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

Eltefaat Shokri* and Patrick Crane
SoHaR Inc.

Kane Kim
University of California, Irvine

Abstract

Reliable design and implementation of emerging highly complex real-time applications require use of state-of-the-art techniques in systems and software engineering. 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 allow 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. Two recent advances in commercial software that have motivated our development of the TMO support facilities reported here are: (i) recent multi-threaded operating systems and (ii) the CORBA standards for distributed object-oriented systems.

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 the implementation of the proposed model realized on top of the Windows NT operating system and the Orbix object request broker (a commercial implementation of the standard CORBA).

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

*: Contact Author:
Eltefaat Shokri
shokri@sohar.com, (213)653-4717x120
SoHaR Inc.,
8421 Wilshire Blvd., Suite 201
Beverly Hills, CA, 90211-3204
1. Introduction

The reliable design and implementation of emerging highly complex distributed real-time applications require employing state of the art techniques in systems engineering and software engineering methodologies. Recent experience has demonstrated the effectiveness of object-oriented analysis and design methodologies in the development of robust and highly-maintainable computer applications [Boo94]. 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 the concept of conventional objects in the direction of viewing timing characteristics as inescapable elements of an object. The Time-Triggered Message-Triggered Object (TMO) structuring, introduced in [Kim94a], is the most promising 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 (time-triggered methods should be 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 use of the TMO structuring, in a variety of applications from military applications to factory automations [Kim94b, Kim97], validated the belief that the TMO-based real-time system design offers a rigorous way to develop complex real-time systems. However, we believe that providing support for the TMO-based system design and implementation approach for commercially-available software/hardware platform is a major step toward widely-available real-time object-oriented system design techniques. This one step-forward is now possible mainly because of (i) the emerging new-generation multi-threaded operating systems, and (ii) the advent of CORBA standards for distributed objects.

Recent advances in the operating system technologies, specifically the thread-level concurrency and real-time thread constructs, have become reality by commercially-of-the-shelf (COTS) operating systems. Commercial operating systems such as Solaris [Kle96] and Windows NT [Ric97] have facilitated the timely application-level concurrency by providing supports for real-time threads. Real-time threads allow the application designer to manage (in some degree) the timing behaviors of the actions to be taken by the applications.
The CORBA standards [OMG96] have emerged as a promising set of specifications for development of 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. Employing CORBA’s desirable location-transparent inter-object communication capability for inter-TMO interactions will facilitate simple and efficient API’s for the interaction among a network of TMO’s [Sho97a, Sho97b].

This paper presents an implementation model for TMO supports in CORBA-compliant COTS software/hardware platforms. The paper provides a brief background discussion on the TMO structuring, CORBA-compliant Object Request Broker (the middleware facilitating inter-object communications), and the minimal features of COTS operating systems 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 paper discusses the lessons learned and provides a conclusion in Section 6.

2. Backgrounds

2.1 The TMO structuring scheme

Several years ago, an abstract precursor of the Time-triggered Message-triggered Object (TMO) model for cost-effective development of hard-real-time systems was proposed [Kop90]. In recent years, the TMO, formerly called the RTO.k object, has been formalized to possess a concrete syntactic structure and execution semantics [Kim94a, Kim94b].

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:

(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 which 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 requests 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, called the Basic Concurrency Constraint (BCC), 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 object models [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 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 implementation. Object interfaces (defining the provided data elements and methods which can be seen by server 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, respectfully. Figure 2 presents the modules to be linked with both client and server objects. What is important from our research viewpoint 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 interface.
Moreover, as implicitly shown in Figure 2, a request from an object to a remote server object has to be passed to the ORB residing in the client host. The request then will 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, when the implementation of a TMO object using the CORBA infrastructure is discussed.

Figure 2: Standard CORBA Architecture

2.3 Minimal 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:**
  To impose a minimal overhead for switching among different “units of execution” within the application, a lightweight application-level concurrency should be supported by the operating systems. This, however, may be provided in different forms. For example, Solaris offers library-implemented application threads which are partially transparent to the kernel and are managed by the thread-management library. Windows NT took a step forward and introduced kernel-known thread constructs which can be created by a traditional process. On the other hand, more focused 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.
• **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, they should have very small bounded delays for their activation, as well as initiations of the corresponding actions. Operating systems such as Solaris and VxWorks possess a POSIX-compliant timer management. Windows NT also provides the timer capability with its “waitable timer” concept [Ric96]. However, the granularity of the clock supporting the timer, which differs in different operating systems, is an essential factor in selecting a platform for a specific application domain.

• **Real-time Threads:**
Here, we use the “real-time thread” term as a generic construct which has the following characteristics: (i) it has a guaranteed access to resources within a bounded delay since it becomes ready, (ii) its execution can be interrupted only by an explicit command from the kernel, and (iii) when a system 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 the majority of the COTS operating systems do not satisfy the third characteristic and is one of several reasons for the difficulty of employing these operating systems into hard real-time systems.

• **An efficient control of support processes:**
The COTS operating systems designed for general-purpose application environments normally create various long-life background support processes (normally 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 presence may make timely executions of application activities difficult, if not impossible. To guarantee real-time characteristics of the application, the operating system should not prevent the privileged applications (such as real-time applications) from enforcing a controlled execution of the support processes. One way to have a controlled execution of support processes is to allocate specific periods of time for these processes to be executed by the operating systems. This issue will be further discussed in Section 4.
3. Mapping TMO’s into CORBA Objects

One major step toward materializing the potentials of the TMO structuring scheme is developing easy-to-use and developer-friendly application-programming interfaces. To achieve this important goal, each TMO is mapped into a single CORBA object. Externally-seen elements of a TMO, meaning its SvM methods, will be defined in the interface, while data elements and SpM methods will be defined only in the implementation of the CORBA object. Figure 3 illustrates the mapping of a simple TMO (a TMO with a single ODSS, one SvM method, and one SpM method), Figure 3 (a), into a CORBA object, Figure 3 (b).

Basic rules for the 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 can see and call the SvM methods.
- Each SpM should be defined as a private method of the CORBA object, but it should not be included in the object interface.
- Each ODSS will be viewed as a contained object of the CORBA object and is considered internal to the object meaning that it should not be included in the object interface.

Figure 3: An Example for TMO Mapping into CORBA Objects
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, it complies with the object-oriented philosophy of hiding the internal structure of an object from its clients as much as possible. Moreover, under the proposed mapping a client of a TMO server views the server exactly as a regular CORBA object. Finally, maximum care has been taken to guarantee that the mapping is of generic type and does not depend on any implementation-dependant aspects which are left unspecified in the CORBA specification. In other words, the mapping can be performed in any available CORBA-compliant ORB.

To further simplify the implementation of CORBA-compliant TMO’s, the TMO-support functionalities, such as “registering TMO methods with the support management” and “timely association of threads to TMO methods”, are isolated from the code to be developed by the application developer and are localized in a base class named TMOBaseClass. A TMO-mapped object is in instance of a class derived from the TMOBaseClass base class.

A CORBA server object (or as sometimes called the implementation object) may also inherit from some library classes (such as CORBA::object) depending on whether inheritance or delegation is used for implementing the server object. However, since the details of the implementation class hierarchy may depend on the CORBA implementation, the complete class hierarchy is not discussed here. Figure 4 depicts the class hierarchy for a TMO-mapped CORBA server object.

![Class Hierarchy for a CORBA-Compliant TMO Server](image)

Figure 4: Class Hierarchy for a CORBA-Compliant TMO Server

TMOBaseClass is a generic class which provides supports for the following functionalities:
• **Registering SpM methods:** TMOBaseClass offers a mechanism for the TMO object to register its SpM’s with the underlying TMO support Management. A SpM is registered by passing the TMO Support Management characteristics such as the method name, the object data storages (ODSS’s) to be accessed, and the autonomous activation conditions (AAC’s). When a SpM is registered, it is assigned an initially-suspended thread. The TMO support management will use the AAC’s to initiate the execution of the SpM by reactivating the associated thread.

• **Registering SvM methods:** A TMO object can register its SvM’s through mechanisms provided by the TMOBaseClass. By registering a SvM, the name of the method, its execution deadline, the ODSS’s accessed by the method, and the concurrency-allowance flag (indicating that if concurrent execution of the method is allowed) are passed to the TMO support management. However, since various CORBA implementations provide different mechanisms for concurrent execution of method calls, assigning a thread to a specific execution of a SvM is assumed to be done (in a implementation-specific way) when a call to the SvM is arrived 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 and the unique identifier of the associated thread will be passed to the TMO support management to schedule the execution of the call in a timely manner.

• **Registering ODSS’s:** Each ODSS has to be registered with the TMO support management to facilitate access authorization control. When an ODSS is registered, it is assigned a unique (node-wide) identification such that the TMO support management can guarantee that illegal accesses to the ODSS will be prevented.

• **Method completion:** When the execution of a method (SpM or SvM) is completed, the TMO support management should be notified so that the necessary housekeeping actions (such as terminating the thread if the completed method is of SvM type) are taken.

It is important to note that TMOBaseClass encapsulate the TMO management features in a single base class such that the designer of the application TMO objects is not preoccupied with the details of TMO management issues such as registering methods and assigning methods to kernel-level threads. This, along with the natural mapping of a TMO into a single CORBA object, will 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 minial 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 will be, specifically, presented 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: The Internal Structure of the TMO Support Management Subsystem
Figure 5 shows a general 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 mapped into 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 created by the application TMO objects; and (ii) the system threads which are created to support system activities such as timely interconnections among TMO objects.

4.1 Timely Execution of the TMO Support Functions

The TMO Support Manager Thread (TMOSMT) is to perform various support activities such as indentifying the TMO methods to be executed in a 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 the TMOSMT in a time-triggered mode (i.e., it should be periodically activated by an external timer). We purpose the use of a high-precision timer interrupt (which is available in majority of new generation operating systems) to guarantee a timely activation and execution of TMOSMT. Since other ongoing activities of the host computers must not interfere with the orderly execution of TMOSMT, the TMOSMT should be implemented as a real-time thread which releases the execution resource only voluntarily when its required activities are completed.

TMOSMT consists of two major components: “TMO Method Reservation Manager” and “Thread Manager”. The TMO Method Reservation Manager component periodically examines the registered methods and identifies the methods to be reactivated in a near future. The identified methods are placed in the Method Reservation Queue for further analysis. On the other hand, the Thread Manager component is resposible for (i) reactivating and scheduling the threads associated with the methods in the Method Reservation Queue, and (ii) scheduling system threads which basically support timely interconnection of TMO objects. This two-level scheduling of the registered methods and their associated threads [Kim94b] 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. A detailed discussion on the functions performed by these two components is presented in Section 4.2.

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, the TMO Support Management Subsystem acts on the registration of a method according to 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 mapped into a thread. Basically, a SpM is mapped to a thread permanently when it is registered, while a SvM is temporarily mapped to a thread each time it is called. A thread associated to a SvM is freed when the SvM execution is completed. The main reason for not allocating a SvM to a thread permanently are (i) the possibility of concurrent activations of a SvM by multiple clients, and (ii) the fact that the API for assigning a thread to a method-call is not standardized in the CORBA specification and is, thus, implementation-independent.

As shown in Figure 5, when a SpM is registered, its execution parameters, such as the autonomous activation conditions, are stored in the “TMO Method Control Block”. Moreover, upon the registration of a SpM, the TMO Support Management Subsystem creates a thread for the execution of the method in an application-transparent manner. The associated thread is bound to the method throughout the life of the application. When a SpM selected from the Method Reservation Queue is to be reactivated, its associated thread (inheriting execution characteristics of the method) is placed in the Application Thread Ready Queue for scheduling by the Thread-Level Scheduler. Upon completion of the method execution, the thread is suspended.

As mentioned earlier, the life-cycle of a SvM and its associated thread(s) is different from that of a SpM. Since there is no standard API’s for developing Multithreaded CORBA servers, to keep the proposed TMO implementation model independent of any specific CORBA-implementation, the following approach is taken for managing a SvM:

- The TMO Support Management is not involved in creating a thread and associating it to the SvM method being called.
- The application designer is responsible for creating a thread (in a suspended mode) for a SvM activation (using the specifics of the employed CORBA implementation) and passes the thread identifier and the arrived “request” to the TMO Support Management.
- When the TMO Support Management receives the request and associated thread, it parses the request, identifies the method being called and updates the SvM information in the TMO Control Block. The associated thread is also placed in the Application Thread Ready Queue to be scheduled later by the Thread-Level Scheduler.

### 4.3 Threads Managed by the TMO Support Management Subsystem

As depicted in Figure 5, the TMO Support Management Subsystem manages two types of threads: (i) the threads which are created to execute TMO methods (categorized as the TMO-Mapped Threads Poll), and (ii) the threads which are created to handle communications among TMOs, or more
precisely among CORBA objects (denoted as the System Threads Pool). TMO-mapped threads and their management were briefly discussed in the preceding subsection. Here, we focus on the System Threads Pool which includes the following thread types:

- **Server-Connection 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 by calling “CORBA::BOA::impl_is_read()”. CORBA::BOA::impl_is_ready is an event-manager which 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 exists only if either its specified time expires or an exception occurs or the server is deactivated by “CORBA::BOA::deactivate_impl()”. To allow the CORBA server to carry out other functionality concurrently with serving service-requests from the client objects, it is imperative to dedicate a thread with the responsibility of initializing the server and executing CORBA::BOA::impl_is_ready(). This special thread, denoted as the Server-Connection Thread, must periodically become active to handle incoming CORBA service-requests. The Server-Connection thread registers itself with the TMO Support Management Subsystem, such that it can be reactivated by the TMO Support Subsystem in a timely manner.

- **Service-Request Threads**: In order to allow concurrency in both client and server processes, we adopted the approach of dedicating a thread for the chore of initiating a method-call (denoted as service-requests here) by a client object. This can be done in two ways: (i) dedicating a thread-poll with a constant number of threads for service-request initiations, or (ii) creating a thread for each service-call and deleting it after the completion of the service-call. As discussed in Section 6, almost all commercial implementations of the CORBA standard are implemented on top of non-real-time operating systems in which there are no integrated real-time scheduling for computations and I/O activities. This makes it very difficult for commercial ORB’s to provide timely client-side initiations of service-calls. Associating specific threads (Service-Request Threads) to initiate service-requests will significantly reduce the impact of the absence of an integrated real-time scheduling because, as will be discussed in the next subsection, these threads are scheduled directly by the TMO Support Management Subsystem in specific periods of time during which other threads (including TMO application threads) are not allowed to be scheduled. This greatly helps limiting the worst-case completion time of sending a service-request to the server host.

### 4.4 Execution Cycle for the TMO Management Subsystem

TMO Support Management Subsystem manages various threads discussed earlier in a series of execution cycles. As shown in Figure 6, an execution cycle consists of three subcycles:
• **Application threads execution subcycle**: During this subcycle, TMO Support Management Subsystem schedules solely application threads. A discussion on the suitable scheduling policies for TMO-mapped application threads is beyond the scope of this paper. Here we assume that a time-slicing scheduling is employed which considers parameters such as the completion deadlines of the threads for selecting the thread to be executed during each application-time-slice.

• **System threads execution subcycle**: System threads consisting of Service-Request threads and the Service-Connection thread will be exclusively scheduled during this subcycle. Since there can be several Service-Request threads ready to be scheduled a suitable time-slicing scheduling policy is needed for system threads as well. However, the duration of time-slices for system threads may be different from those of application threads.

• **Subcycle dedicated for execution of other processes**: Since our focus in this paper is a TMO implementation model in a COTS environment, it is unrealistic to assume that the TMO-based application (which is of our concern) is the only application process running in an host computer. There may be other, non-real-time processes to be executed in the same node. Even when the system designer can restrict the participant nodes (nodes executing the TMO network) only for single application, there are some background house-keeping activities which has to be executed to guarantee orderly execution of the underlying COTS operating system. The execution of these processes should not interfere with the timely execution of the TMO objects. Therefore, we propose using a specific portion of each execution-cycle for execution of these supporting processes. During this subcycle, none of the thread directly managed by the TMO Support Management Subsystem are allowed to be executed. Moreover, the TMO Support Subsystem should guarantee that these supporting processes will not be activated outside of the dedicated subcycle.

Quantitative measures such as the duration of each subcycle, application time-slice duration, the system thread time-slice length depend on the application needs, the frequency of inter-object communications, and, most importantly, the time-granularity provided by the underlying operating
systems and hardware platform. The TMO Support Management Subsystem should provide facilities for adjusting these parameters based on application needs and platform capabilities.

5. Prototype Implementation of TMO Support in Windows NT

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

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

Windows NT is a preemptive multi-threaded operating system which 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 shown in Table 1.

<table>
<thead>
<tr>
<th>Priority-Level</th>
<th>Typical Uses</th>
</tr>
</thead>
<tbody>
<tr>
<td>THREAD_PRIORITY_TIME_CRITICAL</td>
<td>Real-time activities and devise drivers</td>
</tr>
<tr>
<td>THREAD_PRIORITY_HIGHEST</td>
<td></td>
</tr>
<tr>
<td>THREAD_PRIORITY_ABOVE_NORMAL</td>
<td>Kernel threads and normal applications</td>
</tr>
<tr>
<td>THREAD_PRIORITY_NORMAL</td>
<td></td>
</tr>
<tr>
<td>THREAD_PRIORITY_BELOW_NORMAL</td>
<td></td>
</tr>
<tr>
<td>THREAD_PRIORITY_LOWEST</td>
<td>Applications preempted by others</td>
</tr>
<tr>
<td>THREAD_PRIORITY_ABOVE_IDLE</td>
<td></td>
</tr>
<tr>
<td>THREAD_PRIORITY_IDLE</td>
<td></td>
</tr>
</tbody>
</table>

Preemption is based solely on the thread’s priority. Moreover, threads with the same priority-level run in round-robin fashion with a time-slice equal to 25 milliseconds. An important exception is that a thread with the highest priority-level (i.e., 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) may not be able to provide the demanding timely capabilities required by the TMO support subsystem.

To satisfy demanding timing requirements of the TMOSM thread, it is imperative to implemented it as a NT thread with THREAD_PRIORITY_TIME_CRITICAL priority-level. This guarantees that the activities of TMOSM will not be interrupted by other threads. However, it is the responsibility of TMOSM to select a thread (application or system) to be executed based on the characteristics of the registered TMO-based threads. After selecting such a thread, TMOSM suspends all other threads (application or system) and itself such that the only ready thread (in the scope of the TMO application) is the selected application/system thread. To guarantee that the selected thread will be continuously executed and will not be interrupted by other activities in the host computer, the priority-level of both application and system threads is chosen to be THREAD_PRIORITY_HIGHEST. This secures that kernel threads and other supporting activities will not preempt the activities of the selected application/system thread which will not be executed until TMOSM is reactivated to carry out its responsibilities, including selecting a new application/system thread to be executed next. The time-period during which TMOSM is in the suspended mode and an application/system thread is being executed is denoted as the TMO-enforced time-slice (which is much shorter that the deafult NT time-slice). The duration of TMOSM-enforced time-slice is a parameter which can be adjusted by the application designer and is set to 10 milliseconds in the current prototype implementation. From the above discussion it is clear that TMOSM mainly disables the NT scheduling and acts as the scheduler of the TMO-mapped application threads, as well as the TMO support threads.

The TMO implementation model demands that the TMOSM thread be periodically activated by a high-precision timer. Windows NT has several timer components. The most accurate timer with the smallest time-granularity is the “Waitable Timer” [Ric97] construct. The time-granularity of the Waitable Timer construct can be set as small as one milisecond, which is quite sufficient for the majority of target real-time applications for the COTS platforms. The Waitable Timer construct allows a thread to wait for one or more “objects” (a concept similar to events) and provides easy-to-use API’s for creating and managing timers [Ric97].

5.2 Timely Execution of Remote Object Calls under Orbix

As discussed in Section 3, the CORBA specification does not determine a concurrency model for the multi-threaded CORBA servers, and, thus, the server-side thread management is left to the
implementation of the standard. On the other hand, the TMO support subsystem should interact with the server-side thread-creation mechanism since scheduling of remote operation calls (which are, indeed, activations of SvM’s) must be managed by the TMO support subsystem. In this subsection, we will discuss the anatomy of a remote operation call (a SvM call) using Orbix and will illustrate the interactions between the TMO support subsystem (which is of a generic type and does not depend upon any CORBA implementation) and the implementation-specific service-side thread creation for a remote operation call.

When a remote operation call arrives at a multi-threaded CORBA server, a thread is created (we assume a thread-per-call concurrency model here) and the created thread executes the operation and returns the results (if any) to the ORB to be passed to the client object. In Orbix, this is done by use of the “filter” concept [Orb95]. Orbix allows the application designer to create a thread for an arrived remote operation call inside the filter code (details of this can be found in [Orb95]). To enforce the timely execution of all registered methods (including the called method), the TMO support subsystem must intercept the above sequence of operations.

Figure 7 illustrates the anatomy of a remote operation call and the sequence of operations to be performed for servicing a remote operation call. Under Orbix this can be done by using the filter facility in the following way. When the application creates a thread for a remote operation call (action number 3 in the diagram), it can 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). Upon receiving the thread identification and the associated request, the TMO support subsystem parses the request and identifies the names of the object and the method which is called by the arrived operation call. The TMO support subsystem then updates its internal knowledge on the calling method and will 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-implementatiion specific. However, when the employed CORBA-compliant ORB is selected, this component can be coded and created as a library module, such that the TMO application developer can call it in the filter code without any knowledge on how it is implemented.

6. Conclusions

In this paper, we proposed an implementation model for the time-triggered/message-triggered objects (TMO’s) support mechanisms on 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 the potentials of 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’s, realized on top of new-generation COTS operating systems, significantly reduce the development and maintenance cost of large-scale real-time systems and increase their overall robustness and dependability.

A natural mapping which maps a TMO into a single CORBA object was introduced which greatly simplifies the TMO-based structuring of real-time applications. To relieve the application designer from handling generic features of the TMO implementation, such as registering TMO methods and assigning operating system threads to the TMO methods, are encapsulated into a generic base class (denoted as the TMOBaseClass). Each TMO object inherits from the TMOBaseClass.
The proposed implementation model for the TMO support mechanism is of a generic type and can be easily implemented on operating systems with minimal features (such as providing an easy-to-use API for managing application-level concurrency and a high-precision software timer). To experimentally validate this belief, we developed a prototype implementation of the proposed model on top of Windows NT. Our experiments with the prototype implementation (carried out in the Windows NT environment) showed that CORBA-compliant TMO support mechanisms can support developing real-time applications with a time-granularity greater than one millisecond.

7. References


