A Soft Real-Time TMO Platform - WTMOS -
and Implementation Techniques

J. G. Kim
Hankuk University of Foreign Studies
Seoul, Korea
jgkim@maincc.hufs.ac.kr

M. H. Kim
Konkuk University
Seoul, Korea
mhkim@galaxy.konkuk.ac.kr

B. J. Min
Inchon University
Inchon, Korea
bjmin@lion.inchon.ac.kr

D. B. Im
Sangji Junior College
Wonjoo, Korea
dbim@maincc.hufs.ac.kr

Abstract

The TMO (Time-triggered Message-triggered Object) model is a real-time object model pursuing the timeliness-guaranteed computing paradigm. A TMO contains two types of methods: time-triggered method activated by its internal clock and message-triggered method activated by a request from a client. In this paper, we present the functions and the design techniques of a middleware platform to support execution of TMO's on the Windows environment, named WTMOS (Windows TMO System). The WTMOS platform does not support the concept of design-time-guarantee. But, as a middleware platform, it supports soft real-time system by trying to meet the goal of on-time activation and finishing within deadlines imposed on time-triggered and message-triggered methods in TMO's. It also offers used-defined deadline exception handlers to compensate the lack of timeliness guaranteed design. Besides the above functions, the activation sources of message-triggered methods on WTMOS are expanded so that it can be activated not only by distributed IPC messages but also by Windows system messages. As a result, message-triggered methods now can play the role of Windows message callback functions naturally. This is to add window management facilities and interfaces of Win32 API to the TMO model. Therefore, the TMO model and WTMOS together can be regarded as a powerful development environment for typical soft real-time applications such as multimedia services and virtual reality services.

I. Introduction

Currently, researches on real-time object oriented systems have mostly concentrated on the following two areas:

Design of a real-time object-oriented model that can specify real-time activities precisely and can be realized in the form of generalization of non-real-time computing.

Design and implementation of an execution platform for real-time objects that offers a timeliness guaranteed computing environment.

The TMO (Time-triggered/Message-triggered Object) model formalized by Kim and Kopetz [1] can be used in the applications of hard and soft real-time systems, real-time simulations, and general concurrent systems. To support the design-time guaranteed computing of TMO applications, the DREAM Lab. of UCI developed a timeliness-guaranteed kernel named DREAM micro-kernel [3]. Although the usage of real-time TMO applications on the DREAM kernel has continuously expanded, the DREAM kernel platform has some restrictions in the development of real-time applications requiring window management such as multimedia services. That is because the DREAM kernel is a hard real-time stand-alone micro-kernel that does not offer window management and interface facilities.

Recently, the success of multimedia and virtual reality services on window systems has created the need of real-time embedded system environment to support the execution of applications written in WIN32 API. To realize this environment, several real-time systems have

1 This work has been done as a part of the project entitled “Integrated Object-Oriented Development Environment for Distributed Real-Time Systems” funded by KOSEF of KOREA (Project no. 96-01-07-01-3).
been developed by modifying Windows NT or by loading a real-time subsystem on Windows NT. But most of the current systems support conventional real-time processes, but not real-time objects. The development of these types of systems is time-consuming and requires high cost because the underlying Windows system must be modified. Besides the cost problem, the modification of the Windows system may cause conflicts with existing applications [6].

In this paper, we present a middleware TMO platform named WTMOS (Window TMO System) on Windows 95/NT OS to expand the application areas of the TMO model. To support the full window interface and management facilities, WTMOS as been developed as a middleware by using Win32 API. Consequently, it does not support the concepts of design time guaranteed computing since scheduling and basic timing services are dependent on Windows 95/NT. However, as a soft real-time platform, it supports the timely execution of TMO’s by providing a soft real-time scheduling with deadline, distributed IPC, Windows management and exception handlers for deadline violations. Therefore, it can be regarded as a powerful development environment for typical soft real-time applications such as multimedia and virtual reality services. Actually, we are currently developing a multi-point teleconferencing service using the TMO model on WTMOS [4].

Section 2 introduces related works. Section 3 presents the design and implementation techniques of WTMOS. Section 4 provides the concluding remarks.

2. Related Works

In section 2.1, the TMO model and the design philosophy of the DREAM micro-kernel are briefly introduced. In section 2.2, several existing approaches to extend the Windows system to a real-time system are described.

2.1 The TMO Model

The TMO model was formalized as a timeliness-guaranteed computing model for real-time systems. As a real-time programming paradigm, the TMO model aims the following goals:

- Flexible structure applicable to hard real-time applications as well as general concurrent computing applications,
- Guaranteeing timely service at design-time.

Distinguishing characteristics of the TMO model from conventional object models can be summarized as follows:

- There are two clearly separated groups of methods. One is a spontaneous method (SpM) group each of which is triggered by the clock and can be specified at design time, the other is a conventional service method (SvM) group. An SvM is triggered by messages from clients and can not be determined at design time;
- When an SpM conflicts with an SvM in accessing a part of data, the SpM is given higher priority for the execution. This restriction is called the basic concurrency constraint (BBC) and aims at reducing designer’s efforts;
- In a TMO, a deadline is imposed on each execution of a method and real-time data stored in it become invalid after a certain timing interval.

The timing constraints such as execution period and deadline of an SpM called AAC (Autonomous Activation Condition) are determined at design time and has the following form:

"for t = from 9am to 9:15am every 7 min start-during (t, t+10sec) finish-by t+30sec“.

---

**Figure 1. The Structure of a TMO [adopted from [1]]**
The meaning of the above statement is obvious and the structure of the TMO model is shown in Figure 1.

The design philosophy and functions of the DREAM micro-kernel—a TMO platform—of UCI DREAM Lab. are as follows [3, 5].

- A general-purpose timeliness-guaranteed kernel service model with five-layer structure and the principle of time leasing machine layering;
  - Layer 1 contains basic support functions for high-bandwidth I/O such as LAN interface;
  - Layer 2 contains the kernel thread scheduler which is activated upon expiration of a thread time-slice as well as at some other times;
  - Layer 3 contains basic support functions for low-bandwidth I/O such as serial character I/O;
  - Layer 4 contains the user thread scheduler and other support functions for them, CREW monitors, and multicast channels. The user thread scheduler is activated upon expiration of a thread time-slice as well as some other times;
  - Layer 5 contains TMO applications;
- Modularity and expandability;
- Execution of real-time TMO methods with various activation and synchronization requirements;
- Supporting CREW (Concurrent-Read Exclusive-Write) monitors for shared data;
- Real-time multicast logical (RML) channels;
- The DREAM kernel itself was designed as a group of TMO’s;

2.2 Approaches to Extend the Windows System to a Real-time System

Existing approaches to extend the Windows system to a real-time system are classified to two major classes. One is to modify the Windows system itself to support a real-time threads or processes and the other is to port the Windows sub-system on a real-time micro-kernel.

The first approach depicted in Figure 2-a must modify the HAL (Hardware Abstract Layer) of the Windows system to support real-time threads and processes. In the second approach shown in Figure 2-b, the Windows sub-system supporting the user interface runs with lower priority on a real-time micro-kernel. But these two approaches still have some problems in using the full services of the Windows system as well as in stability and development cost [6]. To overcome these problems, the middleware approach has been used to develop a real-time Windows environment in this paper.

3. Design and Implementation of WTMOS

3.1 Design of WTMOS

3.1.1 Characteristics of WTMOS

WTMOS (Windows Real-Time Object System) is a soft real-time Windows system developed by using the middleware approach without modifying the underlying Windows system. WTMOS has some advantages that one can use the full services of the Windows system and the development cost is relatively low. However, since it used an indirect real-time scheduling scheme in which WTMOS controls scheduling priorities of real-time methods mapped into Windows threads, WTMOS can not be stated to support a hard real-time environment. But it tries to support the on-time invocation and execution of real-time methods by the indirect real-time scheduler with the precision of one millisecond. Also, it supports the TMO programming and uses deadline exception handlers to overcome the lack of design-time guaranteed computing. Considering these characteristics, WTMOS can be regarded as a useful object-oriented platform for soft real-time applications such as multimedia and virtual reality services.

Since WTMOS has been developed as a middleware that uses only threads and IPC related services of the Windows 95/NT OS, it can be ported on any kind of hard real-time micro-kernels if they support basic services.
Therefore, we believe that WTMOS can be extended to support hard real-time applications. As shown in Figure 3, we are currently developing a micro-kernel to replace the Windows system in order to support design-time guaranteed applications in which the Windows interface is not necessary.

The functions in the TMO programming supported by WTMOS can be summarized as follows:

- Real-time object oriented programming with TMO’s.
- Time-triggered methods and message-triggered methods of the TMO model are supported. Real-time methods of a TMO are mapped into Windows system threads;
- A TMO: a C++ class with definitions of SpM’s and SvM’s;
- Inheritance and polymorphism;
- On-time activation of SpM’s;
- Earliest finish deadline first priority control;
- User defined deadline exception handler;
- Clock synchronization among distributed WTMOS’s;
- Distributed Data Field IPC;
- An SvM can be activated not only by distributed IPC messages but also by Windows system messages. Therefore, an SvM can be a real-time window interface manager in the WTMOS environment;
- CREW (Concurrent-Read Exclusive-Write) Monitor supported by the inheritance mechanism;

3.1.2 Layered Structure of WTMOS

WTMOS itself has a five-layer structure on the Windows OS as depicted in Figure 4 and services provided by each layer are as follows:

1) Windows system layer
   - Win32 API support
   - Multi-threading
   - Synchronization
   - Multicast platform

2) WTMOS Layer 0:
   - High-Resolution Clock Manager
     - One millisecond resolution (Soft clock);
     - Management of WTMOS system time;
     - Invoking periodic middleware tasks of layer 1;

3) WTMOS Layer 1:
   - WTMT (Watchdog & TMO Management Task)
     - On-time SpM invocation;
     - Checking deadlines of active SpM’s and SvM’s and performing indirect scheduling (reassign priorities of methods according to their deadlines left);
     - Invoking deadline exception handlers;
   - IMMT (Incoming Message Management Task)
     - Fetching incoming messages from Windows interface and distributed multicast platforms and delivering them into buffers of WTMOS layer 2 (WTMOS Service Interface);
   - OMMT (Outgoing Message management Task)
   - CST (Clock Synchronization Task)

4) WTMOS Layer 2: WSI (WTMOS Service Interface)

   Interface functions used by TMO programmers are supported by this layer. Functions are categorized as follows;
● TMO method management;
● Distributed Data Field IPC;
● Windows interface and management for SvM’s

5) WTMOS Layer 3: Object and macro library
● CREW support object library;
● Macros for converting TMO methods into threads;

6) WTMOS Layer 4: TMO applications

3.2 Implementation of WTMOS

3.2.1 High Resolution Timer Manager

The high-resolution timer manager responsible for activating four high-priority periodic system tasks of the WTMOS layer 1 has been implemented as a callback function of the Microsoft supported multimedia timer. The multimedia timer has one millisecond precision and can preempt most of system activities except for some critical hardware handlers.

3.2.2 WTMT (Watchdog & TMO Management Task)

WTMT is a high-priority periodic system task responsible for on-time activation of SpM’s and checks if there are deadline violations in execution of activated SpM’s and SvM’s. Firstly, WTMT activates an SpM whose start time is on. Then it reassigns the priorities of mapped threads (SpM’s and SvM’s) according to the deadlines left. If some deadline violations are found in this step, it invokes the user defined deadline exception handlers.

WTMT uses the EDF (Earliest Deadline First) scheme in periodic reassigning of priorities. To do this dynamic priority control, WTMOS maintains a data structure named MCB (Method Control Block) which contains various information on methods and their corresponding threads. The AAC (Autonomous Activation Condition) information stored in an MCB is the basis of the dynamic priority control of the EDF scheduling. Since this scheduling done by the middleware, the concept of design time guaranteed computing is not supported by WTMOS. To overcome this problem, WTMT activates user-defined deadline exception handlers if there are violations. The contents of AAC and MCB are listed in Table 1 and Table 2 respectively. Table 3 represents operations performed by WTMT briefly.

<table>
<thead>
<tr>
<th>Table 1. AAC Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIME From;</td>
</tr>
<tr>
<td>TIME To;</td>
</tr>
<tr>
<td>TIME Every;</td>
</tr>
<tr>
<td>TIME StartDuring;</td>
</tr>
<tr>
<td>TIME FinishBy;</td>
</tr>
<tr>
<td>HANDLE hThread;</td>
</tr>
<tr>
<td>DWORD dwThreadID;</td>
</tr>
<tr>
<td>METHOD_TYPE nMethod;</td>
</tr>
<tr>
<td>int nCurrentPriority;</td>
</tr>
<tr>
<td>HANDLE hExceptionHandler;</td>
</tr>
<tr>
<td>TIME tStartTime;</td>
</tr>
<tr>
<td>TIME tStopTime;</td>
</tr>
<tr>
<td>TIME tNextInvocation;</td>
</tr>
<tr>
<td>TIME tInvocationDeadline;</td>
</tr>
<tr>
<td>TIME tCurrentExecDeadline;</td>
</tr>
<tr>
<td>MCB_TYPE *next;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 2. The Contents of MCB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wait for periodic invocation;</td>
</tr>
<tr>
<td>Update timer-related informations in MCB’s;</td>
</tr>
<tr>
<td>For each inactive SpM</td>
</tr>
<tr>
<td>If (time to activate) activate the SpM;</td>
</tr>
<tr>
<td>For all active SpM’s and SvM’s</td>
</tr>
<tr>
<td>If (deadline is passed)</td>
</tr>
<tr>
<td>invoke the exception handler;</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>reassign priority due to the deadline left;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 3. Operations of WTMT</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2.3 DFMCS (Data-Filed Message Communication Sub-System)</td>
</tr>
</tbody>
</table>

DFMCS (Data Field Message Communication Sub-System) of WTMOS has been implemented on a multicast platform of the Windows system layer. Although we have used the Microsoft mail-slot system as the multicast platform for simplicity of implementation, this platform will be switched to a real-time guaranteed multicast platform in the near future. DFMCS consists of two middleware system tasks, Data-Field interface stubs, and two message-queues. Two system tasks are IMMT (Incoming Message Management Task) and OMMT (Outgoing Message Management Task). Two message-
Types of messages handled by DFMCS are as follows:

- Distributed IPC messages: messages used in synchronization between distributed TMO methods;
- Messages from Windows system: messages created during interactions between a user and the application through windows.

In the WTMOS environment, an SvM of a TMO can be activated not only by messages delivered from remote or local methods but also by local interactive messages from the window that is registered as a control window of an SvM.

IMMT periodically polls the multicast-platform-buffer and the local Windows-message-buffer and fetches the delivered messages to the IMQ. After message fetch, IMMT wakes up the SvM’s waiting at the associated Data-Field channel so that an SvM can get its messages in the middle of a receive-group Data-Field interface routine in WSI.

Send-group Data-Field interface routines put outgoing messages to remote or local methods into XMQ. To deliver these messages, OMMT periodically sends messages out to the multicast platform.

Local Data-Field channels are classified into three groups as shown in Table 4. The first one is used for communications among distributed middlewares and the second group is reserved for future use. The last group is used for communications among methods of TMO’s. At the time of SvM registration, a service-accept-channel is automatically allocated and associated to the SvM. When a method (an SvM or an SpM) issues a service request to an SvM, it calls the corresponding WSI routine with arguments of the target SvM name, service-request-content, and the reply-channel. The target SvM name that is globally known, is mapped into the local request-accept-channel by receiving WTMOS. This mapping requires the maintenance of the global SvM name (id) table between WTMOS’s.

<table>
<thead>
<tr>
<th>Channel</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>for communications between WTMOS’s.</td>
</tr>
<tr>
<td>1~5</td>
<td>reserved for future use.</td>
</tr>
<tr>
<td>6~15</td>
<td>for TMO applications</td>
</tr>
</tbody>
</table>

Operations of MCS (Message Communication Sub-
Wait for periodic invocation;
Poll and get incoming message from multicast platform and windows message buffer
Put message into system IMQ (per channel);
for (all messages){
    if (SvM waiting at this channel)
        Activate the methods;
}

Table 6. Operations of the IMMT

3.2.4 Handling of Window Interface Messages

As stated in section 3.2.3, an SvM can be activated by window interface messages. In window applications, interfaces are generally implemented as message driven callback functions. However, in the WTMOS environment, it is a natural way to interface users through windows that SvM’s play the roll of callback functions, because an SvM is a message-triggered method. To support this, a universal callback function for all local windows is defined internally and it puts all messages generated by the window interface into buffers with window specific information. These messages are fetched into IMQ by IMMT periodically and finally delivered to a target SvM as in Figure 6.

3.2.4 Execution of TMO methods

Time-triggered methods (SpM’s) and message-triggered methods (SvM’s) are different from methods of conventional objects in that they must be implemented as dynamic threads while preserving the membership characteristics of a class. The creation and execution of a method are shown in Figure 7. At the time of static or dynamic TMO class creation, MCB’s and corresponding threads for the methods f a TMO are created by the system-supported (object library: WTMOS layer-3) constructor. Created methods must register information on its AAC at the beginning of its execution, and then it becomes to be in suspended state until activated by messages or the system timer.

The whole structure of WTMOS is described in Figure 8. Figure 9 is an example implementation of a TMO application in C++ on WTMOS (a ring buffer example).

4. Concluding Remarks

In this paper, the structure and implementation techniques of WTMOS have been described. WTMOS is a soft real-time TMO execution environment designed and implemented as a middleware between the Windows 95/NT OS and TMO applications. WTMOS supports the followings:

- TMO (Time-triggered Message Triggered Object)
- Monitoring of on-time activation and execution deadlines of TMO methods;
- Data-Field distributed IPC;
- User defined deadline exception handlers;
- Window interface and management by SvM’s;

Although WTMOS has some restrictions in being used as a hard real-time platform, it has several advantages as a soft real-time object oriented programming platform. The advantages can be stated as follows;

- Easy implementation and low development cost;
- No limitation in using WIN32 API and window management;
- Powerful and easy programmable platform for applications such as multimedia services.

Since the basic idea of the layered structure and functions of WTMOS came from the DREAM micro-kernel [3] which aims at a design time guaranteed computing system, we expect that the WTMOS middleware can be easily ported on a service time guaranteed micro-kernel and can be used as a hard real-time platform.

```cpp
class Ringbuf_TMO {
private: // ODS
    CREW_Ringbuf Ringbuf; // ODS + CREW

    // SpM definition
    SpM_With_Exception(Reader);
    SvM(Writer);
public:
    Ringbuf_TMO(void) { .. };   
    void init(void) {
        SpM_Init_Exception(Reader);
        // with deadline exception handler
        SvM_Init(Writer);
    }

    SpM_ Body(Reader)
    // register SpM (thread -> SpM)
    WTMOS.Register_SpM (AAC_Information);
    while(1) {
        // wait periodic invocation
        WTMOS.Wait_invocation();
        TMO_name.Ringbuf.Get(&k);
        process the data;
        // end of periodic execution
        WTMOS.SpM_Yield();
    }

    SpM_Exception(Reader)   
    {   do exception handling;
    }

    SvM_ Body(Writer)  
    {
        WTMOS.Register_SvM(& Service_Accept_Ch,
                           Control_Window_Info);
        while(1) {
            // wait Service_ Request(& Msg);
            Switch (Msg_type) {
                // handle request according to the
                // to the type and source
            }
            WTMOS.SvM_Yield();
        }
    }

    class CREW_Ringbuf: public CREW {
private:
    int   buf[BUFMAX], Front, Rear, Nitems;
    Cond   Space_Exist; // cond for space
    Cond   Data_Exist;  // cond for data
public:
    // Constructor
    CREW_Ringbuf(void) { do init; }   
        int    Put(int)
        {   EX_LOCK();
            if (Nitems == BUFMAX)
                if (Cond_Wait(& Space_Exist,
                               Wait_Limit) == -1) { // timeout
                    EX_UNLOCK();
```
```cpp
return -1;
}
Nitems++; Rear = (Rear+1)%BUFMAX;
buf[Rear] = item;
Cond_WakeUp(&Data_Exist);
EX_UNLOCK();
return 1;

int Get(int *);
{
do something;
}

Figure 9. An Example C++ Implementation of a TMO Application on the WTMOS

5. Reference


