Proc. ISADS 2001 (IEEE CS 5th Int'l Symp. on Autonomous Decentralized Systems), Dallas, TX, March 2001, pp.327-335.

TMOES: A CORBA Service Middleware Enabling High-Level Real-Time Object Programming

K.H. (Kane) Kim, J.Q. Liu, H. Miyazaki,
DREAM Lab, UC Irvine
Irvine, CA 92697
khkim@uci.edu, http://dream.eng.uci.edu

and

E.H. Shokri
Sun Microsystems
Palo Alto, CA 94303
Eltefaat.Shokri@eng.sun.com

Abstract: Extending the CORBA programming and execution environments to support real-time distributed applications is a subject of growing interests to both research and industry community. The time-triggered message triggered object (TMO) programming scheme has been established to remove the severe limitations of conventional object structuring techniques in developing real-time distributed applications. To facilitate the construction of TMO-structured real-time distributed applications consisting of CORBA-compliant components, a middleware named TMO Execution Support (TMOES) has been created. TMOES instantiations residing in cooperating computing nodes also communicate among themselves only via ORBs conforming to the standard specifications. TMOES instantiations cooperate among themselves to support timely execution of TMO methods and interactions among remote TMOs. This paper presents the architecture and an implementation model of TMOES.

Keywords: CORBA, real-time, TMO, time-triggered, message triggered, objects, distributed system.

1. Introduction

Distributed object programming such as CORBA-compliant programming [OMG98a, Ion00, Omn00] has become the main-stream approach for constructing highly decentralized computing applications. Also the real-time application branch of the distributed computing market is now growing much faster than in previous decades. As a result, real-time object-oriented computing has become a hot research topic [Bol00, Kim00, Sch00, Sel00, Thu96, Yau00].

In recent years, we have established the *time-triggered message triggered object* (TMO) programming scheme as a high-level programming model for real-time distributed applications [Kim97]. The TMO programming scheme offers a highly abstract and easy-to-handle programming interface to real-time distributed computing (RTdC) application developers. The power of the TMO programming scheme in supporting the design and implementation of distributed computing applications consisting of highly autonomous subsystems was discussed in [Kim95].

Recently, extending the CORBA standards [OMG98b, OMG00] to support RTdC applications has become a subject of considerable interest to the real-time distributed computing research community. We have investigated into whether CORBA-compliant TMOs can be realized without any extension or with minimal extension of the core component of the CORBA standard, i.e., Object Request Broker (ORB) and Interface Description Language (IDL). The answer was given positively in [Kim99a] and it has been further substantiated by our recent development of a support middleware which itself relies only on ORBs for all inter-node communication activities while supporting the execution of CORBA-compliant application TMOs. This middleware was named the *TMO execution support* (TMOES).

TMOES accepts the specifications of desirable timing requirements to be imposed on various components of a TMO, and manages local node resources for timely execution of the components with the cooperation of local operating systems of participating nodes. One of the capabilities that the TMO execution facilities must possess is the timely transmission of remote service requests and service result returns between client and server TMOs. To provide this capability, the local operating system, the middleware and the communication subsystem together should provide a predictable real-time communication service. We can use a real-time ORB if it is available in a reliable form [OMG98a, Sch98]. Otherwise we may rely on a network configuration manager maintaining a network environment in which message traffic is regulated to provide a reasonable communication delay bound.

In this paper we present the architecture and an implementation model of the middleware TMOES. Some API definitions and code examples are also provided in this paper to illustrate the programming style which is based on TMOES and applicable to efficient development of RTdC applications. In addition, our experiences in adapting the TMOES model to different CORBA platforms are briefly discussed.

The rest of the paper is organized as follows. The TMO scheme is reviewed briefly in Section 2. In Section 3, the structure of TMOES and the service request types supported by TMOES are presented. Section 4 introduces the support library and example codes. In Section 5, implementation experiences of TMOES are discussed. Section 6 is a conclusion of the paper.

0-7695-1065-5/01 $10.00 © 2001 IEEE

2. An overview of the TMO programming scheme

The TMO programming scheme was established in early 1990's [Kim97] with a concrete syntactic structure and execution semantics for reliable design and implementation of real-time systems. The basic TMO structure is depicted in Figure 1.

TMO is a syntactically minor and semantically powerful extension of the conventional object(s). Significant extensions are summarized below and the second and third are the most conspicuous unique extensions.

(a) Distributed computing component:

The TMO is a distributed computing component and thus TMOs distributed over multiple nodes may interact via remote method calls. To maximize the concurrency in execution of client methods in one node and server methods in the same node or different nodes, client methods are allowed to make non-blocking types of service requests to server methods.

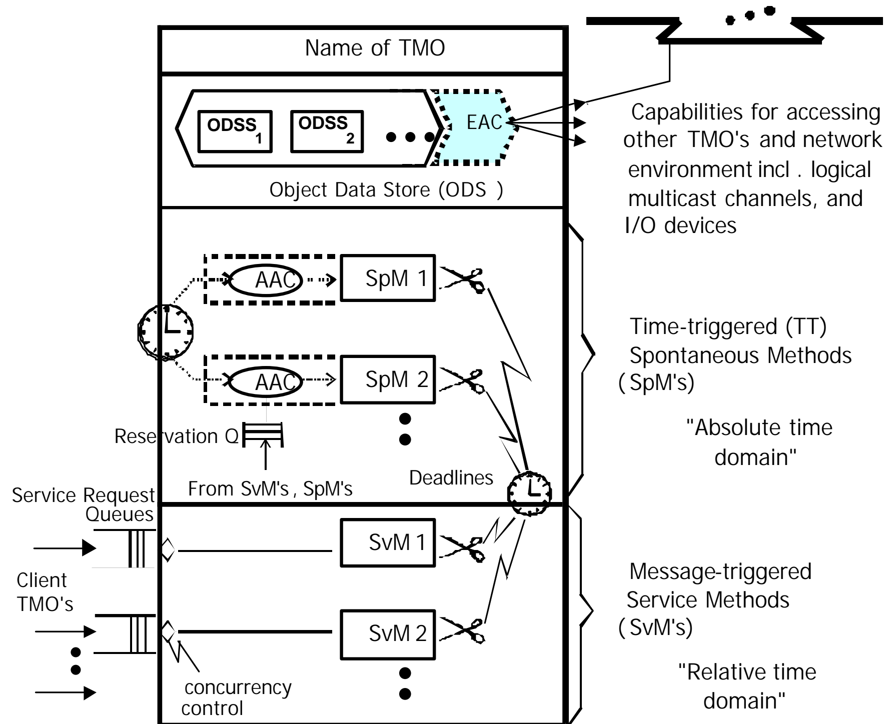


Figure 1. The basic structure of TMO (Adapted from [Kim97]

(b) Clear separation between two types of methods:

The TMO may contain two types of methods, *time-triggered (TT-) methods* (also called the *spontaneous methods* or *SpMs*), which are clearly separated from the conventional *service methods* (*SvMs*). The SpM executions are triggered upon reaching of the real-time clock at specific values determined at the 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 SpMs.

(c) Basic concurrency constraint (BCC):

This rule prevents potential conflicts between SpMs and SvMs and reduces the designer's efforts in guaranteeing timely service capabilities of TMOs. Basically, *the execution of an SpM should not be interfered by the execution of any SvMs*. An SvM is allowed to start only if no SpM that accesses the same *object data store segments* (ODSSs) 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.

(d) Guaranteed completion time for method execution and deadline for result return:

As in other real-time object models, the TMO incorporates deadlines and it does in the most general form. Basically, for output actions and method completions of a TMO, the designer guarantees and advertises execution time-windows bounded by starting times and completion times. In addition, deadlines can be specified in the client's calls for service methods for the return of the service results.

Triggering times for SpMs 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 object requests future executions of a specific SpM. The AAC specifying candidate triggering times rather than

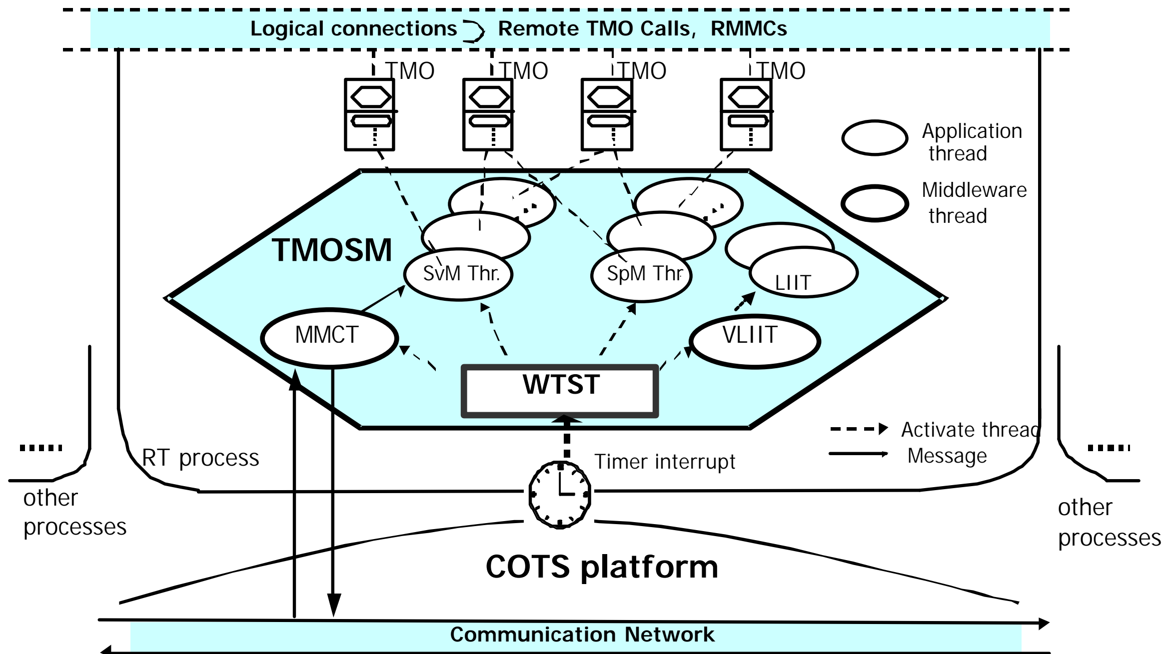


Figure 2. The basic internal thread structure of TMOSM

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

An underlying design philosophy of the TMO scheme is that a real-time computer system will always take the form of a network of TMOs. The designer of each TMO provides a guarantee of timely service capabilities of the object. The designer does so by indicating the *guaranteed execution time-window for every output* produced by each SvM as well as that for every output by each SpM and the *guaranteed completion time* for each SvM in the specification of the TMO. Such specification of a TMO is advertised to the designers of potential client objects. Before determining the time-window specification, the server object designer must convince himself/herself that with the *object execution engine* (hardware plus operating system) available, the server object can be implemented to always execute each SvM such that relevant output actions are performed within the associated time-windows. The BCC contributes to major reduction of these burdens imposed on the designer.

3. TMOES structure

A cost-effective way to support TMO-structured distributed real-time programming is to build a TMO execution engine as a middleware running on well established commercial software/hardware platforms. An efficient middleware architecture named *TMO support middleware* (TMOSM) that could easily be adapted to many commercial off-the-shelf (COTS) platforms was developed [Kim99b]. It has turned out that in adapting TMOSM to the CORBA environment, i.e., the environment where all communications among application objects (TMOs) as well as all communications among distributed instantiations of the middleware must be done via ORBs, the basic structure of TMOSM can remain intact. The TMOSM adapted into the form of a CORBA service is the *TMO execution support* (TMOES). In this section, we first briefly introduce the major structural aspects of TMOSM, and then discuss how TMOES executes all types of communications among distributed CORBA-compliant TMOs.

3.1 Thread structure

The internal thread structure of TMOSM is shown in Figure 2. This architecture has been devised to enable relatively easy analysis of the worst-case execution times of TMO methods.

TMOSM consists of three types of threads, *application threads, middleware threads,* and the *super-micro thread.* TMOSM assigns one application thread to each SpM or SvM of an application TMO. Middleware threads are *periodic threads* (periodically activated to run for a time-slice), each being responsible for a major part of the functions of TMOSM. The authors believe that structuring of middleware threads as periodic threads is a fundamentally sound approach which leads to easier analysis of the worst-case time behavior of the object execution engine without incurring any significant performance drawback.

The super-micro thread is called the WTST (*Watchdog Timer & Scheduler Thread*). It is a "super-

thread" in that it runs at the highest possible priority level. It is also a "micro-thread" in that it manages the scheduling / activation of all other threads in TMOSM. Therefore, WTST is activated whenever a thread switching needs to be performed, e.g., upon expiration of a time-slice. Even those threads created by the node OS before TMOSM starts are executed only if WTST allocates some times-slices to them. Also, WTST checks for any deadline violations and if a violation is found, it provides an exception signal to the relevant computation unit.

The three middleware threads function as follows:

(1) MMCT (Middleware Message Communication Thread): This periodic thread manages the sending of *middleware messages* through the communication network. Middleware messages are the messages exchanged among the middleware instantiations running on different nodes to support interaction among TMOs. MMCT also distributes middleware messages coming through the network to their destination threads.

(2) VMST (Virtual Main System Thread): Periodically a time-slice is conceptually given to this virtual thread which merely represents all application threads running TMO methods. The actual time-slice allocations are taken by WTST that executes the application scheduler function. In summary, every time-slice conceptually belonging to the VMST is allocated to a fairly selected application thread.

(3) VLIIT (Virtual Local I/O Interface Thread): This virtual thread maintains a pool of threads which are called *local I/O threads* (LIITs) and execute the I/O requests from application threads. The time-slices allocated to VLIIT are actually distributed to LIITs. Each LIIT is assigned to execute an I/O function that utilizes the I/O capabilities of the host node platform including serial character I/O, disk I/O, network I/O involving messages which are not middleware messages, etc. This VLIIT approach has been motivated by the desire to make it easier to analyze with high precision the temporal predictability of application program-segments not involving I/O (and, to a less extent, the temporal predictability of I/O activities).

Several features of this TMOSM architecture contribute to simplying the analysis of the execution time behavior of application TMOs running on TMOSM. First, the strictly periodic nature of middleware threads and dedication of each middleware thread to a specific functionality enable largely independent analysis of the part of the execution time behavior of application TMOs that depends on a particular middleware thread. For example, in computing the maximum time taken for transmitting a message α in a queue attached to MMCT to a queue attached to the MMCT in a remote node, one needs to focus on a few factors only: the number and sizes of the messages in the queue ahead of α, the size of α, guaranteed bandwidth of the network path between the source and the destination, and the frequency and the size of the time-slice given to MMCT in each node.

Secondly, the execution time of an I/O can be in general specified, analyzed, and measured in a larger time gain than that which can be used in specifying and analyzing the computation relying on the CPU only. Therefore, dedicating LIITs to handling such I/O activities enables the high-precision analysis of the execution time behavior of the CPU-intensive computation.

3.2 TMO network configuration manager (TNCM)

Each TMOSM instantiation that resides in a node participating in an RTdC application must know the locations and status of other participating nodes in order to support the service request communications between TMOs. The collection, distribution, and maintenance of those information are the responsibility of a component of the TMOSM, called the *TMO network configuration manager* (TNCM). The roles of TNCM are:

(1) To manage the TMO network configuration information, such as node number, location, status, etc; and

(2) To synchronize the real-time clocks of all the participating nodes.

An instantiation of TNCM runs on every participating node and distributed TNCM instantiations cooperate with each other. Each instantiation maintains a record on the host node where it is running and it includes the ID of the host, information on the links connecting the host to the neighbor nodes, etc. One TNCM instantiation, designated as the *master TNCM*, takes the responsibility for gathering registrations of all the other TNCM instantiations, called the *worker TNCMs*. The master TNCM also orders worker TNCMs to synchronize their clocks with its own clock and announces the starting time of distributed computation to them.

3.2.1 Initialization of TNCMs in TMOES

In TMOES, an instantiation of TNCM is a CORBA object and every communication among TNCM instantiations goes through ORBs. During the initiation of the distributed computing nodes which will execute an application TMO network, each TNCM instantiation hosted on a participating node performs recognition of other cooperating TNCM instantiations and clock synchronization as follows.

In the beginning, the master TNCM announces its location so that each worker TNCM may learn it. For the announcement, the current TMOES provides two options, one relying on the CORBA Name Server and the other without it. Application programmers need to specify either option to be used in configuration files of TMOES instantiations on all nodes. In the first option, the master registers itself with the CORBA Name Server by reporting the pre-defined name (“MASTER”, “TNCM”),

where the first parameter is the "ID" and the second parameter is the "kind". Then every worker contacts the CORBA Name Server at the starting time and obtains the object reference of the master. In the second option, the master writes its object reference into a file that all workers can access. The location of the file stored must be specified in configuration files.

Using the object reference of the master TNCM obtained, each worker TNCM sends a registration request containing its configuration information and the object reference of itself. Then it waits to hear about the list of registered TNCM instantiations from the master.

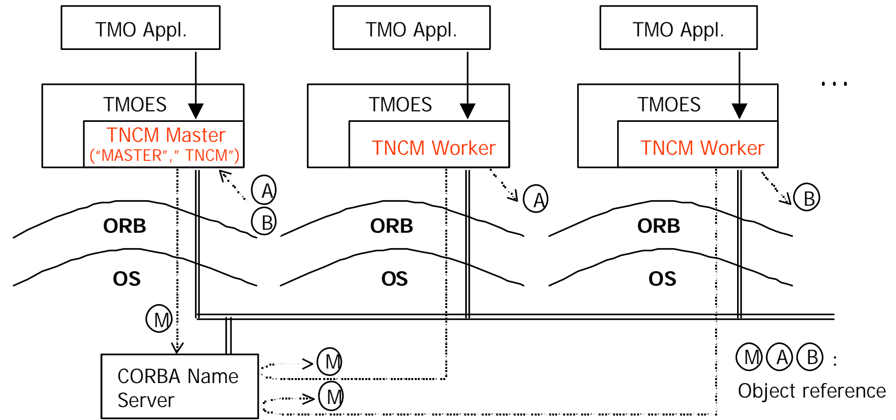


Figure 3. Registration of TNCMs with the CORBA Name Server

When the master TNCM receives a registration request from a worker TNCM, it records the configuration information included in the request appropriately. After all of the worker TNCMs have been registered (the total number of participating nodes comes from a configuration file), the master broadcasts the list of registered TNCM instantiations so that every worker node may use this information in supporting the *Real-time Multicast and Memory replication Channel* (RMMC) [Kim00].

After announcing the list, the master conducts clock synchronization with worker TNCMs and then announces the distributed computation starting time to all the workers.

The following is the interface definition for TMOES including TNCM expressed in the CORBA Interface Definition Language (IDL):

```
#pragma prefix "DREAM-LAB.UCI"
module TMOES {
  typedef long long  C21_age;
  struct TMOESnode {
    long     NodeID;
    short    SNS_Status;
    short    HealthyStatus;
    string   IOR_TNCM;
    string   IOR_ClockSync;  };

  typedef sequence<TMOESnode>  TMOESlist;

  interface TNCM_ORB {
    short RegisterTMOES (in TMOESnode   node);
     // Worker -> Master
    void RecvTMOESList(in TMOESlist   Namelist);
     // Master -> Worker
    void StartTMOApp(in C21_age   time);
     // Master -> Worker  };

  interface ClockSync_ORB {
   short ClockSync();  // Master -> Worker
   void RecvMasterClockTime( in C21_age   time);
   // Master -> Worker};

  interface RMMC_ORB {
        ...
  };
};
```

Currently, the *TNCM_ORB* interface defines three operations: (1) receiving a registration request from a worker, (2) receiving the TNCM instantiation list from the master, and (3) receiving the announcement of the distributed computation starting time from the master. The *ClockSync_ORB* interface defines two operations: (1) receiving the order from the master for an immediate return, and (2) receiving the order for synchronizing the local clock to the ordered time value.

3.3 Communication types and paths in TMOES

There are seven types of communications among TMOs and among TMOES instantiations:

1. Inter-node service request
2. Inter-node result return
3. Clock synchronization
4. Intra-process (node) service request
5. Intra-process (node) result return
6. RMMC support message
7. Network configuration management

Types 1, 2, 4, and 5 are communications between application TMOs, and types 3, 6, and 7 are communications between TMOES instantiations.

These seven communication types can be classified into three groups depending on the communication path used as shown in Figure 4.

Two paths, path 1 and path 2, facilitate the service requests between application TMOs. In both paths, when the requested server method is activated, the first action in the method execution is to report the starting of the execution to TMOES and the last action is to report the completion of the execution. These two notifications

enable TMOES to control the execution of the methods and detect deadline violations. When the client calls the stub (which is created by the IDL compiler and leads to the remote server method) directly, path 1 is used.

In path 2, the client calls TMOES instead of a stub function. The TMOES then dispatches a thread from its thread pool to handle this service request and the client can continue to do other jobs after calling TMOES. Therefore, the main difference between path 1 and path 2 is in that in path 1 the client expecting result returns from the server is blocked to wait for the completion of the requested service, while in path 2 the client can continue executing and check the requested service result at a later time. Path 2 is thus useful for making non-blocking service requests. In both paths, the server's guaranteed execution time is involved and its violation is detected by the server node. On the other hand, while the clients using path 2 can set clients' deadlines, the clients using path 1 cannot.

Path 3 facilitates the communications between TMOES instantiations. Path 3 is similar to path 1 logically, but it is used only for communications between TMOES instantiations.

4. TMO Execution Support Library (TMOESL)

As a user-friendly API for TMO programmers, a library named *TMO Execution Support Library* (TMOESL) has been developed. It consists of a collection of C++ classes and functions, and provides a user-friendly interface wrapping TMOES services for real-time CORBA application programmers.

4.1 Service request APIs

Application programmers might issue service requests by either direct calls to stubs generated from programmers' IDL files or indirect calls to the stubs through TMOES API. In the former case TMOES in a caller node of service requests isn't involved in handling service requests. Thus application programmers cannot exploit the capability of TMOES for checking timeliness of service completions and result returns. So, the following types of service requests are made this way (i.e., direct calls to stubs):

(1) *Blocking service requests with no specification of clients' deadlines*;

(2) *One-way service requests* which do not lead to any result return.

Only in the latter case of using TMOES API calls, they can take advantage of such timeliness-checking capability of TMOES. The types of such service request APIs are briefly described below.

(3) *Blocking service requests with clients' deadlines*

In this type of a service request, the client specifies a deadline for the return of the result of the service request.

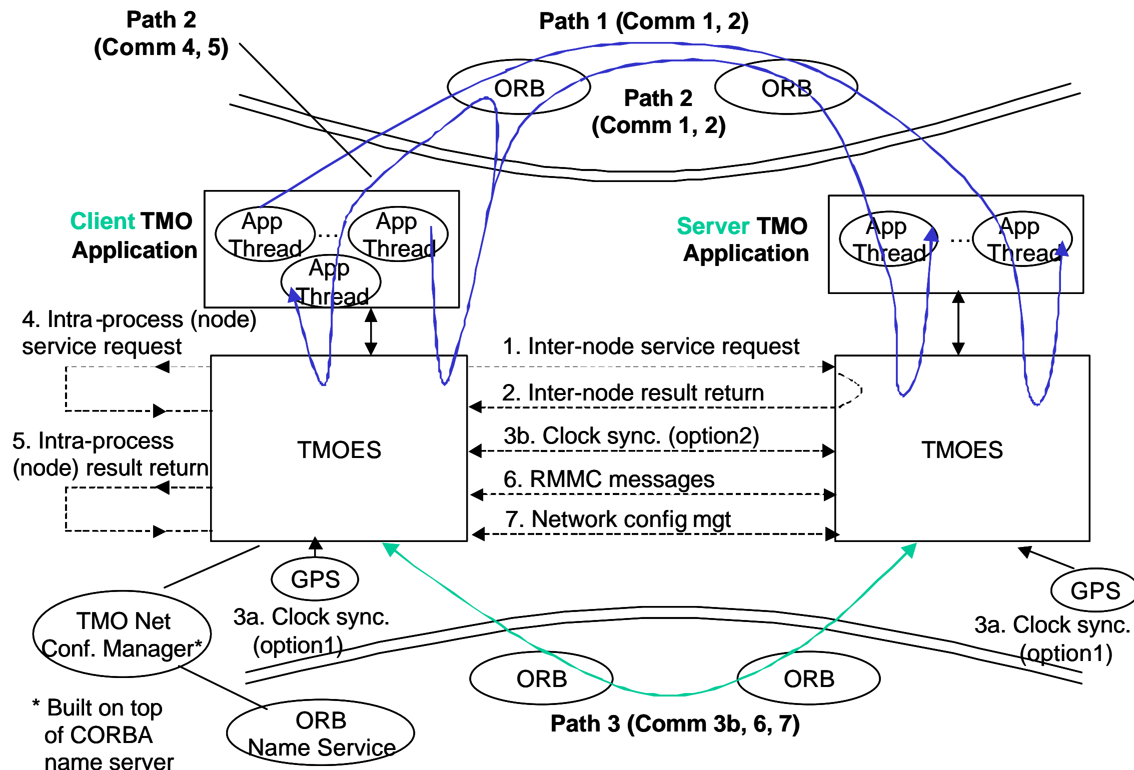


Figure 4. Service request paths in TMOES

TMOES on the client side, with the cooperation of other involved TMOES instantiations, tries to process the service request within the deadline and returns an error flag if a deadline violation is detected. The client is blocked until the service result returns.

```
typedef void (*TMOESL_WRAPPER_FUNC)();

int BlockingCallForAgent(
  TMOESL_WRAPPER_FUNC   WrapperFunc,
  const MicroSec &   ResponsePeriod);

int BlockingCallFo rAgent(
  TMOESL_WRAPPER_FUNC   WrapperFunc,
       const tms &   RTDeadline);
```

A wrapper function *WrapperFunc* is a user-supplied function that actually calls the stub which corresponds to an operation defined in programmers' IDL file and is generated by an IDL compiler. Calling for *BlockingCallForAgent()* results in passing *WrapperFunc* to TMOES along with a deadline for result return. The expression "CallForAgent()" here implies that TMOES is asked to execute *WrapperFunc* and thus TMOES may be viewed as an agent in this case. Programmers may specify a deadline in two ways: *ResponsePeriod*, an interval from the time the request is issued to the time the service result returns, or *RTDeadline*, a deadline represented as a calendar time.

An example of a wrapper function and its use in a *BlockingCallForAgent()* are shown in Figure 5. The figure shows code-segments of an example application consisting of two simple TMOs which exchange one integer value with each other. *SpM1::SpMBody()* calls *BlockingCallForAgent()* with two parameters, a pointer to a wrapper function *SpM1::WrapperFunc()* and 50 milliseconds as the client's deadline. *SpM1::WrapperFunc()* reads a counter from *ODSS1*, passes it as a parameter to *T2::Decrement()* which is defined in TMO2, and then stores the returned value back into *ODSS1*. T2*::Decrement()* just decrements the value received and returns it.

(4) *Non-blocking service requests*

In this type of a service request, the client is not blocked during the processing of the service request and might continue doing other jobs. The client may check later to see if the result has returned. TMOES assigns another thread to execute *WrapperFunc.*

```
int NonBlockingCallForAgent(
  TMOESL_WRAPPER_FUNC   WrapperFunc,
  tmsp &   Timestamp);
```

The *Timestamp* returned from TMOES is used later as an identifier of the service request when checking the availability of the return result.

(4a) *Non-blocking check for result*

The client immediately receives the status of the corresponding non-blocking call.

```
int NonBlockingCheckOfAgentResult(
  const tmsp &   Timestamp);
```

The return value of this *NonBlockingCheckOfAgentResult()* indicates whether the result of the remote method call made by the agent has been returned. If not yet, the client may continue other jobs and then later check the result. Otherwise the client may access the results contained in some parameters.

(4b) *Blocking check for result with deadline*

Here the client will be blocked until the result returns. A violation of the deadline will be handled in the same way as in the case (3).

```
int BlockingCheckOfAgentResultWithDeadline(
  const tmsp &   Timestamp,
  const MicroSec &   ResponsePeriod);

int BlockingCheckOfAgentResultWithDeadline(
  const tmsp &   Timestamp,
  const tms &   RTDeadline);
```

In the case of a failure of the server SvM in providing the service, TMOES on the server side will, if it can, return a service failure notice to the TMOES on the client side. If there is no result returned from the server's TMOES upon the arrival of the deadline imposed by the client, the client's TMOES returns an error flag to the client SvM or SpM that invocated the *BlockingCheckOfAgentResultWithDeadline()*.

4.2 Configuration of TMOES

A configuration file is used to define how TMOES should be configured on each participating node. The information to be specified is as follows:

```
TOTAL_NODE_NUM     2     // Number of nodes
MASTER_NODE          yes  // or no
NODE_NAME            node1
MASTER_NODE_IOR_FILE      \\pc1\master.ior
```

One node must be specified as the master node of a TMO application. MASTER_NODE_IOR_FILE specifies the name of the file that contains the object reference of the master TNCM. The master TNCM writes its object reference into the file, and the worker TNCMs read the file to obtain the object reference of the master TNCM. Unless a file name is specified, the checking TNCM would contact the CORBA Name Server and if it is the master, it would register its object reference and otherwise, it would obtain the object reference of the master.

5. Implementation experiences

We have implemented several prototypes of TMOES. The prototype running on the Windows NT platform with the OmniORB2 [Omn00] ORB and the prototype using Orbix [ION00] ORB were implemented first. Later we implemented TMOES running with two newer ORBs, the OmniORB3 and TAO 1.1 [Sch98], both of which are

```
//*** IDL definition of TMO1 ***
interface TMO1 {
  void Increment( inout long para);
  ...;  // might have multiple operations
};

//***  Source code of TMO1 ***
include "TMO2.h"       // Import a stub header of TMO2
TMO2_var g_T2_obj; // Object ref. of a CORBA object

class ODSS1:public ODSSBaseClass  {
private:  int m_inc, m_dec;
public:   ODSS1() : m_inc( 0), m_dec( 0) {}
  void GetInc( int& i) {    ...;    i = m_inc;    ...;  }
  void SetInc( int i) {    ...;    m_inc = i;    ...;  }
  void GetDec( int& i) {    ...;    i = m_dec;    ...;  }
  void SetDec( int i) {    ...;    m_dec = i;    ...;  }
};

class SpM1 : public SpMBaseClass  {
private: static ODSS1* m_pOdss1;
public:
  SpM1( ODSS1* pOdss1, access_mode_type mode1)
     :  m_pOdss1( pOdss1)  {
    // Associate ODSS, AAC info with this SpM
    build_regist_info_ODSS( m_pOdss1->get_id(), mode1);
    ...
    RegisterSpM();
  }
  static void WrapperFunc()  {
    CORBA::Long val;
    m_pOdss1->GetDec( val);
    g_T2_obj->Decrement( val);
    m_pOdss1->SetDec( val);
  }
  virtual void SpMBody()   {
    BlockingCallForAgent( WrapperFunc, 50*1000);
  }
};

ODSS1* SpM1::m_pOdss1 = NULL;

class Increment_Support : public SvMBaseClass  {
private: ODSS1* m_pOdss1;
public:
  Increment_Support(
     ODSS1* pOdss1, access_mode_type mode1)
      : m_pOdss1( pOdss1)   {
    // Associate ODSS, deadline info with this SvM
    build_regist_info_ODSS(m_pOdss1->get_id(), mode1);
    build_regist_info_guranteed_completion_time(20*1000);
    RegisterSvM();
  }
};

// Define an SvM support class for each of the additional
CORBA
// operations which are served by TMO1
```

```
class TMO1: public TMOBaseClass,
              virtual public POA_TMO1  {
private:
  ODSS1 m_Odss1;  SpM1 m_SpM1;
  Increment_Support m_Increment_Support;
public:
  TMO1( const char* TMO1_name,
          const tms& TMO1_start_time)
    : m_SpM1( &m_Odss1, RW),
      m_Increment_Support( &m_Odss1, RW)  {
    activate(TMO1_name, TMO1_start_time);
  }
  void Increment( CORBA::Long& para)  {
    ReportSvMStart( &m_Increment_Support);
    m_Odss1.SetInc( ++para);
    ReportSvMCompl( &m_Increment_Support);
   }
};

DWORD WINAPI ORB_Thread( LPVOID  lpParam)  {
  CORBA::ORB_ptr = (CORBA::ORB_ptr) lpParam;
  orb->run();
  return 0;
}

int main(int argc, char* argv[])  {
   StartTMOengine();  // TMOES initialization
  // ORB, POA, POA manager  initialization
  CORBA::ORB_ptr orb = CORBA::ORB_init ( argc, argv);
  ...;
  // T1 initialization
  tms T1_start_time = tm4_DCS_age( 5*1000*1000);
  TMO1 T1( "T1", T1_start_time);
  // Incarnate T1 object
  TMO1_var T1_obj = T1._this();
  // Pass this obj ref to clients of T1 in some way
  ...;
  // Obtain the obj ref of T2 in some way.
  g_T2_obj = ...;
  // Get ready to accept calls for server methods
  DWORD   tid;
  CreateThread( NULL, 0, ORB_Thread, orb, 0, &tid);
  // Start TMO application
  MainThrSleep();  // Let main thread sleep
  return 0;
}
// ******************************************

//*** IDL definition of TMO2 ***
interface TMO2 {
  void Decrement( inout long para);
};

//***  Source code of TMO2 ***
// TMO2 are the same as TMO1 except for the names of //
classes, types & variables. Replace the names;
// TMO1->TMO2, SpM1->SpM2,
// Increment->Decrement, g_T2_obj->g_T1_obj, ...
```

Figure 5. Code-segments of an example application consisting of two TMOs, T1 and T2

compliant with CORBA 2.3 specification that includes the Portable Object Adapter (POA).

Compared with our first implementations using Basic Object Adapter (BOA), later implementations using POA

showed much better portability. However, during the test of several ORB implementations, we found that some of them did not allow us to specify a multithreading model that would describe how POA would dispatch concurrently arriving service requests to threads.

For instance, some ORB implementations have only one thread to handle all service requests which a server receives. Suppose that a TMO supported by such an ORB has several SvMs (i.e. CORBA operations) with different deadlines. If the TMO receives service requests simultaneously, it cannot optimize the execution order of SvMs according to their deadlines. Since the underlying ORB has serialized the requests in the order of their arrivals and invokes a server operation after the operation invoked previously by itself has finished, TMOES cannot realize that multiple service requests are pending and it has no chance to adjust their execution order.

6. Conclusion

In this paper, we have discussed a cost-effective approach for developing CORBA-compliant TMO-structured real-time distributed application programs. It is realized via provision of a middleware and does not require any change in the ORB core. The initiation and scheduling of the executions of TMO methods are managed by TMOES which functions as a CORBA service and is a derivation of the TMO Support Middleware (TMOSM) model. With a user-friendly TMOES library, application programmers can easily build CORBA-compliant RTdC applications that take advantage of the capabilities provided by TMOES. We plan to perform further studies on the performance of TMOES implementations.

Acknowledgements: The research work reported here was supported in part by the US Defense Advanced Research Project Agency under Contract N66001-97-C-8516 monitored by SPAWAR, in part by the NSF Next-Generation Software (NGS) Program under Grant 99-75053, and in part by the NSF Information Technology Research (ITR) program under Grant No. 00-86147.

Reference

[Bol00] Greg Bollella and James Gosling, "The Real-Time Specification for Java", *Computer,* June, 2000, pp. 47-54.

[Ion00] IONA Technologies, www.orbix.com, 2000.

[Kim95] Kim, K.H.(Kane), Mori, K., and Nakanishi, H., "Realization of Autonomous Decentralized Computing with the RTO.k Object Structuring Scheme and the HU-DF Inter-Process-Group Communication Scheme", *Proc. ISADS '95 (IEEE Computer Society's 2nd Int'l Symp. on Autonomous Decentralized Systems),* April 1995, Phoenix, AZ, pp.305-312.

[Kim97] Kim, K.H., "Object Structures for Real-Time Systems and Simulators", *IEEE Computer*, Vol. 30, No.8, August 1997, pp. 62-70.

[Kim99a] Kane Kim, Eltefaat Shokri, "Two CORBA Services Enabling TMO Network Programming", *Proc. IEEE CS 4th Workshop on Object-oriented Real-Time Dependable Systems)*, Santa Barbara, Jan. 1999.

[Kim99b] Kim, K.H. Ishida, Masaki, Liu, Juqiang, "An Efficient Middleware Architecture Supporting Time-Triggered Message-Triggered Objects and an NT-based Implementation", *Proc. 2nd IEEE CS Int'l Symp. on Object-Oriented Real-time Distributed Computing (ISORC '99),* St. Malo, France, May, 1999, pp.54-63.

[Kim00] Kim, K.H., "APIs for Real-Time Distributed Object Programming", IEEE *Computer*, June 2000, pp.72-80.

[OMG98a] Object Management Group, Realtime Platform Special Interest Group, *Realtime CORBA*, May, 1998.

[OMG98b] Object Management Group, *The common Object Request Broker: Architecture and Specification,* Revision 2.3, June, 1998.

[OMG00] Object Management Group, *The common Object Request Broker: Architecture and Specification,* Revision 2.4, Oct, 2000.

[Omn00] AT & T Laboratories Cambridge, http://www.uk.research.att.com/omniORB/index.html, 2000

[Sch98] Douglas C. Schmidt, David Levine, and Sumedh Mungee, The Design of the TAO Real-Time Object Request Broker, *Computer Communications* Special Issue on Building Quality of Service into Distributed Systems, Elsevier Science, Volume 21, No. 4, April, 1998.

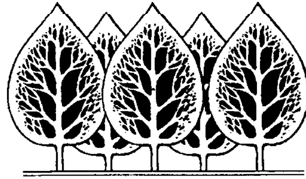
[Sch00] Douglas Schmidt, Fred Kuhns, "An Overview of the Real-Time CORBA Specification", *Computer,* June, 2000, pp. 56-63.

[Sel00] Bran Selic, "A Generic Framework for Modeling Resources with UML", *Computer,* June, 2000, pp. 64-69.

[Tsa96] Jeffrey Tsai, Thomas Weigert, "A Logic-Based Requirements Language fro the Specification and Analysis of Real-time Systems", *Proc. Second Workshop on Object-Oriented Real-time Dependable Systems (WORDS 96),* Laguna Beach, California, Feb 1-2, 1996, pp. 8-16.

[Thu96] Bhavani Thuraisingham, Peter Krupp, Victor Wolfe, "On Real-time Extensions to Object Request Brokers: A Panel Position Paper", *Proc. 3th IEEE CS Int'l Symp. on Object-Oriented Real-time Distributed Computing (ISORC '00),* Newport Beach California, March 2000, pp.182-185.

[Yau00] Stephen Yau and Fariaz Karim, "Component Customization for Object-Oriented Distributed Real-time Software Development", *Proc. 3th IEEE CS Int'l Symp. on Object-Oriented Real-time Distributed Computing (ISORC '00),* Newport Beach California, March 2000, pp.156-163.



Proceedings

5th International Symposium on

Autonomous Decentralized Systems

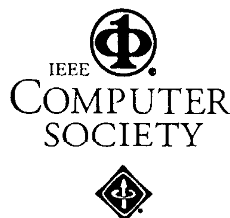
26–28 March 2001 • Dallas, Texas

Sponsored by

IEEE Computer Society
Information Processing Society of Japan
Society of Instrument and Control Engineers of Japan
Institute of Electronics, Information, and Communication Engineers, Japan

In cooperation with

International Federation for Information Processing
International Federation of Automatic Control
OMG
TINA-C
Manufacturing Science and Technology Center, Japan



Los Alamitos, California

Washington • Brussels • Tokyo