SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2004; (in press) (DOI: 10.1002/spe.620)

SP&E

Implementation of a TMO-structured real-time airplane-landing simulator on a distributed computing environment

Min-Gu Lee[1], Sunggu Lee[1,*,†] and K. H. (Kane) Kim[2]

[1]*EE Department, POSTECH, San 31, Hyoja Dong, Pohang 790-784, South Korea*
[2]*Electrical and Computer Engineering Department, University of California, Irvine, CA 92697, U.S.A.*

SUMMARY

In real-time simulation, the simulated system should display the same (or very close) timing behavior as the target system. The simulation accuracy is increased as the simulation time unit is decreased. Although there are several models for such systems, the TMO model is particularly appropriate due to its natural support for real-time distributed object-oriented programming. This paper discusses the results of the implementation of a real-time airplane-landing simulator on a distributed computing environment using the TMO model. Copyright © 2004 John Wiley & Sons, Ltd.

KEY WORDS: TMO; real time; distributed system; simulation; object-oriented programming

1. INTRODUCTION

The objective of *computer simulation* is to examine the behavior of a system by using a computer program to model the target system, possibly because actual testing using the target system may be dangerous, time-consuming, costly to implement, etc. A *real-time system* is a system in which there are timing constraints, and the timeliness (meeting a deadline) of a computation is as important as producing the correct computation result. *Real-time simulation* refers to computer simulation of a system in which the simulated system displays the same (or very close) timing behavior as the original system [1].

*Correspondence to: Sunggu Lee, EE Department, POSTECH, San 31, Hyoja Dong, Pohang 790-784, South Korea.
†E-mail: slee@postech.ac.kr

Contract/grant sponsor: The Korean Ministry of Information & Communication through its University Fundamental Research Program; contract/grant number: 2001-051-3
Contract/grant sponsor: NSF; contract/grant numbers: 00-86147 and 02-04050

Received 22 August 2002
Revised 5 April 2004
Accepted 5 April 2004

Real-time simulation can be divided into two categories: *scaled* and *non-scaled* real-time simulation. Scaled real-time simulation is a simulation method in which the target system is modeled in a time-accurate manner, but its time base is scaled (to longer or shorter time units) in order to achieve a faster simulation response or a more accurate analysis. Implementation of a global climate simulation system, which simulates the earth's climate changes over a period of several decades or centuries in order to learn about global warming and other global climate phenomena, is a good example of a scaled real-time simulation. For obvious practical reasons, the simulation should be performed using a scaled time base. It may also be desirable to increase or decrease the time base as the simulation progresses in order to view some sequences of climate changes in more detail. In a non-scaled simulation system, the simulation should progress at the same rate or speed as the target system. Non-scaled real-time simulation is particularly appropriate for real-time applications in which there is human interaction, such as in a real-time traffic control system or a pilot training simulator system. A pilot training simulator should be built as a real-time system with external input events such as the pilot's commands to control the airplane. A pilot who controls the airplane in the simulation system must feel that it has the same timing behavior as a real airplane. In performing non-scaled real-time simulation, it is very important to not only enhance the computation speed as much as possible, but also to execute the specified task (real-time task) with a predictable delay. That is, the deviation of the actual execution start time of the real-time task from its specified execution start time must be kept small regardless of the other system and application processes being executed by the computer system performing the simulation.

Commercial simulation tools typically do not have enough functionality to support the efficient development of real-time simulation systems. Some tools also do not provide effective support for simulation using distributed systems. To increase the accuracy of real-time simulation, an accurate modeling method (possibly involving long computations) is required and the simulation time unit should be decreased as much as possible. Due to the increased computational capability required by such an approach, many real-time simulation tasks will require the use of parallel or distributed systems. Distributed simulation using the object-oriented programming methodology would also be desirable in order to facilitate the programming task [2–5].

The TMO (*Time-triggered Message-triggered Object*) model [6–8] provides natural support for real-time distributed object-oriented programming. This paper discusses the implementation of a real-time simulation task, specifically an airplane-landing simulation, using the TMO model. On the basis of this implementation study and other independent measurements, the TMO model is evaluated with regards to its distributed computing and real-time support in addition to its ease of programming. The rest of this paper is organized as follows. A brief overview of the TMO model is presented in Section 2. Then the real-time support provided by the current TMO implementation is evaluated in Section 3. Sections 4 and 5 discuss our implementation of the airplane-landing simulator using TMOs. Section 6 discusses performance issues for TMO applications, and conclusions are presented in Section 7.

2. OVERVIEW OF THE TMO MODEL

2.1. TMO model

The TMO model, previously referred to as the RTO.k model [1,9], was introduced and developed by Kane Kim and his collaborators. This model supports real-time object-oriented programming

efficiently due to its syntactic structure and execution semantics. For a detailed description of the TMO model, the interested reader is referred to [6–8]. In the following, a brief overview of the TMO model is presented in order to aid in understanding the rest of the paper. The main features of the TMO model are summarized below.

(1) *Distributed computing object model*: the TMO model supports a distributed computing environment. TMOs can be distributed among multiple nodes. TMOs can communicate with each other using remote method calls, which are also referred to as service requests. To increase concurrency, client TMOs can make non-blocking requests to server TMOs. TMOs can also communicate with each other using logical multicast channels.
(2) *Separation of two types of methods*: the TMO makes a clear separation between SpMs (*spontaneous methods*) and SvMs (*service methods*). An SpM is a method which is triggered by the global time base, which is represented by the internal real-time clock; it is also called a time-triggered method. On the other hand, an SvM is a method which is triggered by the external service request of another TMO. Since this external service request occurs through a message, an SvM is a message-triggered method.
(3) *Basic concurrency constraints* (BCCs): this is a rule devised for preventing potential conflicts between SpMs and SvMs. If SpMs and SvMs attempt to access the same data groups in the ODS (Object Data Store), SvMs are delayed from executing until there is no longer a conflict. That is, SvMs can be run only when their executions will not disturb the execution of any SpMs through data access conflicts.
(4) *Guaranteed completion time for method execution* and *deadline for result arrival*: for output actions and method completions of a TMO, the designer guarantees and advertises execution time windows bounded by start times and completion times. Deadlines are also specified in the client's call for service methods for the return of the service results.

The structure of a TMO in the TMO model is shown in Figure 1. A TMO contains its name, an ODS, SpMs, SvMs, AAC and EAC (Environment Access Capability). The role of each component is described below.

(1) ODS: used for storage of properties and states of the TMO such as a simple variable or a complex class object. For example, in our simulator, as will be shown later, an airplane simulator object stores in this location information regarding its state and properties such as position, velocity, etc. Data members in the ODS are grouped into harmoniously sharable data store units called *object data store segments* (ODSSs) in a TMO. Each ODSS is thus a group of data members and is a unit that can be locked for exclusive use by one method execution at a time as well as for shared use by multiple concurrent method executions which perform read-only operations on the data members contained.
(2) SpM: a time-triggered method, which may be executed in a complex periodic manner. An SpM may represent a real-time periodic task. A part of the SpM is the *autonomous activation condition* (AAC), which defines the time windows for the execution of that SpM, the completion deadline, and the iteration period. The application level scheduler examines the AAC section to select the next running thread (a detailed description of scheduling is covered in Section 2.2). In the airplane simulator the state of an airplane, such as position, velocity, etc., must be updated during each period. A TMO can have multiple SpMs, and each one can have its own period.

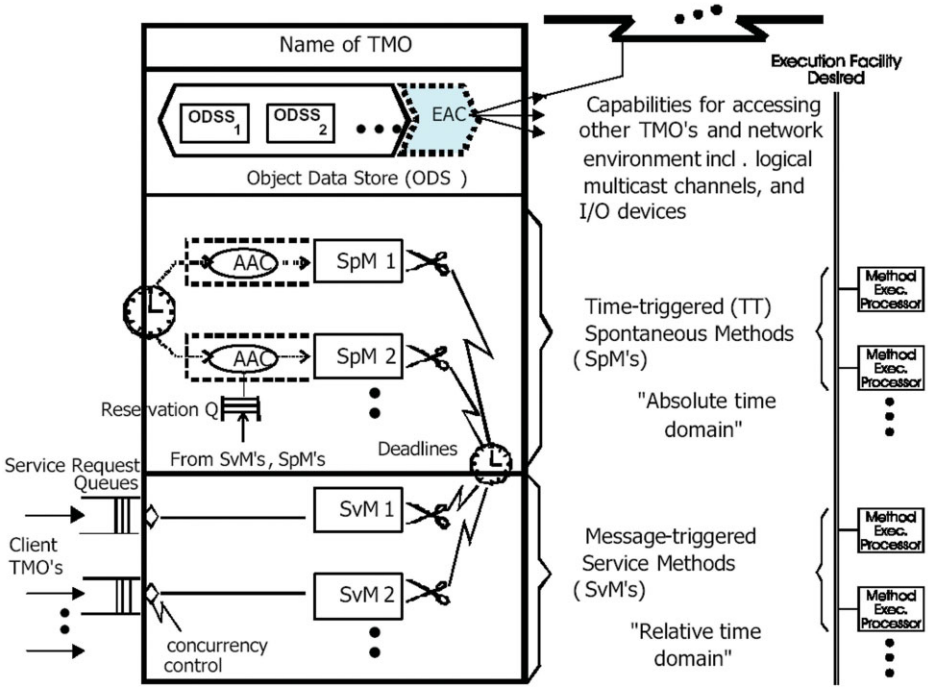

Figure 1. Structure of a TMO (adapted from [10]).

(3) SvM: a message-triggered method, which responds to external service requests. In the airplane simulator, an airplane TMO needs to make a request for an available runway to the control tower TMO. Then, when an appropriate runway is found by the control tower object, it responds to the airplane object by using another SvM.
(4) EAC: list of gates (access points) to remote object methods, logical multicast channels, etc., as shown in Figure 1. These gates provide efficient call paths to remote object methods, logical communication channels, and I/O interfaces.

For example, to see how an actual system can be modeled using TMOs, let us consider the modeling of an airplane-landing simulator. There should be a user-controllable airplane object—referred to as MyAirPlane TMO—and other environmental objects such as other airplanes, a control tower, hangars, etc. Naturally, MyAirPlane object needs to periodically update its state and process user input in a real-time manner. But these are independent tasks. Therefore, MyAirPlane object has two periodic tasks to be performed. There are two SpMs in MyAirPlane TMO, as shown in Table I. In addition, there should be data segments which store states, properties, and control events for MyAirPlane object. Furthermore, MyAirPlane object should communicate with ControlTower object and Space object, which represents the three-dimensional environmental space and detects any collision occurrences. There are two SvMs.

Table I. TMO model of MyAirPlane object.

Access capability: Space, Control Tower
ODS —UserControl // flaps, rudder, landing gears, spoilers, etc. —Properties // mileage, weight, max speed, lower bound speed to fly, etc. —States // velocity, position, remaining fuel, views, flight distance, alarm balances, etc. —Copy_Weather_From_Space

SpMs
—Update States
// Update velocity, position, current weight, remaining fuel, balances, etc.
—Processing User Control
// Update flaps, rudder, landing gear, request landing runway to Control Tower, etc.

SvMs
—ReceiveFromSpace
// Get Weather Info. from Space.
—ReceiveFromControlTower
// Get result of requesting for landing, alarm messages, etc.

The AAC is not shown in Table I, but each SpM has its own AAC section—the AAC section contains the period, deadline, and other real-time constraints for an SpM.

2.2. TMO middleware

To support the TMO model, the DREAM Laboratory at UCI (University of California, Irvine) developed the middleware named TMOSM (TMO support middleware) and the API wrapping the services of TMOSM named TMOSL (TMO support library) [10]. TMOSM is a middleware model used to support execution of TMO-structured applications on commercial, off-the-shelf operating systems; the implementation used in the research reported here runs on a Microsoft Windows NT environment and is referred to as TMOSM/NT. There are versions running on other OS platforms also (WinCE, Linux, etc.). TMOSL is a user-level API used to support programming in the TMO model. TMOSL basically consists of a set of C++ classes that can be inherited.

The internal structure of TMOSM/NT is shown in Figure 2. TMOSM/NT has three types of threads: middleware threads, application threads, and a super-micro thread. Application threads consist of threads that execute SpMs and SvMs of application TMOs, and the super-micro thread is the WTST (Watchdog Timer and Scheduler Thread), which runs at the highest priority in order to schedule all other threads on the node. As can be seen in Figure 2, the WTST is invoked by timer interrupt. The current version of TMOSM/NT sets the period of the watchdog timer at 3 ms. At the start of each period (i.e. every 3 ms), the WTST is invoked by a watchdog timer interrupt, at which time it manages scheduling/activation of any other threads in TMOSM. It also checks if there are any

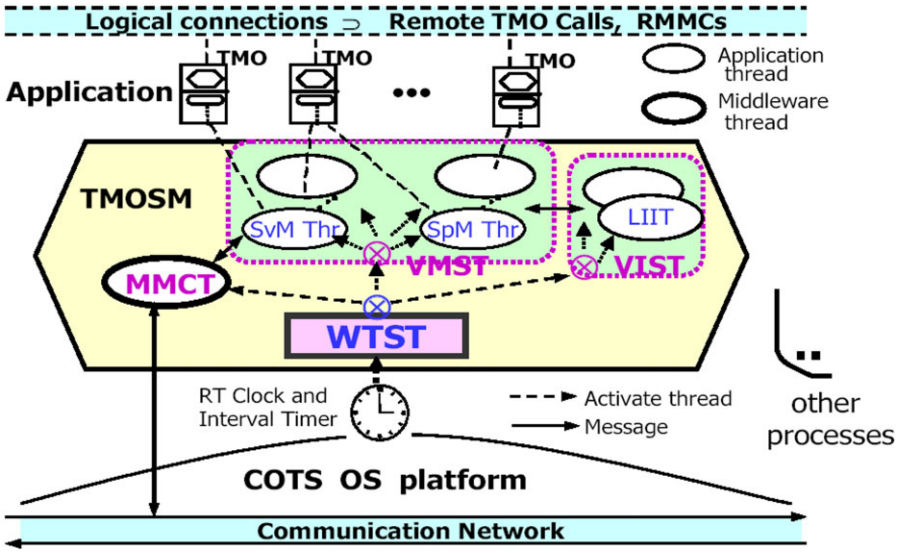

Figure 2. Internal structure of TMOSM (adapted from [11]).

deadline violations. If any violation is found, it signals this to the user. Also, as will be elaborated on in Section 3.3, the WTST is responsible for maintaining the global system clock.

The middleware threads are periodic threads, each responsible for a major portion of the functions of the TMOSM. The middleware threads consist of the following three main threads.

(1) MMCT (Middleware Message Communication Thread): this 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 interactions among TMOs.
(2) VMST (Virtual Main System Thread): this thread allocates CPU time slices to application threads such as SpMs and SvMs. When the VMST gets a CPU time slice, the application thread scheduler is called and one of the available SpMs or SvMs is selected to run.
(3) VIST (Virtual I/O System Thread): this virtual thread maintains the pool of local I/O interface threads (LIIT in Figure 2) and executes the I/O requests from application threads. The time slices allocated to VIST are distributed to LIITs. The LIIT performs disk I/O and network I/O involving messages which are not middleware messages.

Communication messages which are generated directly by TMO methods are sent usually by use of LIITs. In a real-time application, the timing behavior of application threads should be predictable to a great extent. By partitioning the time domain into a part used by the computation segments involving CPU(s) and memory only, i.e. the part used by VMST, and a part used by the computation segments

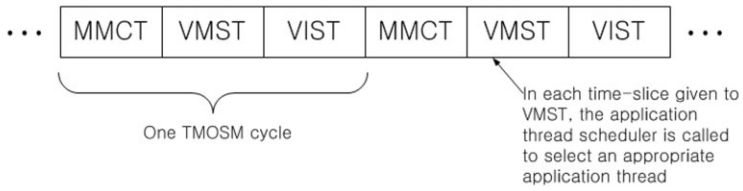

Figure 3. Two-level scheduling scheme (adapted from [10]).

involving I/O devices, i.e. the part used by VIST, control and analysis of the timing behavior of TMO method execution become easier. In particular, the analysis of the timing behavior of the application threads becomes much easier. For example, TMO supports a TMOSLprint() API, handled by the VIST, for printing messages to the console—this is similar to the printf() function in the C programming language.

TMOSM uses two-level scheduling. Figure 3 shows a two-level scheduling cycle. Middleware threads are scheduled by the WTST (i.e. middleware-level scheduler) and application threads are scheduled when time slices are given to them by the VMST. The VMST selects an application thread (SpM or SvM), and permits it to run its task. Therefore, the VMST is a second-level scheduler (i.e. application-level scheduler). The middleware scheduler uses the *round-robin* scheduling. The VMST runs every 9 ms, at which time it can select and invoke a user application thread.

The VMST selects an SpM or SvM from a Ready-Application-Thread-Queue using an *earliest deadline first* algorithm. There are two more application queues which contain SpMs (i.e. SpM-Reservation-Queue) and SvMs (i.e. Waiting-SvM-Thread-Queue). SpMs in the SpM-Reservation-Queue are examined periodically by the WTST and moved into the Ready-Application-Thread-Queue at appropriate times. In contrast, when receiving service request messages, the MMCT (Middleware Message Communication Thread) identifies the requested SvM and then inserts it into the Waiting-SvM-Thread-Queue. If the SvM invocation passes the BCC (see Section 2.1) check, it is moved to the Ready-Application-Thread-Queue. Therefore, an SpM is triggered by the WTST (which is in turn triggered by a real-time interrupt), and an SvM is triggered by an incoming message.

3. TIMING PERFORMANCE OF THE CURRENT TMO IMPLEMENTATION

Before using the TMO model to create our airplane-landing simulator, the performance of the current TMO implementation was tested to ensure that it would satisfy our needs. All experiments were performed on a cluster of Intel Pentium III 933 MHz dual-CPU server PCs with 256 MB of main memory each and a 100 Mbps fast Ethernet network connection. During our experiments, there was no other user-level system load or any other network load. The operating environment used in the cluster was Microsoft Windows 2000 professional with TMOSL and TMOSM/NT.

Table II. Maximum number of schedulable SpMs in a node.

Period (ms)	16	30	50	75	100	125
Number of SpMs	2	4	5	9	9	13

3.1. Maximum number of schedulable SpMs in a node

At the initial phase of this investigation, we thought that it would be useful to find out how many SpMs could run on a node. To test this boundary, we increased the number of SpMs which run on a node until TMOSM/NT was unable to schedule SpMs properly. The results are shown in Table II. The minimum period that TMOSM/NT could support in the node machine configuration adopted turned out to be about 16 ms. TMOSM/NT uses the time slice of 3 ms and the VMST runs every 9 ms under the two-level scheduling scheme adopted. An SpM can run only when the VMST gets a time slice. With a more powerful node and network configuration which allows smaller time slices, the minimum period that can be supported becomes smaller. Therefore, Table II starts at 16 ms. The table shows that the maximum number of SpMs supported is closely related to the SpM period. As this period is increased, the number of schedulable SpMs is also increased. To reduce the overhead of scheduling, the current implementation of TMOSM examines a 200 ms time window in advance to fill up the SpM-Reservation-Queue (refer to Section 2.2), and the maximum number of items is 30. Therefore, the number of simultaneously executable SpMs is limited by the queue size and time window. The notation '*N/A (not available)' in Tables III and IV shown in the next section indicates a situation in which the system could not execute because there were too many SpMs for that time period.

3.2. Timing performance of SpMs within a single node

SpMs are often specifications of periodic real-time computations in the TMO model. In executing each iteration of an SpM, it is desirable to keep the difference between the actual and desired execution start time to a minimum. The desired execution start time is the earliest possible execution time in that period. As stated in Section 2.2, TMOSM uses an earliest-deadline-first algorithm to select the next running SpM (i.e. it is a user-level scheduling algorithm). Therefore, it is desirable for a periodic real-time task to run at the beginning of its period. Experiments were performed by using simple SpMs, which just read the elapsed time from the beginning of the TMO application till the printing to the console.

Table III shows the experiment results that involved one TMO with several SpMs on a single node. It shows the minimum, maximum, average, and standard deviation of the difference between the actual execution start times and the desired execution start times of a single SpM. This result was obtained during 10 000 iterations on a single node. The first column, labeled period, refers to the period of the SpM, i.e. the time interval between consecutive iterations of that SpM. Within a single node, time differences were measured by using the Pentium timer, resulting in highly accurate

Table III. Difference between actual and desired execution start times of SpMs in a node (in μs).

Period	Number of SpMs	Minimum	Maximum	Average	Standard deviation
16 000	1	33	11 747	5859	3385
	2	39	11 821	5895	3383
	3	*N/A	*N/A	*N/A	*N/A
	4	*N/A	*N/A	*N/A	*N/A
30 000	1	33	11 744	5845	3391
	2	38	13 118	5830	3390
	3	43	12 078	5966	3391
	4	50	12 053	5983	3383
50 000	1	110	11 744	5801	3403
	2	42	11 878	5795	3404
	3	61	11 799	5868	3314
	4	54	11 935	5778	3366
75 000	1	36	11 757	5771	3384
	2	42	11 825	5866	3333
	3	46	11 870	6189	3374
	4	51	11 936	5995	3394
100 000	1	39	11 762	5522	3360
	2	43	11 831	5487	3343
	3	52	11 901	5679	3401
	4	53	11 975	5612	3355

timing measurements. '*N/A' in the table represents a case where experiments were not possible, as already explained in Section 3.1.

Table III shows that the average difference is about 5.5 ms, and as expected, there is no relation between the period of an SpM and the observed difference in execution start times. If a TMOSM implementation is perfect, then the difference should be zero. An increase in this difference means that the beginning of an SpM execution is delayed, thus lowering the timing precision of real-time tasks. Therefore, the average values should be small. For an application which has a period of at least a few tens of milliseconds, a difference of a few milliseconds does not cause any problems, since a real-time periodic task simply needs to be run once during each iteration period. As mentioned earlier, the VMST runs every 9 ms. This factor clearly has major impacts on the average difference between the actual and the desired execution start times.

Table III shows that the minimum difference value is about 40 μs, the variance is about 3.3 ms, the average is about 5.5 ms, and the maximum value is about 12 ms. The differences are unpredictable but bounded below 13 ms in all cases. This implies that the current TMOSM implementation is not suitable

Table IV. Differences in execution start times of SpMs distributed among multiple TMOs in a node (in μs).

Period	Number of TMOs	Minimum	Maximum	Average	Standard deviation
16 000	1	33	11 747	5859	3385
	2	43	12 113	5914	3387
	3	*N/A	*N/A	*N/A	*N/A
	4	*N/A	*N/A	*N/A	*N/A
30 000	1	33	11 744	5845	3391
	2	4	13 156	5945	3385
	3	45	13 229	5947	3387
	4	50	11 934	5888	3387
50 000	1	110	11 744	5801	3403
	2	43	11 809	5770	3370
	3	46	11 862	5820	3368
	4	54	11 955	5793	3369
75 000	1	36	11 757	5771	3384
	2	40	11 823	5358	3367
	3	47	11 884	5889	3331
	4	50	11 918	6286	3372
100 000	1	39	11 762	5522	3360
	2	49	11 811	5551	3331
	3	48	11 888	5645	3394
	4	56	11 946	5636	3370

for applications with a very low simulation time unit (with a resolution less than 10 ms) because a real-time task may not always begin its execution before its extremely short deadline expires. However, since the maximum difference value still remains under 13 ms, the current TMOSM implementation is suitable for applications with a required time resolution of at least a few tens of millisecond (which is the case with our airplane-landing simulator).

Table IV shows the results of using multiple TMOs, with a single identical SpM in each TMO, on a single node. As expected, the results are only dependent on the number of SpMs in an application, and not on the number of TMOs. These results show that the difference between the execution start times of identical SpMs in different TMOs is as expected.

To check the performance trend of the current TMOSM implementation, the number of SpMs executed on a single node was increased and the resulting differences in execution start times measured. These results are shown in Figure 4. The measured values are shown with a solid curve, and the asymptotes of the measured values are shown with a dashed curve. As can be seen, the execution start time differences increase only slightly with increasing numbers of SpMs up to 13 SpMs.

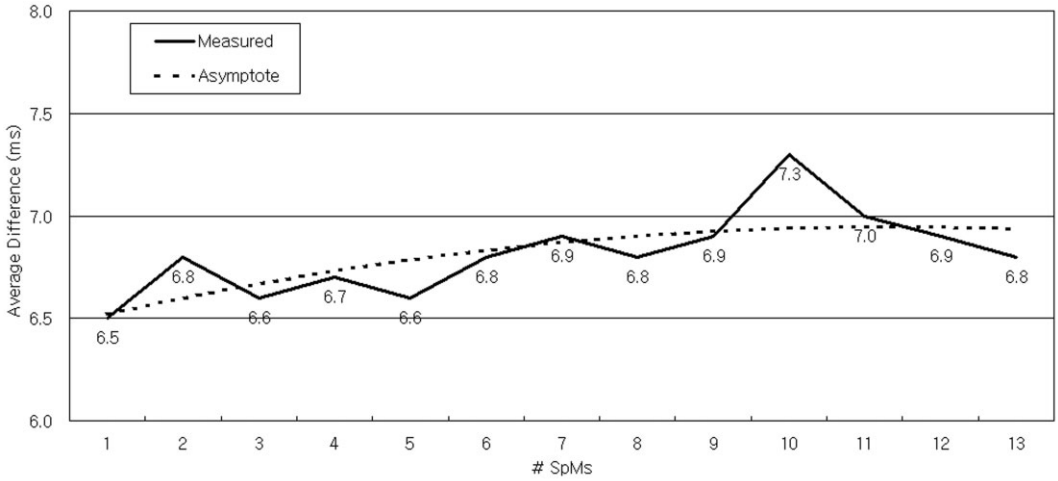

Figure 4. Differences in execution start times of SpMs assigned to a single node.

3.3. Inter-node timing differences

The TMO model provides support for a distributed computing environment; that is, TMOs can be distributed among multiple nodes. Thus, any timing differences among the different nodes in execution become an important issue. Based on the global clock maintained by the TMOSM, objects in different nodes should exhibit the same relative timing behavior as when those objects are located on the same node. To check the accuracy of this assumption, the difference in execution start times between identical TMOs assigned to multiple nodes was measured. The results are shown in Figure 5 in detail and in Figure 6 in a summarized form.

TMOSM incorporates a global clock synchronization scheme in order to efficiently support TMOs distributed across multiple nodes. Every node has its own system clock, typically based on a crystal oscillator. However, crystal oscillators have a tendency to drift slightly, and no two oscillators are perfectly synchronized. Thus, even if all nodes are initially synchronized, the system clocks of different nodes can become unsynchronized after a period of time.

TMOSM deals with this problem using a periodic re-synchronization scheme. After every second, the WTST in the master node sends a broadcast message to worker nodes requesting clock synchronization. Then, worker nodes which receive this message calculate clock differences, taking into consideration the communication delay (measured during the initialization stage), and update their own clocks. As stated in [11], the maximum difference between the master node and worker nodes in the current TMOSM implementation is bounded by about 250 μs. Thus, the global clock can be considered to be accurate to within about 500 μs. With a more powerful network configuration in terms of delay and jitters, a global time base of a higher precision can be established.

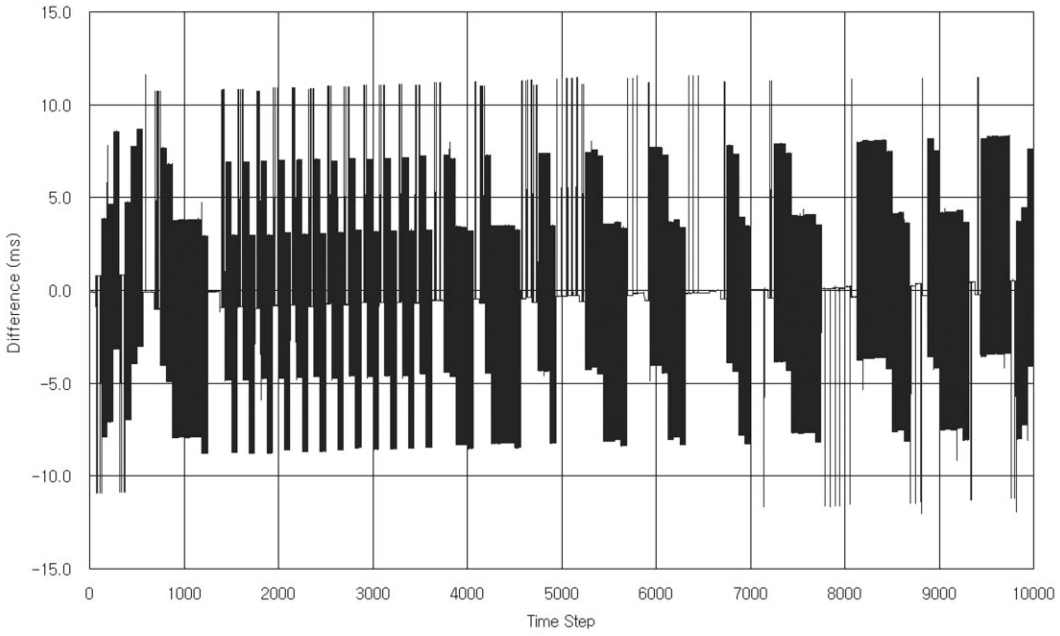

Figure 5. Differences in execution start times of two SpMs assigned to two different nodes.

The difference in execution start times between two identical TMOs assigned to two nodes is shown in Figure 5. The horizontal axis corresponds to the simulation time step, and the vertical axis is the difference in execution start times between the two nodes in microseconds. The graph shows the results of 10 000 iterations of an SpM with a 16 ms period. The result does not depend on its period, i.e. a graph for a different period shows similar timing behavior. The average difference observed during the two-node experiment is about 3.5 ms and the standard deviation is about 2.8 ms. The maximum difference is bounded within about 13 ms. The result shows that the timing precision of SpMs distributed among multiple nodes is in an acceptable range when the iteration period of every SpM in the target application is much larger than 13 ms.

Figure 6 shows the average differences in execution start times of SpMs distributed among multiple nodes. Ideally, this value should be zero. With increasing numbers of nodes, the actual value was found to increase and then reach a plateau at around 7.1 ms. In all cases, the maximum difference values were found to be around 12 ms. Thus, the current TMOSM implementation should be scalable to medium-sized clusters as long as timing accuracies of a few tens of milliseconds are sufficient.

If the number of nodes grows much beyond a dozen, the average differences in execution start times of SpMs will not be affected much unless the precision of the global time base realized through synchronization of distributed clocks deteriorates significantly. During clock synchronization, the system normally operates with minimal message traffic. A more serious performance degradation will

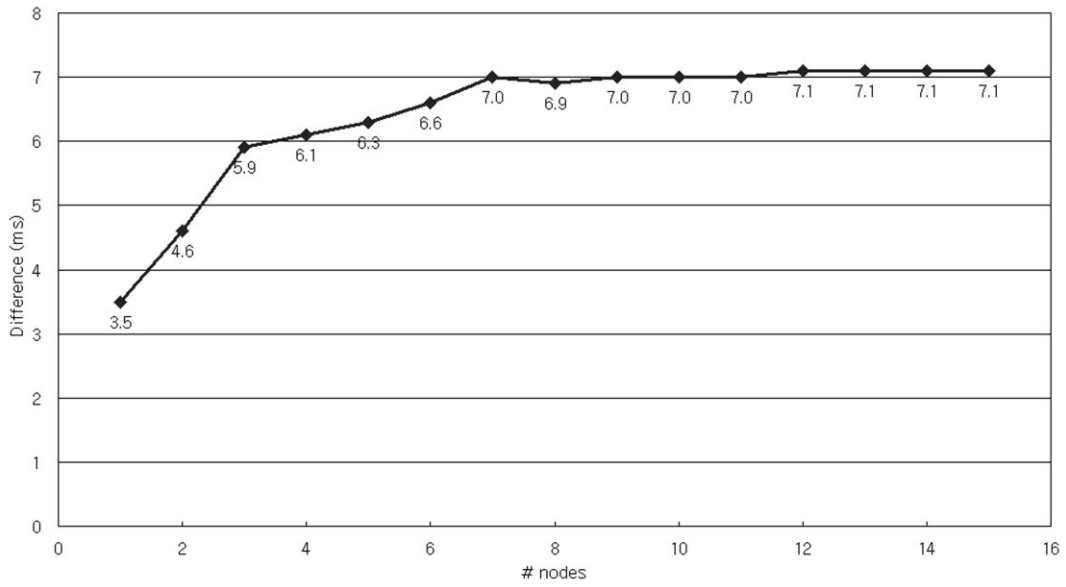

Figure 6. Average differences in execution start times of SpMs distributed among multiple nodes.

show up in the form of increased average latency in message communication among the nodes and increased overheads in executing remote SvM calls.

4. APPLYING THE TMO MODEL

4.1. The target application

In this work, we implemented an airplane-landing simulator to investigate how the TMO model can be applied to a real application in a distributed computing environment. As mentioned above, the airplane-landing simulator is representative of applications that require non-scaled real-time computations.

4.2. TMO modeling of the target system

Figure 7 shows a model of the target system. First of all, we defined a three-dimensional space, which provides the environment for the actual simulation. There must be an airplane, referred to as MyAirPlane, which is controlled by the user, performing the role of the pilot. Also, there may be other airplanes, referred to as TheOtherAirPlanes, in our simulation space. Finally, there need to be hangars, runways, and a control tower.

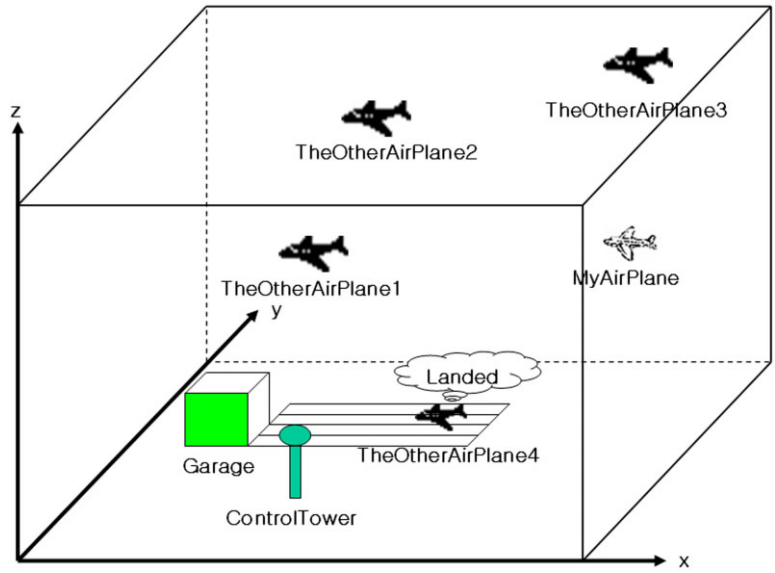

Figure 7. Simulation environment of the airplane-landing simulator.

A top-down methodology was used to develop the TMO model for our simulation. First, we modeled the entire system using one TMO. Then, we extracted one object from this original model and created a separate TMO for it. This process was repeated until sufficiently detailed logic was incorporated and the TMO network appeared to be sufficiently modularized for our application. The resulting system consists of the four TMOs described below.

(1) MyAirPlaneTMO: an airplane TMO which is controlled by the user. This object has three types of information: (1) the state of the airplane such as velocity, position, etc.; (2) the properties of the airplane such as size, maximum speed, etc.; (3) user controls such as rudder, flaps, spoilers, etc. Also, this object must communicate with the control tower, referred to as ControlTowerTMO. There must also be mechanisms for requesting the runways, receiving alarm messages (about to crash with another airplane, etc.), getting weather information, etc.
(2) SpaceTMO: this object represents the three-dimensional space, which provides the environment for our simulation. This model has all the information of the environment pertaining to the simulation. Also, this object has information on TheOtherAirPlanes, and an SpM of this object updates the state of TheOtherAirPlanes. MyAirPlaneTMO periodically informs SpaceTMO of its position so that the latter may detect occurrences of collisions between MyAirPlane and other airplanes.
(3) ControlTowerTMO: this TMO represents the control tower. This object monitors and regulates the flight, landing, and take-off of all airplanes in the simulated space. This TMO always checks

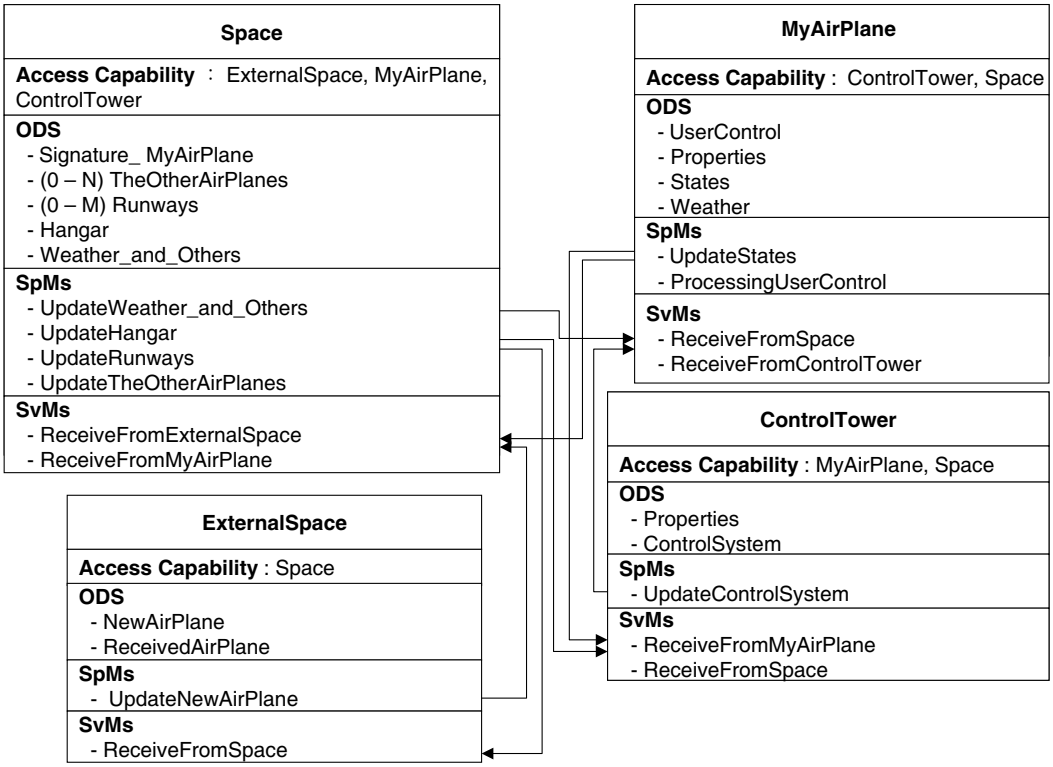

Figure 8. The relation among the TMOs in our simulator.

the space, and sends an alarm to MyAirPlaneTMO when the probability of a crash is increased beyond a certain threshold. It also processes requests from all airplanes such as landing-runway requests, etc.

(4) ExternalSpaceTMO: this TMO models the creation and termination of TheOtherAirPlanes in our simulation space. Although airplanes are not actually created and destroyed, such actions model what happens when airplanes enter or leave the three-dimensional simulated space.

Figure 8 shows the relation among the TMOs in our model. Each TMO contains its name, the access capabilities of the TMO, the object data store (ODS), spontaneous methods (SpMs), and service methods (SvMs). The ODS contains a list of the data objects to be maintained by that TMO. The SpM and SvM fields show a list of the spontaneous (real-time-clock-triggered) and service methods (called by client objects), appropriately named so that the function of each method is self-evident. The arrows in the figure show the direction of messages in the model.


```
class MyAirPlaneTMO_ODSS : public ODSSBaseClass
{
private :
    int m_fuel;
    struct cartesian_vector m_vel;
    struct  cartesian_vector    m_pos;
    struct  flaps       m_flaps;
    struct  spoilers    m_spoilers;
    struct  size    m_size;
    int     m_rudder,   m_landinggear,  m_mileage;
    int     m_weight, m_maxspeed;
    struct  balances    m_balance;
    struct  engineThrotle   m_enginethrotle;
    struct  weather m_weather;
    void    setDefault();
    int m_AlarmReceived;
```

Figure 9. A part of ODS of MyAirPlaneTMO.

5. IMPLEMENTATION

5.1. Implementation of an airplane-landing simulator

It is much easier to implement a sizable real-time distributed computing application by using the TMO programming scheme than by using conventional lower-level programming schemes. TMOSL, the TMO support library, contains a set of base classes that can be inherited for the creation of object data stores (ODSs), SpMs, SvMs and TMO frames. As an example, Figure 9 shows a part of the ODS for MyAirPlaneTMO. This ODS contains the variable m_pos, used to represent the position of MyAirPlane in the three-dimensional simulated space, m_vel, used to represent its velocity, and other variables used to represent the current state of MyAirPlaneTMO. The variables in an ODS are accessed and updated by SpMs or SvMs in the TMO of which the ODS is a part.

Figure 10 shows the class definition for an SpM that is a member of MyAirPlaneTMO. This SpM is named MyAirPlaneTMO_UpdateState_SpM, and it runs in a periodic manner. Naturally, a model of a real system can contain several SpMs, with each SpM possibly having its own period. However, better execution performance can be realized if all SpMs for a particular TMO are merged into a single SpM in the case where each node is a single-CPU machine. The model of the airplane-landing simulator also originally had several SpMs in each TMO. However, for performance reasons, all SpMs in a given TMO application were merged into one since the nodes in our initial experimental network configuration were to be single-CPU machines. The function of a particular SpM can be inferred from its name and its description in the design chart. SpMs are used to update the state of each airplane, process user inputs, send messages to other TMOs, etc. To communicate with other TMOs, SpMs use gate objects. Thus, for example, MyAirPlaneTMO_UpdateState_SpM uses one gate object to communicate with SpaceTMO and another gate object to communicate with ControlTowerTMO. Although not shown here, the gate object for the ReceiveFromMyAirplane in SpaceTMO is declared in one statement which is an instantiation of the standard TMO Gate Class, with the names of the TMO (SpaceTMO) and the SvM (ReceiveFromMyAirplane) as parameters.

```
class MyAirPlaneTMO_UpdateState_SpM : public SpMBaseClass
{
private :
    MyAirPlaneTMO_ODSS  * m_MyAirPlaneTMO_ODSS;
    TMOGateClass     *
        m_gate_MyAirPlaneTMO_UpdateState_SpM_to_SpaceTMO_ReceiveFromMyAirPlane_SvM;
    TMOGateClass *
        m_gate_MyAirPlaneTMO_UpdateState_SpM_to_ControlTowerTMO_ReceiveFromMyAirPlane_SvM;
    struct ParamStruct_FromMyAirPlane_to_Space mp_toSpace;
    struct ParamStruct_FromMyAirPlane_to_ControlTower mp_toControlTower;
public :
    MyAirPlaneTMO_UpdateState_SpM(const char  * SpM_name, TMOGateClass  &gate1,
        TMOGateClass &gate2, MyAirPlaneTMO_ODSS & odss, access_mode_type mode);
    ~MyAirPlaneTMO_UpdateState_SpM();
    virtual void SpMBody();
};
```

Figure 10. Definition of SpM of MyAirPlaneTMO.

```
class MyAirPlaneTMO : public TMOBaseClass
{
private :
    MyAirPlaneTMO_ODSS  m_MyAirPlaneTMO_ODSS;
    MyAirPlaneTMO_UpdateState_SpM
        m_MyAirPlaneTMO_UpdateState_SpM;
    MyAirPlaneTMO_ReceiveFromSpace_SvM
        m_MyAirPlaneTMO_ReceiveFromSpace_SvM;
    MyAirPlaneTMO_ReceiveFromControlTower_SvM
        m_MyAirPlaneTMO_ReceiveFromControlTower_SvM;
public :
    MyAirPlaneTMO(const char * TMO_name, TMOGateClass &,
        TMOGateClass &, tms & TMO_start_time);
};
```

Figure 11. Definition of MyAirPlaneTMO class.

Figure 11 shows the definition of MyAirPlaneTMO. This class has the single SpM corresponding to the two SpMs shown in Figure 8 and two SvMs that have remained intact. The function of each of the SpMs and SvMs can be inferred from its name. MyAirPlaneTMO_ODSS is the ODSS class for MyAirPlaneTMO and MyAirPlaneTMO_UpdateState_SpM is the periodic method described above. MyAirPlaneTMO_ProcessingUserControl_SpM is an SpM that processes the key inputs from users. Other ODSs, SpMs, SvMs, and TMO frames are implemented similarly.

5.2. User input/output interface

The user input/output interface consists of a keyboard input interface and a graphic output interface. This graphic user interface is implemented as a separate non-TMO process by using the Microsoft MFC library. The graphic user interface communicates with TMOs by using the UDP. Graphical processing

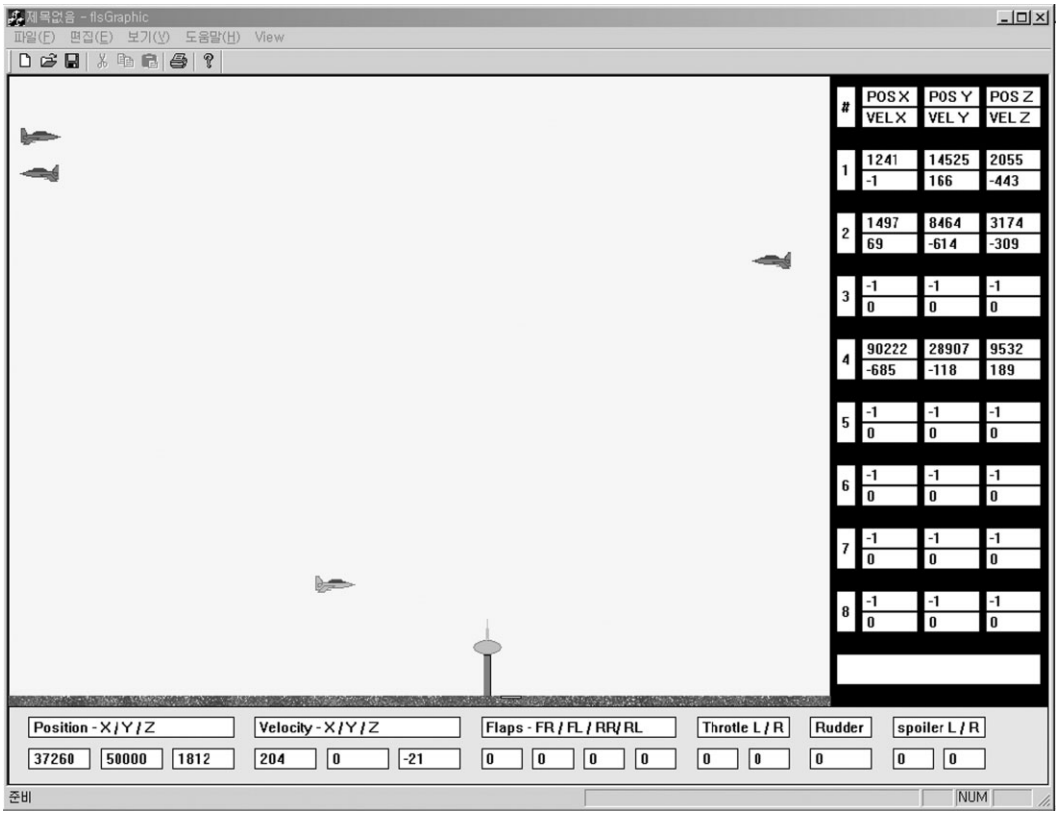

Figure 12. Screenshot of the user interface program.

functions can consume a lot of CPU processing power. Thus, for performance reasons, the graphical interface for a TMO application is typically executed on a processing node different from the node that executes the TMO. Figure 12 shows the graphic user interface for the airplane-landing simulator.

5.3. Implementation on a distributed computing environment

In order to adapt an application configured on a single-node system to a multi-node system configuration, the application developer can distribute the TMOs to the desired number of nodes in fairly simple manners. The TMO support middleware provides support for communication between TMOs located on different nodes (or on the same node) in a manner to the application programmer transparent. Although a TMO can technically be split across multiple nodes, the current implementation

Table V. Differences between actual and desired execution start times (in μs) for an SpM in the airplane-landing simulator.

Number of nodes	Minimum	Maximum	Average	Standard deviation
4	40	14 091	5870	3381

permits this type of split only on a machine supporting shared memory. In distributed computing environments where there are more processing nodes than TMOs, the application designer can check if it is worth splitting each of these TMOs with heavy computational requirements into a number of smaller TMOs.

The current version of the airplane-landing simulator has four TMOs. Thus, it can be executed on a cluster of one to four nodes (not counting the node executing the graphic user interface). If it is determined that a larger cluster must be used (in order to provide better performance), then four TMOs need to be restructured into a larger number of TMOs. For instance, in order to exploit parallelism within SpaceTMO, it can be divided into several TMOs, each covering a separate non-overlapping portion of the three-dimensional simulated space. Splitting the simulated space into several TMOs is a frequently used method for distributing work to multiple nodes.

6. PERFORMANCE OF THE TMO APPLICATION

Various performance measurements were taken with our TMO-based airplane-landing simulator. First, we measured the difference between the actual and desired execution start times of an SpM. Next, we measured the difference in execution start times of SpMs on different nodes that are supposed to execute at the same time. Lastly, we measured the shortest simulation time step that could be used with our application while meeting all real-time deadlines. A shorter simulation time step implies a more accurate simulation result, since calculations of position, velocity, and other attributes are performed more frequently.

6.1. SpM timing accuracy

Table V shows the differences between actual and desired execution start times for an SpM in our airplane-landing simulator. As mentioned above, our airplane-landing simulator has four TMOs, and each has its own set of one to two SpMs and several SvMs. The results shown in Table V are similar to those shown in Table III, which showed the analogous results for a minimal TMO-based system. The maximum value is still around 14 ms, and the average value is around 5.5 ms. These results show that TMOSM/NT can be used effectively for real-time applications with timing accuracy requirements of around 14 ms or more.

Table VI. Differences in execution start times (in μs) for SpMs distributed among multiple nodes in the airplane-landing simulator.

Number of nodes	Minimum	Maximum	Average	Standard deviation
4	2943	11 590	6071	1908

Table VII. Minimum simulation time step possible with the airplane-landing simulator.

Number of nodes	1 node	2 nodes	4 nodes
Minimum period	30 ms	20 ms	18 ms

6.2. Inter-node timing differences

Table VI summarizes the time measurements of the differences in execution start times for SpMs assigned to different nodes in the airplane-landing simulator. Because the current simulator implementation only uses four TMO objects, the maximum number of computing nodes that can be used is four. This table contains results that are similar to the results shown in Table IV, which measured the same values in a minimal TMO-based system. Since the target simulation period of our simulator is a few tens of milliseconds, an average difference of 6 ms is acceptable.

6.3. Simulation performance and number of nodes

An important metric of performance in a real-time simulation system is the smallest time unit that can be used while continuing to meet all required deadlines. This is especially important for scaled real-time simulation. However, even for non-scaled real-time simulation, the use of the minimum simulation time step possible is desirable since it results in accurate and detailed simulation. Table VII shows the minimum simulation time step possible with our airplane-landing simulator on cluster systems with one to four nodes. Each result shown is the minimum value possible in 10 000 trials (out of which none missed their deadlines). The amount of computation performed by each SpM is the primary factor in determining this minimum simulation time step. For improved performance in a larger cluster system, SpaceTMO could be split into a number of smaller TMOs, thus distributing the work required in one SpM of the original SpaceTMO.

Empirical studies [12] have shown that in first person shooter games smooth play can be realized if the response time is in the range of 50–300 ms. Such response time can be realized by use of SpMs with iteration periods at the level of a few tens of milliseconds. In most interactive real-time applications such iteration periods are sufficient as they are in our airplane-landing simulator.

7. CONCLUSION

In recent years, distributed *time-triggered* simulation has emerged as a viable approach for distributed real-time simulation. This type of simulation approach can be implemented efficiently by using the TMO specification and programming scheme, which enables high-level high-precision real-time distributed computation programming, and provides support tools based on Windows NT, named TMOSM/NT. Measurements of the real-time performance of TMOSM/NT (whether deadlines were met and whether methods executed exactly when they were supposed to execute) were taken for a minimal TMO-based system and a non-trivial TMO-based application. The experimental results showed that TMOSM/NT was able to support applications subject to timing precision requirements at the level of about 12 ms. (Other implementations of the TMO support tools, such as Linux implementations which involve kernel level extension can support even tighter timing accuracies.)

The TMO model was used to implement a demanding non-scaled real-time simulation, an airplane-landing simulator. Because of the real-time object-oriented middleware support provided by the TMO scheme, the development of the airplane-landing simulator was completed in about 4 months. This included the time required to learn the TMO model for the first time; in addition, about half of the 4-month period was spent on the graphic user interface. Moving the application from a single-node system to a four-node system was a simple matter of moving the four TMOs into separate nodes. Communication, synchronization, coordination of tasks hosted on different nodes, etc. were all taken care of by the TMO middleware. Modification was also simple because of the object-oriented and encapsulated nature of the TMO model. Finally, the end product displayed more-than-satisfactory real-time performance for our application, and scaled fairly well from one node to four nodes.

ACKNOWLEDGEMENTS

This research was supported in part by the Korean Ministry of Information & Communication through its University Fundamental Research Program, grant 2001-051-3. The research conducted at UCI was supported in part by the NSF under grant numbers 00-86147 and 02-04050 (NSG).

REFERENCES

1. Kim KH, Nguyen C, Park C. Real-time simulation techniques based on the RTO.k object modeling. *Proceedings of COMPSAC '96*, Seoul, August 1996; 176–183.
2. Ishikawa Y, Tokuda H, Mercer CW. An object-oriented real-time programming language. *IEEE Computer* 1992; **25**(10):66–73.
3. Ellenberger R, Ling R, Buscher D, Uhde-Lacovara J, Shuler R. Automatic generation of real-time Ada simulation for space station freedom. *Simulation* 1993; **65**(5):337–345.
4. Fujimoto RM. Parallel discrete event simulation. *Communications of the ACM* 1990; **33**(10):30–53.
5. Zeigler B, Kim J. Extending the DEVS-scheme knowledge-based simulation environment for real-time event-based control. *IEEE Transactions on Robotics and Automation* 1993; **9**(3):351–356.
6. Kim K, Liu J, Ishida M, Kim I. Distributed object oriented real-time simulation of ground transportation networks with the TMO structuring scheme. *Proceedings of COMPSAC'99*, Phoenix, AZ, October 1999; 130–138.
7. Kim K. Object structures for real-time systems and simulators. *IEEE Computers* 1997; **30**(9):62–70.
8. Kim K. Real-time object-oriented distributed software engineering and the TMO scheme. *International Journal of Software Engineering and Knowledge Engineering* 1999; **9**(2):251–276.
9. Kim KH, Subbaraman C, Kim Y. The DREAM library support for PCD and RTO.k programming in C++. *Proceedings of WORDS '96*, Laguna Beach, February 1996; 59–68.

10. Kim KH, Ishida M, Liu J. An efficient middleware architecture supporting time-triggered message-triggered objects and an NT-based implementation. *Proceedings of ISORC '99*, May 1999; 54–63.
11. Kim KH, Im C, Athreya P. Realization of a distributed OS component for internal clock synchronization in a LAN environment. *Proceedings of ISORC 2002*, Washington, DC, April 2002; 263–270.
12. Henderson T. Latency and user behavior on a multiplayer games server. *Proceedings of NGC 2001*, UCL, U.K., November 2001; 1–13.

