverly Hills, CA

Kane Kim
University of California, Irvine

Abstract

Object-oriented analysis and design methodologies have become popular in development of non-real-time business data processing applications. However, conventional object-oriented techniques have had minimal impacts on development of real-time applications mainly because these techniques do not explicitly address key characteristics of real-time systems, in particular, timing requirements. The Time-triggered Message-triggered Object (TMO) structuring is in our view the most natural extension of the object-oriented design and implementation techniques which allows the system designer to explicitly specify timing characteristics of data and function components of an object.

To facilitate TMO-based design of real-time systems in the most cost-effective manner, it is essential to provide execution support mechanisms in well-established commercial software/hardware platforms compliant with industry standards. In this paper, we present an implementation model for TMO support mechanisms in CORBA-compliant commercial-off-the-self (COTS) platforms. We first introduce a natural and simple mapping between TMO's and CORBA objects. Then, we identify the services to be provided by the TMO support subsystem and an efficient way these services should be implemented. The rest of the paper discusses an implementation of the proposed model realized on top of the Windows NT operating system and the Orbix object request).

Keywords: Real-time Systems, CORBA, Time-Triggered Message-Triggered Objects, Distributed Systems, COTS.

1. Introduction

The reliable design and implementation of emerging highly complex distributed real-time applications require the use of the state of the art techniques in systems and

software engineering. The effectiveness of object-oriented analysis and design methodologies in the development of robust and highly-maintainable computer applications [Boo94] has been amply demonstrated in the past decade. However, conventional object structuring techniques cannot be efficiently utilized in developing real-time applications mainly because these techniques do not explicitly address the key characteristics of real-time systems, i.e., timing requirements.

Recently, there have been several efforts [Att91, Ish90, Tak92, Kim94a] for extending conventional objects with mechanisms for specifying the timing characteristics. The Time-Triggered Message-Triggered Object (TMO) structuring, formulated in recent years [Kim94a, Kim97], is in our view the most promising and natural extension of the conventional object-oriented design and implementation techniques with the following important characteristics: (a) in addition to traditional service methods (methods called by client objects), each TMO may also have *time-triggered methods* (activated when the real-time clock reaches specific points in time); (b) each method of a TMO is associated with a completion deadline, and (c) data elements of a TMO have a specific validation period beyond which the data value is invalid and should not be provided to the application. Several experiments on the TMO structuring, in a variety of applications ranging from military applications to factory automations [Kim96, Kim97], validated the proposition that the TMO-based real-time system design offers a rigorous way to develop complex real-time systems in easily understandable forms. To facilitate TMO-based design of real-time systems in the most cost-effective manner, it is essential to provide execution support mechanisms in well-established commercial software/hardware platforms compliant with industry standards. Two recent advances in commercial software that have motivated our development of the TMO support facilities are: (i) recently developed multi-threaded operating systems, and (ii) the advent of CORBA standards for distributed objects.

Recent advances in the operating system technologies such as the thread-level concurrency and real-time thread constructs are demonstrated in some commercial-of-the-shelf (COTS) operating systems, e.g., Solaris [Kle96] and Windows NT [Ric97]. Real-time threads allow the application designer to manage (to some degree) the timing aspects of the actions to be taken by the applications.

The CORBA standards [OMG96] have emerged as a promising set of base for enabling development of heterogeneous distributed object-oriented systems. The CORBA architecture provides a high-level location-transparent language-independent software bus through which a client object can call the methods of another object (remote or local) without any knowledge of either the location of the server object or the way the server object is implemented. If TMO's can be realized in CORBA-compliant forms, then CORBA's attractive location-transparent inter-object communication facility can support inter-TMO interactions and thus interactions among a network of TMO's can be designed by use of simple efficient API's [Sho97a, Sho97b].

This paper presents an implementation model for TMO supports in CORBA-compliant COTS software/hardware platforms. A brief background discussion is given in Section 2 on the TMO structuring, the CORBA-compliant Object Request Broker (the middleware facilitating inter-object communications), and the essential features of COTS operating systems used for supporting the proposed implementation model. Mapping TMO's into CORBA objects will be discussed in Section 3. Section 4 presents a detailed discussion on the proposed implementation model. Section 5 presents our prototype implementation of the model on the Windows NT platform. The lessons learned and the conclusion are provided in Section 6.

2. Backgrounds

2.1 The TMO structuring scheme

In recent years, the *Time-triggered Message-triggered Object (TMO) model*, formerly called the RTO.k object, for cost-effective development of *hard-real-time* systems was established with a concrete syntactic structure and execution semantics [Kim94a, Kim96]. The basic structure of the TMO is shown in Figure 1. It is an extension of the conventional object model(s) in three major ways:

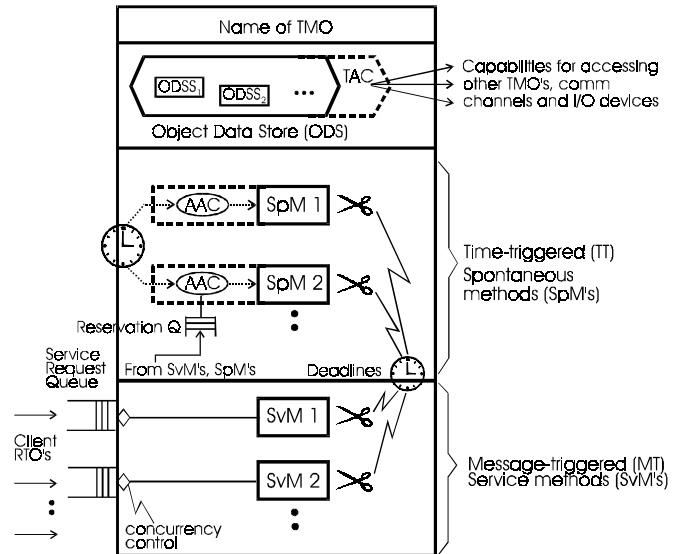


Figure 1. The TMO structure (Adapted from [Kim97])

- (a) *Spontaneous method*: The TMO contains a new type 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.

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

```
"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"}.
```

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 makes reservations for future executions of a specific SpM. The AAC that specifies candidate

triggering times rather than actual triggering times starts with a declaration "if-demanded".

(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 deadline is imposed for each output action and completion of a method of a TMO.

Extensions (a) and (b) are unique to the TMO model in comparison with other proposed object extensions [Att91, Ish90, Tak92]. Client methods (SpM's or SvM's) may request the service 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 *deadline 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 deadline 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 deadlines.

2.2 The Common Object Request Broker Architecture (CORBA)

The underlying philosophy of CORBA is to provide a common architectural framework across heterogeneous hardware platforms, operating systems, and inter-node communication protocols. The core of the CORBA architecture is the Object Request Broker (ORB), a mechanism that facilitates transparency of object location, activation, and communication.

The function of the ORB is to deliver requests from clients to server objects and return output values (if any) back to the clients. Clients do not need to know where in the network server objects reside, how they communicate, or how they are implemented. This implementation transparency is achieved by separating object interfaces from object implementations. Object interfaces (defining the provided data elements and methods that can be seen by client objects) are defined using an "Interface Definition Language (IDL)". The IDL compiler translates object interfaces into detailed language-dependent object definitions and produces IDL stubs and IDL skeletons to be used in client and server sides, respectively.

Figure 2 presents the CORBA support modules to be linked with both client and server objects. An important characteristic here is that the interface created from a server object need not include all the data members and methods of the server object. Only the methods and data elements to be seen by the client objects should be included in the corresponding interfaces. Moreover, as implicitly shown in Figure 2, a request from an object to a remote server object must be passed to the ORB residing in the client host. The request will then be sent to the ORB in the server host. The server ORB then locates the requested object and initiates the execution of the requested methods. When the execution of the requested method is completed, the result will be transferred to the client object through the server ORB and the client ORB. It is important to note that the client object issuing the request does not have any control on the timing of the series of the actions to be taken for the execution of the remote methods. This issue will be discussed in Section 4, where the implementation of a TMO object using the CORBA infrastructure is discussed.

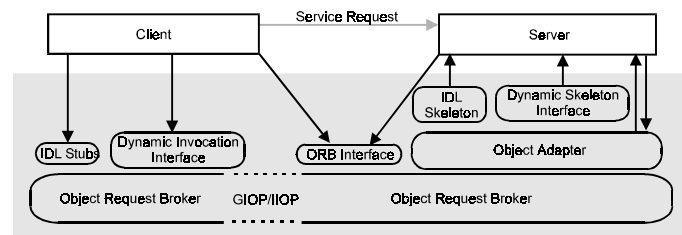


Figure 2. Standard CORBA Architecture

2.3 Essential Features of A Candidate COTS Operating System

The COTS operating system on which an efficient TMO support management is to be built, should provide mechanisms for the following capabilities:

- *Lightweight application-level concurrency unit:* To impose a minimal overhead for switching among different “units of execution” within the application, lightweight application-level concurrency units should be supported by the operating systems. This, however, may be provided in different forms. For example, Solaris offers library-implemented application threads that are partially transparent to the kernel and are managed by the thread-management library. Windows NT took a step forward and introduced kernel-supported thread constructs that can be created by a traditional process. On the other hand, more specialized operating systems such as VxWorks [Kle97] consider a task as a single-threaded unit of execution. A task with a very small context-switch overhead usually has its own private stack while it is permitted to access the entire memory in a well-coordinated manner.

- *High-Precision Timer Interrupt:* The operating system should also allow the application to define an interval timer which can create an event (such as generating a signal, or reactivating a specific process) when it expires. Since all the time-triggered actions in our implementation model will be supported using the timer, the timing of the 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 very small bound. Operating systems such as Solaris and VxWorks possess a POSIX-compliant timer manager. Windows NT also provides the timer capability with its “waitable timer” mechanism [Ric96]. The granularity of the clock supporting the timer, which differs in different operating systems, is a major factor to consider in selecting a platform for a specific application domain.

- *Real-time Threads:* Here, the term “real-time thread” refers to 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 execution can be interrupted only by an explicit command from the kernel, and (iii) when a system service call is issued inside a real-time thread, the system call inherits the real-time characteristics of its issuer thread. These characteristics guarantee timely initiations of the actions to be taken by a real-time thread. However, it should be noted that a majority of the COTS operating systems do not possess the third characteristic and this is one of several reasons for the difficulty of employing those operating systems into hard real-time systems.

- *Efficient control of support processes:* The COTS operating systems designed for typical business data processing environments create various long-life background support processes (often called daemons) for

performing day-to-day management and housekeeping activities. Since these processes, which are transparent to the application designer, steal the resources from the applications to carry out their activities, their actions at unpredictable times may make timely executions of application activities difficult, if not impossible. To guarantee real-time characteristics of the application, the operating system should appropriately control the execution of the support processes, e.g., allocate specific periods of time for their actions. This issue will be further discussed in Section 4.

3. Mapping TMO’s into CORBA Objects

One major step toward realizing the potentials of the TMO structuring scheme is to develop easy-to-use and developer-friendly application-programming interfaces. The approach adopted here is to map each TMO into a single CORBA object. In this section, we discuss a simple mapping approach called the S-mapping that does not require extension of the CORBA IDL and yet covers a major portion of TMO applications. Under S-mapping, externally-seen elements of a TMO, i.e., its service methods (SvM’s), are defined in the interface, while data elements and spontaneous methods (SpM’s) are defined only in the implementation of the CORBA object. Figure 3 illustrates the mapping of a simple TMO (a TMO with a single ODS segment, one SvM, and one SpM) in Figure 3 (a) into a CORBA object in Figure 3 (b).

Basic rules for the S-mapping are the followings:

- A TMO is mapped into a single application-level CORBA object.
- Each SvM is considered as an operation to be defined in the interface definition so that the client objects may see and call the SvM.
- Each SpM should be defined as a private method of the CORBA object and it should not be included in the object interface.
- Each ODS segment (ODSS) will be viewed as a contained object of the CORBA object and it should not be included in the object interface.

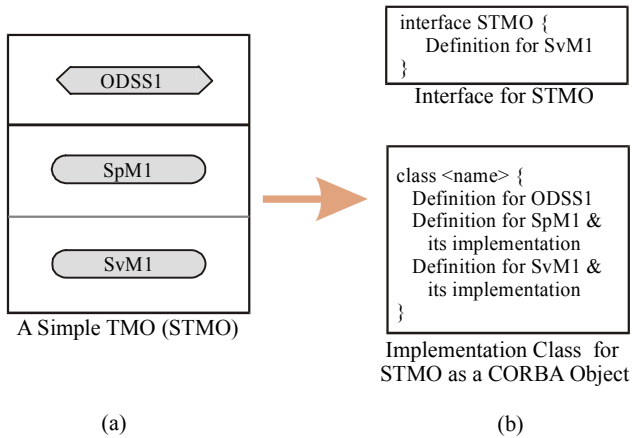


Figure 3. S-Mapping of a TMO into a CORBA Object

This natural mapping provides several major advantages. First, an application designed as a network of TMO's, can easily be implemented in a CORBA-compliant manner. Secondly, a client of a TMO server views the server exactly as a regular CORBA object. The mapping does not depend on any implementation-dependent aspects that are left unspecified in the CORBA specification. In other words, TMO's can interact with the aid of any available CORBA-compliant ORB.

To further simplify the implementation of CORBA-compliant TMO's, common functions, such as "registering TMO methods with the support manager" and "timely association of threads to TMO methods", are isolated from the application-specific code to be developed by the application developer and are localized in a base class named TMOBaseClass. A CORBA-compliant TMO is an instance of a class derived from the TMOBaseClass class.

A CORBA-compliant TMO may also be derived from some library classes (such as CORBA::Object). However, since the details of the implementation class hierarchy may differ among the ORB vendors' offerings, the complete class hierarchy is not discussed here. Figure 4 depicts the class hierarchy for a CORBA-compliant TMO.

TMOBaseClass is a generic class that provides the following functionalities:

- *Registering SpM's*: TMOBaseClass offers a mechanism for the TMO to register its SpM's with the underlying TMO support Manager. An SpM is registered by passing the TMO Support Manager parameters such as the method name, the object data store segments (ODSS's) to be accessed, and the autonomous activation conditions (AAC's). When an SpM is registered, it is assigned a thread with suspended mode. The TMO support manager will obey the AAC's to choose the times for reactivating the associated thread.

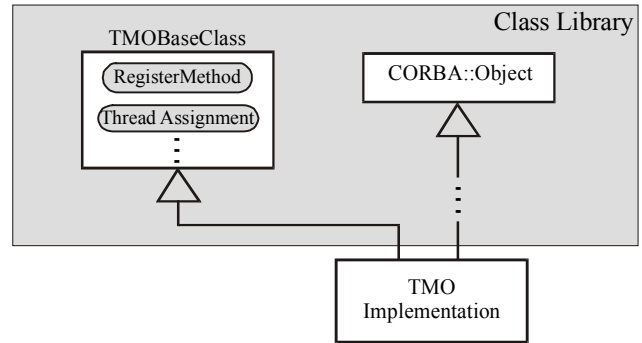


Figure 4. Class Hierarchy for a CORBA-Compliant TMO

- *Registering SvM's*: A TMO can register its SvM's through the mechanism provided by the TMOBaseClass. By registering an SvM, parameters such as the name of the method, its execution deadline, the ODSS's accessed by the method, and the concurrency-allowance flag (indicating that if multiple concurrent executions of the method are allowed) are passed to the TMO support manager. Assigning a thread to a specific execution of a SvM is done when a call to the SvM arrives at the server ORB. When a call to a specific SvM has arrived at the server side, a thread will be assigned to the SvM (in an implementation-dependant way) and then the identifier of the created thread will be passed to the TMO support manager which in turn schedules the execution of the call appropriately by obeying the BCC and other concurrency constraints.
- *Registering ODSS's*: Each ODSS must be registered with the TMO support manager to facilitate access control. When an ODSS is registered, it is assigned a unique identification and the TMO support manager must lock and unlock each ODSS appropriately as methods attempt to use the ODSS.
- *Method completion*: When the execution of a method (SpM or SvM) is completed, the TMO support manager should be notified so that the necessary housekeeping actions (such as terminating the thread if the completed method is of SvM type) may be taken.

It is important to note that TMOBaseClass encapsulates the basic TMO management functions in a single base class such that the designer of the application TMO's is not preoccupied with the details of basic TMO management issues such as registering methods and assigning methods to kernel threads. This, along with the natural mapping of a TMO into a single CORBA object, should greatly simplify TMO-based design and implementation of large-scale applications.

4. TMO Support Implementation in COTS multi-threaded operating systems

In this section, essential elements and implementation characteristics of a TMO support management subsystem to be realized on top of a multi-threaded operating system (with the essential features listed in Section 2.3) and a CORBA-compliant ORB will be discussed. Such a subsystem should support timely executions of the registered methods of application TMO's as well as timely maintenance of data elements of these TMO's using capabilities offered by the underlying operating system and ORB. The main goal here is to present a generic implementation model which does not depend on any specific ORB implementation or specialized features of any operating systems. The following aspects of the TMO support management subsystem are discussed in this section: (i) Timely execution of the TMO support functions; (ii) various types of threads supporting the timely executions of the application TMO's; (iii) the mechanisms for activation and execution of time-triggered and message-triggered methods; and (iv) the execution cycle of the TMO support management subsystem.

Figure 5 shows an abstract architecture of the TMO support management subsystem and its internal thread structure. The main functions of the TMO support mechanisms are carried out by the TMO Support Management Thread (TMOSMT) which is a real-time thread (in a generic term). The TMOSMT is periodically activated by a high-precision timer interrupt and manages the orderly executions of two type of threads: (i) application threads which are assigned to the execution of the application TMO's; and (ii) the system threads which are assigned to support system activities such as timely interconnections among TMO's.

4.1 Timely Execution of the TMO Support Functions

The TMO Support Manager Thread (TMOSMT) performs various support activities such as identifying the TMO methods to be executed in the near future, and scheduling ready-to-run threads such that the execution deadlines of the corresponding

TMO methods are met. Due to the nature of the support functions, it is more cost-effective to execute TMOSMT in a time-triggered mode (i.e., it should be periodically activated by an external timer). We adopted the approach of using a high-precision timer interrupt (which is available in a majority of new-generation operating systems) to guarantee timely activation and execution of TMOSMT. Since other ongoing activities of the host computers must not interfere with the orderly execution of TMOSMT, TMOSMT should be implemented as a real-time thread that releases the execution resource only voluntarily when its required activities are completed.

TMOSMT consists of two major components: "TMO Method Reservation Manager" and "Thread-Level Scheduler". The TMO Method Reservation Manager component periodically examines the registered SpM's and identifies the methods to be executed in the near future. The identified methods are placed in the Method Reservation Queue for further analysis. On the other hand, the Thread-Level Scheduler component is responsible for (i) activating the TMO method-execution threads associated with the methods in the Method Reservation Queue, and (ii) scheduling both the activated TMO method-execution threads and the system threads. This two-level scheduling of the registered methods and their associated threads [Kim96] equips the TMO Support Management subsystem with the look-head capabilities and thus significantly reduces the probability of failing to schedule a thread during a specific allowed time-period. Future details are provided in the next section.

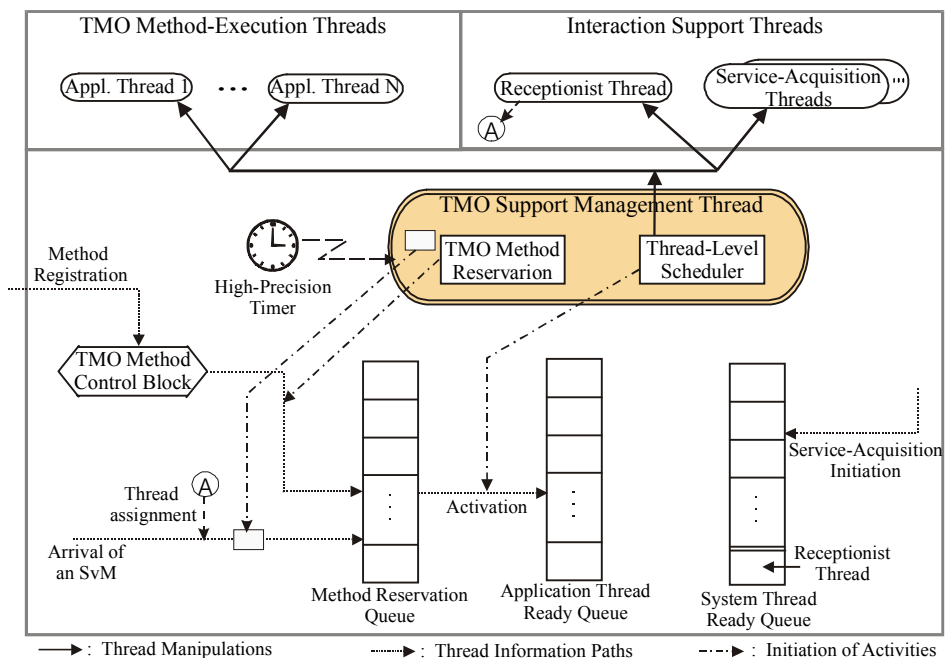


Figure 5. The Internal Structure of the TMO Support Manager

4.2 Mechanisms for Activation and Execution of SpM's and SvM's

Both SpM's and SvM's are registered by the application, however, TMOSMT handles it differently depending upon the type of the methods to be registered. The main difference between the registration of a SpM and that of a SvM is the way the method is assigned to a thread. Basically, an SpM is assigned to a thread permanently when it is registered, while an SvM is temporarily assigned to a thread each time it is called. A thread associated with an SvM is freed when the SvM execution is completed. The main reason for not assigning an SvM to a thread permanently is the possibility of concurrent activations of an SvM by multiple clients.

As shown in Figure 5, when an SpM is registered, its execution parameters, such as the autonomous activation conditions, are stored in the "TMO Method Control Block". During its first execution cycle after the registration of an SpM, TMOSMT assigns a thread to the method. The associated thread is bound to the method throughout the life of the application. When an SpM selected from the Method Reservation Queue is to be activated, its associated thread is placed in the Application Thread Ready Queue of which the contained threads are scheduled by the Thread-Level Scheduler. Upon completion of the method execution, the thread is suspended.

As mentioned earlier, the duration of an assignment of a thread to an SvM is different from that of an assignment to an SpM. The following approach is taken for managing SvM's:

- The system thread called the receptionist thread is responsible for receiving all incoming service-request method calls and assigning a TMO method-execution thread to each method call.
- When TMOSMT receives a request along with the associated thread, it parses the request, identifies the method being called, and obtains the SvM information from the TMO Method Control Block.

4.3 System Threads

System Threads include the following thread types:

- *Receptionist Thread*: Upon startup, a CORBA server should first initialize the server and create declared objects. It should then inform the local ORB of its readiness to accept service-requests from the clients. This is done by a call for the ORB service

"CORBA::BOA::impl_is_ready()". This ORB service function blocks until an event (usually the arrival of a service-request) occurs. It then handles the event and reblocks itself waiting for the next event. It exits only if its specified time expires, an exception occurs, or the server is deactivated by a call for the ORB service "CORBA::BOA::deactivate_impl()". In order to allow the CORBA server to carry out other functions concurrently with the honoring of service-requests from the client objects, it is imperative to dedicate a thread with the responsibility of executing **CORBA::BOA::impl_is_ready()**, thereby devoting itself to the task of receiving the service-requests. This special thread, called the receptionist thread here, must periodically become active to handle incoming CORBA service-requests. The Receptionist thread registers itself with the TMO Support Management Subsystem and it is periodically activated by the TMO Support Subsystem.

- *Service-Acquisition Threads*: In order to allow clients to make non-blocking calls for services from remote objects, we adopted the approach of dedicating a thread for the chore of initiating a method-call on behalf of a client object. Such a thread is called a service acquisition thread. As discussed in Section 6, almost all commercial CORBA implementations are based on non-real-time operating systems in which there are no integrated scheduling of real-time computations and I/O activities. This makes it difficult to ensure timely transmissions of service-requests from the client objects to servers via commercial ORB's. Use of dedicated threads to transmit service-requests significantly improves the situation because these threads are scheduled directly by the TMO Support Management Subsystem in specific periods of time during which other threads are not allowed to be scheduled. This greatly helps limiting the worst-case delay from the initiation of a service-request to its arrival at the server host.

4.4 Execution Cycle for the TMO Management Subsystem

In order to achieve highly predictable timing behavior, various threads in a cyclic fashion. As shown in Figure 6, an execution cycle consists of three subcycles:

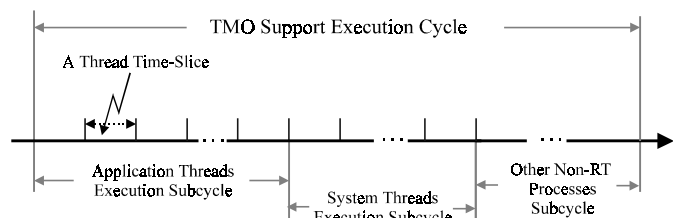


Figure 6. A TMO Support Execution Cycle

- *Application threads execution subcycle:* Comparative evaluation of various low-level scheduling policies for TMO method-execution threads is beyond the scope of this paper. Here we assume that a time-sliced scheduling approach is employed which reflects parameters such as the action completion deadlines in selecting the thread to be executed during each application-time-slice.
- *System threads execution subcycle:* Service-Acquisition threads and the Receptionist thread will be exclusively scheduled during this subcycle. The duration of a time-slice for system threads may be different from that for application threads.
- *Subcycle dedicated to execution of other processes:* Although our focus in this paper is a TMO implementation model in a COTS environment, the presence of other non-real-time processes in typical system configurations cannot be ignored. There are typically some background housekeeping activities that must occur to guarantee orderly operation of various peripheral devices. The execution of these processes should not interfere with the timely execution of TMO's. Therefore, it seems prudent to use a specific portion of each execution-cycle for execution of these supporting processes.

System parameters such as the duration of each subcycle, the duration of an application time-slice, the duration of a system thread time-slice can be chosen by reflecting the application and environment characteristics, in particular, the time-granule provided by the underlying operating system and hardware platform.

5. Prototype Implementation of TMO Support on Windows NT

The TMO support model discussed in Section 4 can be easily implemented on any operating system with the essential features identified in Section 2. In this section, a realization of the TMO support subsystem on top of the Windows NT [Ric97] operating system and the CORBA-compliant ORB Orbix [Orb95] will be discussed. Real-time thread and high-precision timer constructs are the centerpieces of the TMO Support Management Subsystem. Their realizations on the Windows NT environment are discussed first.

5.1 Enforcing the periodic execution of the TMOSM thread under Windows NT

Windows NT is a preemptive multi-threaded operating system 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-slicing scheduling policy and supports eight discrete thread priority levels.

Preemption is based solely on the thread's priority. Moreover, threads with the same priority-level run in the round-robin fashion with a time-slice equal to 25 milliseconds. An important exception is that a thread with the highest priority-level (called `THREAD_PRIORITY_TIME_CRITICAL`) does not share the execution resources with other threads and releases the resources either when it's execution is completed or it voluntarily suspends itself to allow other threads to be executed. It is clear that the NT's default scheduling with a relatively large time-slice (25 milliseconds) is not attractive for use in executing TMO-structured applications.

Because of the demanding timing requirements of TMOSMT, it is useful to implement it as an NT thread with the highest priority-level. This guarantees that the activities of TMOSMT will not be interrupted by other threads. However, it is the responsibility of TMOSMT to select a thread (application or system) to be executed based on the characteristics of the associated TMO methods. After selecting such a thread, TMOSMT suspends all other threads (application or system) and itself such that the only ready thread (in the domain of the TMO application) is the selected application/system thread. To guarantee that the selected thread will not be interrupted by other activities in the host computer during its time-slice, the priority-levels of both application and system threads are chosen to be the second highest. This ensures that other supporting activities will not preempt the activities of the selected application/system thread which will be executed until its time-slice expires at which time TMOSMT is reactivated to carry out its responsibilities, including selecting a new application/system thread to be executed next. The time-slice of an application/system thread should preferably be much shorter than the default NT time-slice. The duration of this time-slice is a parameter that can be adjusted by the application designer and is set to 10 milliseconds in the current prototype implementation. Therefore, TMOSMT disables the NT scheduling and acts as the scheduler of TMO-structured applications as well as TMO support threads.

The TMO implementation model requires that the TMOSMT be periodically activated by a high-precision timer. Windows NT has several timer components. The most accurate timer with the smallest time-granule is the "Waitable Timer" [Ric97] construct. The time-granule of the Waitable Timer construct can be set as small as one

millisecond, which is quite sufficient for a majority of target real-time applications for the COTS platforms.

5.2 Timely Execution of Remote Object Calls under Orbix

In this subsection, we will discuss the anatomy of a remote method call (a SvM call) using Orbix.

As discussed earlier, when a remote method call arrives at a multi-threaded CORBA server, a thread is created to execute the method and return the results (if any) to the ORB which will in turn pass the results on to the client object. In Orbix, concurrent multiple executions of a method can be facilitated by use of the “filter” approach [Orb95]. A filter is a code segment that is executed before any method called by a remote client. Actually a filter is executed while the method-call is in the buffer. The filter we designed creates a thread, sets it to execute the called method and return the method results to the ORB, and then puts it in the suspended mode for activation later by TMOSMT.

Figure 7 depicts the sequence of operations performed in servicing a remote method call. When the filter creates a thread for executing the called method (action number 3 in the diagram), it passes the thread identification and the associated “request” to the TMO support system and then suspends the created thread (action number 4 in the diagram). The TMO support subsystem then parses the request and identifies the names of the object and the method that is designated in the arrived method call. The TMO support subsystem will then activate it at an appropriate time (action number 5 in the diagram).

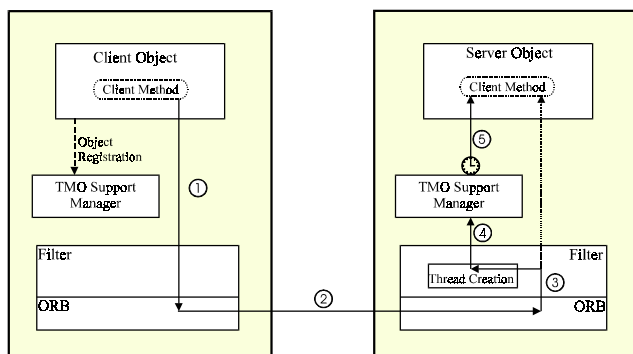


Figure 7. Anatomy of a remote operation in TMO support subsystem using Orbix

The “Thread Creation” component is CORBA-implementation specific. When a CORBA-compliant ORB is selected for employment, this component can be coded and set as a library module such that the TMO

application developer can call it in the filter code segment without any knowledge on how it is implemented.

6. Conclusions

In this paper, we proposed an implementation model for the time-triggered message-triggered object (TMO) support mechanisms that is applicable to CORBA-compliant COTS platforms. The main objective of this research was to integrate the high-level inter-object communication abstraction (provided effectively by CORBA-compliant ORB’s) with an effective TMO structuring of complex real-time systems (a unique characteristics of the TMO structuring approach). We believe that the resulting CORBA-compliant TMO structuring and execution facilities, realized on top of new-generation COTS operating systems, can lead to significant reduction of the development and maintenance costs of large-scale real-time systems and to increases in the overall robustness and dependability of application systems.

A simple and natural mapping that maps a TMO into a single CORBA object was introduced and it greatly simplifies the TMO CORBA-compliant structuring in a major portion of real-time applications. To relieve the application designer from handling common features of the TMO implementation, such as registering TMO methods and assigning operating system threads to the TMO methods, these features are encapsulated in a base class (denoted as the TMOBaseClass). Each TMO object inherits from the TMOBaseClass. An effective mapping for more complicated cases may require an extension of the current IDL and this is a meaningful topic for future research.

The proposed implementation model for the TMO support mechanism can be easily implemented on operating systems with features discussed in Section 2.3. To experimentally validate this proposition, we developed a prototype implementation of the proposed model on top of Windows NT. Our experiments indicated that the implemented CORBA-compliant TMO support mechanisms can support developing real-time applications with a time-granule as small as several milliseconds.

7. References

- [Att91] Attoui, A. "An Object Oriented Model for Parallel and Reactive Systems", *Proc. IEEE CS 12th Real-Time Systems Symp.*, 1991, pp. 84-93.
- [Boo94] Booch, G., *Object Oriented Analysis and Design with Applications*, Redwood City, CA: Benjamin/Cummings Publishing, 2nd ed., 1994.

- [Ish92] Ishikawa, Y., Tokuda, H., and Mercer, C. W., "An Object-Oriented Real-Time Programming Language", *IEEE Computer*, October 1992, pp. 66-73.
- [Kim94a] 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.
- [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.
- [Kim97] Kim, K. H., "Object Structures for Real-Time Systems and Simulators", *IEEE Computer*, Vol. 30, No. 8, August 1997, pp. 62-70.
- [Kle96] Kleiman, S., et al, *Programming with Threads*, Sunsoft Press, 1996.
- [Kle97] Klein, E., "RTOS Design: How is Your Application Affected?", *West Embedded Systems Conference*, San Jose, CA, Sept. 1997.
- [OMG95] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, Revision 2.0, 1995.
- [OMG96] OMG Real-Time Interest Group, "Real-Time CORBA Issue 1.0", Working Document, OMG, 1996.
- [Orb95] *Orbix Programming Guide*, IONA Technologies, Dublin, 1995.
- [Ric97] Richter J., *Advanced Windows*, 3rd Edition, Microsoft Press, 1997.
- [Sho97] Shokri, E., et al., "An Approach for Adaptive Fault Tolerance in Object-Oriented Open Distributed Systems", *Proc. 3rd Workshop on Object-Oriented Real-Time Dependable Systems*, Newport Beach, CA, February 1997, pp. 298-305.
- [Tak92] Takashio, K., and Tokoro, M., "DROL: An Object-Oriented Programming Language for Distributed Real-Time Systems", *Proc. OOPSLA*, 1992, pp. 276-294.