



# The Distributed Time-Triggered Simulation Scheme: Core Principles and Supporting Execution Engine

K. H. (KANE) KIM  
University of California Irvine, CA, USA

khkim@uci.edu

**Abstract.** Distributed real-time simulation is a young technology field but its practice is under increasing demands. In recent years the author and his collaborators have been establishing a new approach called the distributed time-triggered simulation (DTS) scheme which is conceptually simple and easy to use but widely applicable. The concept was initiated in the course of developing a new-generation object-oriented real-time programming scheme called the time-triggered message-triggered object (TMO) programming scheme. Some fundamental issues inherent in distributed real-time simulation that were learned during recent experimental studies are discussed along with some approaches for resolving the issues. An execution engine developed to support both the TMOs engaged in control computation and the TMOs engaged in DTS is also discussed along with its possible extensions that will enable significantly larger-scale DTSs.

**Keywords:** real-time, embedded, clock, tick, simulation, update, dependency, DTS, TMO, time triggered, message, object, middleware, distributed, parallel, programming

## 1. Introduction

Object-oriented (OO) real-time (RT) distributed computing has become a rapidly growing branch of computer science and engineering (IEEE, 2000; ISORC 1998–2001; OMG, 2000; WORDS 1994–2001). Among several branches of this young OO RT distributed computing field, a less developed branch which possesses huge potential application areas is *OO distributed RT simulation*. Here *RT simulation* refers to the accurate mode of simulation in which the simulator components (or *simulator objects*) show the timing behavior that are the same as or similar to the timing behavior of the simulation targets (i.e., simulated entities).

The demands for RT simulator developments are steadily increasing (Mathworks, 2002; Otter and Elmqvist, 2001; Papini and Baracos, 2000). Next-generation virtual reality applications require more powerful RT simulation capabilities. RT computer control applications such as those found in manufacturing plants also require such capabilities. Simulation of application environments is often performed as integral steps of validating control computer system designs (Ellenberger et al., 1993; Guyse et al., 1994; Kim, 1997; Papini and Baracos, 2000; Zeigler and Kim, 1993). Here RT simulators of application environments can often enable highly cost-effective testing of the control computer systems implemented. Such testing can be a lot cheaper than the testing performed in actual application environments while being much more effective than the testing based on non-RT simulators of environments (Kim et al., 1996).

As the complexities of RT simulators grow, the use of distributed/parallel RT simulation approaches becomes imperative (Ozard and Desira, 2000; Papini and Baracos, 2000). However, effective decomposition of a simulation model for parallel RT execution is not a well-understood subject at this time. A fundamental issue related to this will be discussed in some detail in this paper. A good RT simulation modeling approach should yield an easy analysis procedure that can be applied in judging the suitability of various conceivable decompositions of a monolithic RT simulation model. Therefore, practical distributed RT simulation techniques have not been established in sufficiently reliable forms for industrial use although a few special cases have been handled at a reasonable level.

In fact, even in cases of RT simulation where a single computing node is used, there is a lot to be desired in the industrial state of the art. A approach to model the simulation targets, especially those which enable *multi-fidelity representation of the timing behavior of the simulation targets*, is highly desired.

Since the early 1990s this author and his collaborators have been establishing a new-generation OO RT programming scheme called the *time-triggered message-triggered object* (TMO) programming scheme (Kim et al., 1994, 1997, 2000). The TMO is a natural, syntactically minor, and semantically powerful extension of the conventional object(s). In the course of developing an RT system engineering methodology centered around this TMO programming scheme, a new approach to RT simulation which is conceptually simple and easy to use but widely applicable, has also been born (Kim et al., 1994, 1996, 1999b). This approach called the *distributed time-triggered simulation* (DTS) scheme is an attractively simple approach to parallel and distributed RT simulation. Simulation targets are modeled as TMOs and the simulation engine based on a parallel or distributed computing platform faithfully executes TMO models to effect RT simulation. Therefore, the TMO is capable of representing uniformly and with variable degrees of precision both RT embedded computer systems and their application environments.

The key idea of DTS is to let distributed simulator nodes, or more generally, simulator objects, advance to the next simulation step simultaneously when the globally available RT clock (Kopetz, 1997) reaches a certain time-point. Costly message exchanges are required in previously studied distributed simulation approaches in order to keep the simulator nodes properly synchronized in advancing to the next simulation step. Such message exchanges are obviated in DTS. The essential data flow among distributed simulator objects is of course facilitated via inter-object communication mechanisms, e.g., remote method calls or logical multicast channels connecting distributed objects.

We have performed several experiments in recent years (e.g., Kim, 1997; Kim et al., 1999b, etc.) to validate the TMO scheme for specification, design, and implementation of RT embedded computer systems as well as the DTS scheme. Two good examples of these experiments are the one based on a defense command-control application scenario and the other based on an advanced freeway traffic control scenario. These experiments reinforced our belief that the TMO model had the necessary representational power and also offered an efficient and rigorous way to develop complex RT systems and their application environment simulators. The experiments are repeated as new enhancements continuously occur in the TMO programming methods and styles, the TMO execution engine architecture and implementation techniques, and the DTS techniques.

Through the recent experiments, we also learned some major design principles and implementation techniques useful in constructing efficient DTS object networks. One of the main goals of this paper is to discuss a major part of these principles and techniques.

In addition, we needed to establish an architecture and a prototype implementation of an execution engine capable of executing TMOs. The execution engine has the form of the middleware running on a commercial computing platform equipped with hardware and an operating system (OS) kernel. The middleware architecture has been named the *TMO support middleware* (TMOSM) (Kim et al., 1999a). This has also been evolving as new optimization techniques have been discovered. The TMOSM supports both application computing TMOs and environment simulator TMOs. Some major techniques incorporated into this TMOSM as well as those extensions desirable to future RT simulator developers are also discussed in this paper.

In the following section, major challenges in developing distributed RT simulation technologies are discussed along with the core underlying concept of the DTS scheme. Section 3 provides an overview of the TMO programming scheme and Section 4 presents the basic framework of the TMO-structured DTS scheme. Then in Section 5, some fundamental issues arising in realizing effective RT simulators based on the DTS scheme as well as some resolution techniques are discussed. The TMOSM architecture and some major implementation techniques are discussed in Section 6. The paper concludes in Section 7.

## 2. Main Challenges in Distributed RT Simulation and the DTS Framework

In a computer-based simulation, an integer called the *simulator clock* is used and each new incremented value of the simulator clock drives new simulation activities (a new *simulation step*). Since an RT simulator must exhibit the timing behavior which is the same as or very close to that of the simulation target, the simulator clock must “tick” at a steady rate. Each tick of the simulator clock is commenced and administered by referencing an RT clock in the *simulation execution engine* (a computer system running the simulation program). All computational activities taking place during a *ticking interval* of the simulator clock may be viewed as one simulation-step. The *ticking rate* of the simulator clock in an RT simulator must be chosen with the following understanding:

For all (real-time) events which should be simulated during any ticking interval of the simulator clock, the exact microscopic timings of those events may be transparent to the user of the RT simulator.

Only the resulting state of the simulator at the end of the ticking interval may be seen by the user. That is, only the fact of whether each such event occurred or not at some (unknown) point during the interval should be of interest to the user.

To be more general, consider two different executions of the same simulator. In Execution I, the ticking rate of the simulator clock is  $\alpha$  and in Execution II, it is  $n * \alpha$ , where  $n$  is an integer greater than 1.

Then the computational complexity of the algorithm executed at a tick of the simulator clock to simulate the events occurred between the most recent tick and the second most recent tick in Execution I, is much less than the combined computational complexity of the corresponding  $n$  algorithm executions needed in Execution II to simulate the events occurred during the same period.

As an illustration of the relationship between the ticking rate and the simulation precision, consider the case of simulating cars moving on a freeway. Since the moving pattern of each car is affected by those of the cars moving in the neighborhood area, the selection of the ticking interval of the simulator clock is an important matter. If the ticking interval is chosen to be 5 seconds, then the state of every car will be updated every 5 seconds. However, if a typical car can change its lane in 1 second, and a lane change by a car can have a big impact on the subsequent behavior of some other cars, updating the state of all cars every 5 seconds means a very low-fidelity, low-precision simulation.

To be more specific, assume that car collisions involving a car changing a lane are to be dealt with. In the real world, the state of the freeway at 5 seconds past 10 am may be drastically different between the case where a collision occurs at 0.1 seconds past 10 am and the case where a collision occurs at 4.9 seconds past 10 am. In the former case, the collision can impact the behavior of the cars in collision and the nearby cars during the remaining 4.9 seconds. Yet, due to the complexity constraint, a typical simulation algorithm used may not reflect such drastic differences. A typical algorithm would update the state of each car once, or a small number of times, during one simulation step.

On the other hand, a more precise simulation of cars with a large ticking interval would often lead to a situation where a car is *re-simulated* a large number of times. The reasons are as follows. Typically, a new position of a car, say  $Z$ , valid at 5 seconds past 10 am is first calculated (i.e., simulated) with the assumption that  $Z$  will not be affected by any collision. Then a collision in a nearby location occurring at 0.1 seconds past 10 am is “discovered” later within the current simulation step and this new knowledge is used to re-simulate car  $Z$  in order to obtain a more reasonable estimate of the position which  $Z$  will occupy at 5 seconds past 10 am. Car  $Z$  may go through multiple re-simulations during the current simulation step.

These multiple re-simulations of cars during one simulation step can degrade the performance of the simulator down to an unusable level. Therefore, in general, multiple re-simulations of the same simulation target in one simulation step must be avoided. This then leads the simulator designer to adopt a simulation model which uses the same state-update logic independent of the ticking rate of the simulator clock. This is why the precision of a simulator becomes low in practice when the ticking rate of the simulator clock is low or equivalently, the ticking interval becomes large.

On the other hand, if the states of all (say, 10,000) cars were to be updated every 50 ms for the sake of realizing high-precision RT simulation, then the simulation model of the cars and the freeway may lead to an excessive computational load imposed on the execution engine. Therefore, the ticking interval of the simulator clock cannot be made indefinitely small.

In distributed RT simulation, simulator objects are distributed among multiple nodes. This is a compelling mode of simulation when a large-scale simulation model is used.

Synchronization of the simulation-steps of distributed simulator objects is then a key challenge. In other words, a simulation-step executed by every member of the distributed simulator object group must be synchronized with the corresponding simulation-step executed by any other member. This also means that the simulator clock for one simulator object must commence the  $n$ -th tick neither before the  $(n - 1)$ -th tick by the clock for another simulator object nor after the  $(n + 1)$ -th tick by the clock for another simulator object.

Therefore, every member must perform some activities necessary to stay synchronized with other members. For example, distributed nodes may exchange completion reports at the end of each simulation-step as in the case of non-RT simulations (Fujimoto, 1990; Misra, 1986). However, this is not an efficient approach when RT simulation is needed and the number of nodes used is large. The essence of the DTS approach is the following:

1. Every node is equipped with an RT clock and executes each simulation-step upon reaching of the RT clock at the predetermined value.
2. Every simulation-step is designed to be completed within one ticking interval.

The DTS approach has major advantages over other distributed simulation approaches, even if we assume that the latter approaches can be adapted somehow to enable RT simulation (Fujimoto, 1990; Misra, 1986). This is because synchronization of simulation-steps executed by distributed simulator objects under the DTS scheme does not require message exchanges among the host nodes (not counting the message exchanges which may be needed at a certain low frequency for re-synchronizing the RT clocks of the nodes). The advantages become decisive in heavy-load distributed simulation situations.

However, even with the DTS approach, exchanges of messages that represent movements of certain simulation targets from the territory covered by one simulator component (e.g., TMO) to the territory covered by another simulator component are inevitable. Therefore, the ticking interval must be long enough to cover this kind of message exchanges.

In addition, the TMO-structured DTS approach enables easy design of simulator objects which use different ticking rates. For example, the ticking rate in one simulator object may be 1000 ticks per second whereas the ticking rate in another may be 500 ticks per second. This means that simulation of different parts of the target system with different degrees of fidelities can be easily realized.

### 3. An Overview of the TMO Scheme

The TMO scheme was established in the early 1990s (Kim et al., 1994; Kim 1997, 2000) with a concrete syntactic structure and execution semantics for economically reliable design and implementation of RT systems. The TMO programming scheme is a general-style component programming scheme and supports design of all types of components including distributable hard-RT objects and distributable non-RT objects within one general structure.

TMOs are devised to contain only high-level intuitive and yet precise expressions of timing requirements. No specification of timing requirements in (indirect) terms other than *start-windows* and *completion deadlines* for program units (e.g., object methods) and *time-windows for output actions* is required. For example, priorities are attributes often attached by the OS to low-level program abstractions such as threads and they are not natural expressions of timing requirements. Therefore, no such indirect and inaccurate styles of expressing timing requirements are associated with objects and methods.

At the same time the TMO scheme is aimed at enabling a great reduction of the designer's efforts in guaranteeing timely service capabilities of distributed computing application systems. It has been formulated from the beginning with the objective of enabling design-time guaranteeing of timely actions. The TMO incorporates several rules for execution of its components that make the analysis of the worst-case time behavior of TMOs to be systematic and relatively easy while not reducing the programming power in any way.

### 3.1. TMO Structure and Design Paradigms

TMO is a natural, syntactically minor, and semantically powerful extension of the conventional object(s). As depicted in Figure 1, the basic TMO structure consists of four parts:

ODS-sec = object-data-store section: *list of object-data-store segments* (ODSSs, each being a sharable group of data members);

EAC-sec = *environment access-capability* section: list of *gates* to remote object methods, logical communication channels, and I/O device interfaces;

SpM-sec = *spontaneous-method* section: list of spontaneous methods;

SvM-sec = *service-method* section.

Major features are summarized below. The second and third are the most conspicuous unique extensions of conventional object(s).

- 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.
- 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

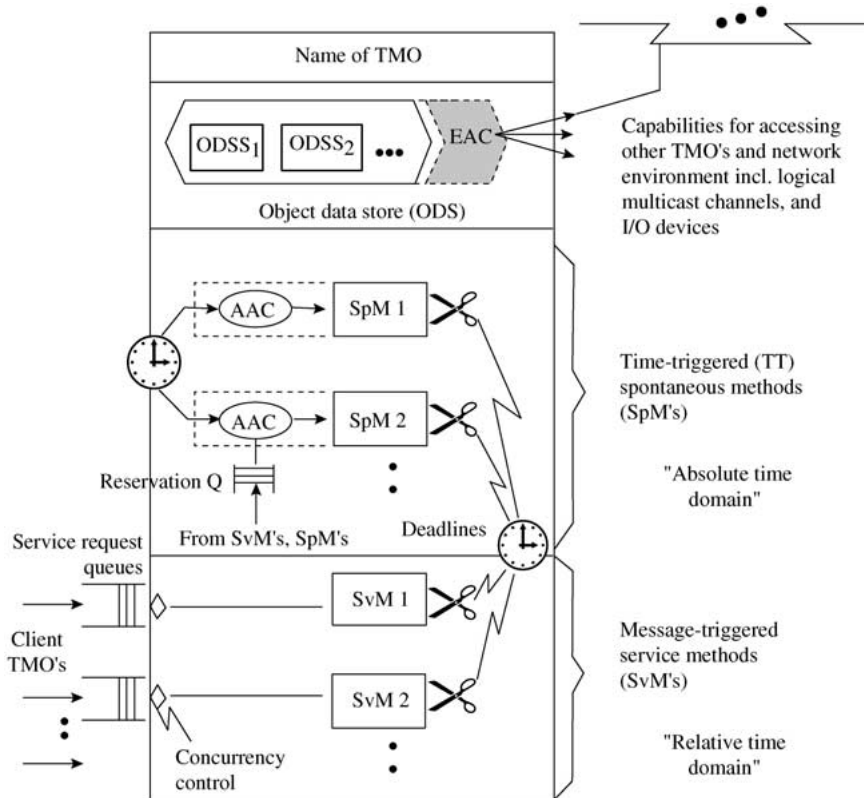


Figure 1. The basic structure of TMO (adapted from Kim (1997)).

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, *activation of an SvM triggered by a message from an external client is allowed only when potentially conflicting SpM executions are not in place*. An SvM is allowed to execute only when an execution time-window is big enough for the SvM that does not overlap with the execution time-window of any SpM which accesses the same ODSSs to be accessed by the SvM, opens up. 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*: The TMO incorporates deadlines 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 start 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 RT 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”}
```

An underlying design philosophy of the TMO scheme is that an RT 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 by each SpM executed on requests from the SvM and the *guaranteed completion time* (GCT) for the SvM in the specification of the SvM. Such specification of each SvM 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* (a composition of hardware, node OS, and middleware) available, the server object can be implemented to always execute the SvM such that the output action is performed within the time-window. The BCC contributes to major reduction of these burdens imposed on the designer. In determining the GCT of an SvM, one must consider several factors such as the worst-case (accepted) rate of invoking the SvM, worst-case execution times of various atomic segments of the SvM under worst-case scheduling scenarios, worst-case interactions with other remote SvMs which have their own GCTs, worst-case data arrivals from local peripherals and logical multicast channels, etc.

### **3.2. TMO Structuring in Environment Modeling and Multi-step Multi-level Design and Implementation**

The attractive basic design style facilitated by the TMO structuring is to produce a network of TMO's meeting the application requirements in a top-down multi-step fashion

(Kim, 1997). The engineering of an application system can start with a single TMO representation of the entire application environment (including the computer system to be designed) and proceeds through step-by-step expansion of the initial single TMO model toward a final implementation in the form of a network of TMOs executing on engines. This top-down process can also produce an RT simulator of the application environment, again in the form of a TMO network performing DTS.

#### 4. TMO-Structured DTS

The DTS approach facilitated by the TMO programming scheme uses distributed TMOs of which SpMs execute simulation-steps (Kim et al., 1996, 1999a; Kim 1997).

For example, a freeway-segment can be represented at a high level and simulated by the TMO in Figure 2.

The object data store (ODS) in this TMO contains state representations of the cars, the meters on entry-ramps, and the freeway structure.

Each TT method or SpM, when executed, updates a variable-set in the ODS representing the state of some simulation target item (i.e., physical item such as car, ramp meter, etc.) to reflect the current state of the target item. Ideally the TT methods should be activated continuously and each of their executions be completed instantly. However, the limited power of the execution engine dictates the activation frequency of any TT method to be a fraction of the ticking rate of the RT clock in the execution engine. The activation frequency of the TT method may be viewed as the ticking rate of the target item simulator clock. Each execution of a TT method must be completed within one ticking interval of the target item simulator clock. Therefore, TT methods are the mechanisms for approximately simulating continuous state changes that occur naturally in the target items in the environment.

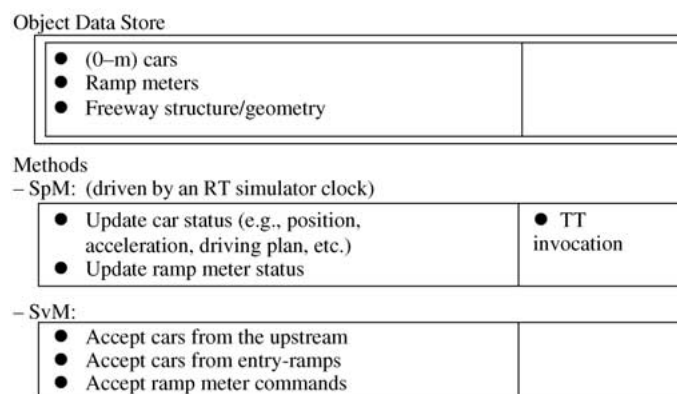


Figure 2. A TMO-structured simulation model for a freeway-segment.

The natural parallelism that exists among the simulation target items in the environment can be precisely represented by use of multiple TT methods which may be activated simultaneously. An alternative, which sometimes leads to a more efficient but maybe less easily understandable implementation, is to have a single TT method update the state variables representing multiple heterogeneous types of simulation target items. This alternative is the normal choice when the TMO execution engine contains only one CPU. In general, the precision of a TMO-structured simulation of the environment is a direct function of the activation frequencies of TT methods (which are equivalent to the ticking rates of the target item simulator clocks).

The simplicity and broad applicability of the structure depicted in Figures 2 and 3, are just some of many attractive features of the TMO-structured DTS scheme that offers significant potential for improving the economy and efficiency of distributed/parallel RT simulation over the state of the art. Other major features include: the possibility of systematic expansion of a TMO into a TMO network, and both the global time base support and the abstract programming support from the networked cooperating TMO execution engines in RT distributed computing systems.

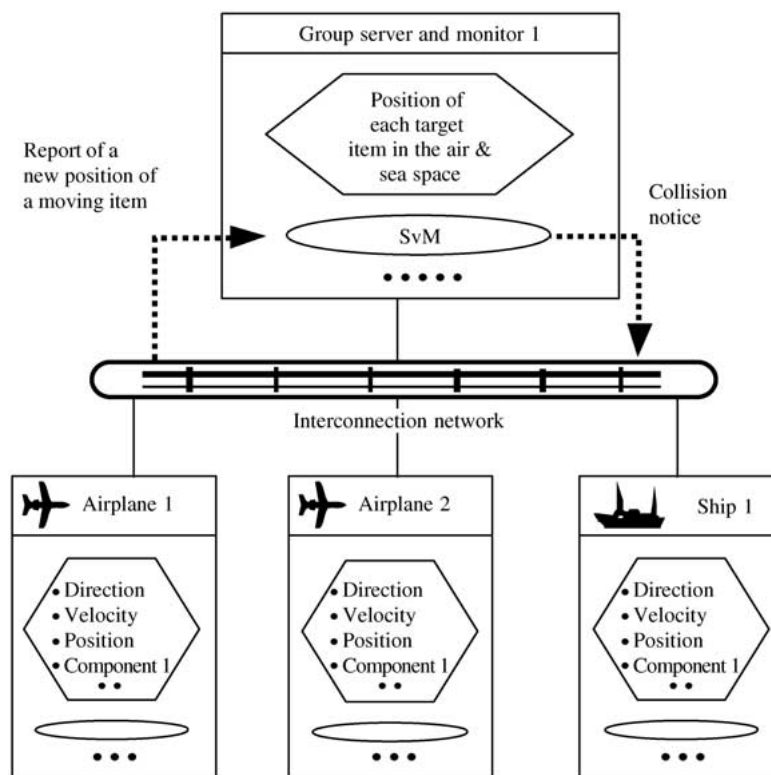


Figure 3. An expanded network of TMOs representing the simulation target in more details (adapted from Kim (1996)).

## 5. Major Issues in Efficient Implementation of DTS

In this section, some major issues in realizing efficient distributed RT simulators are discussed. Some techniques for resolving the issues within the DTS framework are also presented.

### 5.1. Group Server and Monitor (GSM) Object

Suppose that  $(0-m)$  cars in the ODS in Figure 2 are replaced by  $(0-m)$  airplanes, ramp meters by  $(0-n)$  ships, and the freeway structure/geometry by air-sea theater space. Also assume that the TT methods in Figure 2 are replaced by appropriate TT methods. The resulting single TMO representation, which may be called the Theater TMO, can be expanded into a more detailed representation structured in the form of a network of TMOs, each representing an airplane or ship, as shown in Figure 3.

Figure 3 also depicts an interesting role played by the TMO representing the air and sea space. This Space TMO maintains information on how the space is occupied. It facilitates detecting collisions between moving items such as airplanes, etc. As the TMO structured simulator of an airplane progresses after each simulator clock tick, the position of the airplane is updated and recorded within the TMO. In addition, this TMO notifies the Space TMO of the new position of the former. The latter TMO in turn checks if the airplane has now collided with any other environment item such as an airplane, ship, etc., and, if so, notifies the TMOs simulating the collided items. The notified TMOs then simulate the post-collision behavior of their simulation targets starting with the following tick of the simulator clock.

In a sense, the Space TMO supports the TMOs simulating the moving environment items. Therefore, it is called a *group server and monitor* (GSM) TMO. Each ticking interval of the simulator clock must cover the time spent in interaction between a GSM TMO(s) and monitored TMOs. Often the GSM TMO may contain service methods (SvMs) only, i.e., no TT methods. If the workload for a GSM object becomes too large, then multiple GSM objects, each supporting a different group of TMOs simulating the physical environment items, can be utilized.

On the other hand, the fact that there is the possibility of a collision among airplanes, means that the state descriptors for airplanes are tightly coupled in nature. In most cases it is not worth decomposing a TMO containing such tightly coupled data members into a group of TMOs supported by a GSM TMO. In the case of the simulation in Figure 3, such decomposition can be justified only if each update of the state descriptor for an airplane and that for a ship is highly time-consuming and thus parallel updating of multiple Airplane TMOs and Ship TMOs is really necessary. Without decomposition, parallel updating can be done if the computing node executing the monolithic Theater TMO has a sufficient number of processors and the TMO contains multiple TT methods designed to update state descriptors of different airplanes and ships. With decomposition, parallel updating could be done by either using a multi-processor based TMO execution engine or distributing Airplane TMOs and Ship TMOs to different nodes. However, the speed gain

from such parallel updating should be measured against the overhead incurred in interactions between the GSM TMO and Airplane TMOs and Ship TMOs.

It may also be possible to exploit parallelism inside each Airplane TMO and each Ship TMO. If the speed gain achievable through such parallelism exploitation is substantial, then it becomes an added incentive for decomposing the Theater TMO which involves creation of GSM TMO(s).

### 5.2. Update-Dependency among Simulator Objects

Figure 4 shows the simulated states of cars at 10:00:00 am and at 10:00:01 am, respectively. Let  $X(10:00:00)$  denote the simulated state of car X at 10:00:00 am. The terms “simulated car”, “car simulator”, “car simulator object”, and “car” are used interchangeably whenever there is no ambiguity. Assume that the entire simulator is a single TMO and thus runs on one node.

Therefore, at 10:00:01 am, the simulator must update the states of the cars as shown in Figure 4, using the information on the states of the cars at 10:00:00 am. Each car is characterized by the parameters such as (i) the *speed change probability* represented in the form of a function of the distance from the car to each adjacent car, the current speed, the preferred speed, and the road condition, (ii) the *lane change probability* represented as a function of the distance to the target exit and the distance to each adjacent car, etc. Random numbers are used in each update.

In updating car D to  $D(10:00:01)$ , not only  $D(10:00:00)$  but also at least  $B(10:00:00)$ ,  $C(10:00:00)$ ,  $E(10:00:00)$ , and  $F(10:00:00)$  must be used. In principle, cars can be updated in any order since in updating each car, only the old states (i.e., the states at 10:00:00) of adjacent cars and other parts of the freeway are used.

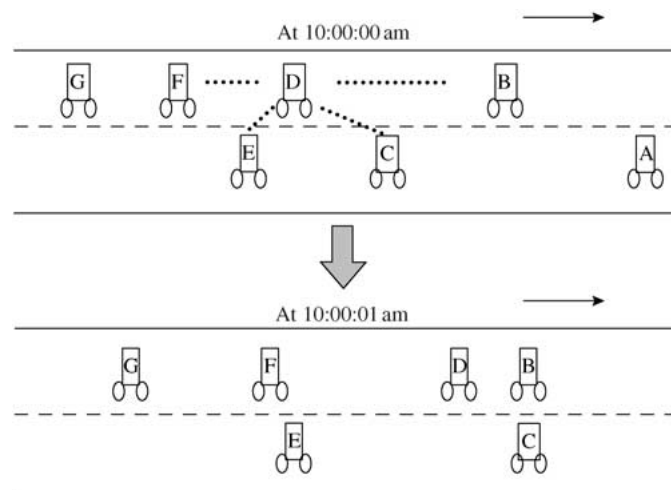


Figure 4. Single-node simulation of cars.

Actually it is not that simple.  $D(10:00:01)$  includes a new position of car D at 10:00:01. The new position can fall within a certain range depending upon the value of the random number used and other parameters such as possible speed range, etc. Suppose all cars have been independently updated to the states effective at 10:00:01. It is then possible for a car, say D, to be positioned ahead of its previous predecessor, say B, at 10:00:01. This can occur due to the effects of random numbers. However, this new state of the simulator is an inconsistent state (since D could not have flown over B).

This means that either  $D(10:00:01)$  or  $B(10:00:01)$  must be corrected. For one of the two cars, simply the current state value for 10:00:01 is discarded and a new state value for 10:00:01 is calculated. The new state value is then checked whether it leads the entire simulator to a consistent state.

Therefore, it is better to sort cars and update them in the sorted order. If the order is to update front cars first and rear cars later, then whenever a value for the new state of a car is calculated, a check is made whether it is in conflict with the already calculated new states of the cars in the front. If a conflict is detected (i.e., if the value is such that it leads the simulator to inconsistency), the value is discarded and an attempt is made to produce a new value until a conflict-free value is produced. When two simulator objects, e.g., car simulators B and D, are in such relationship and thus cannot be updated independently, they are said to be *update-dependent* upon each other. Note that the *update-dependency* is a *transitive relation*.

On the other hand, suppose the distance between car D and any of its adjacent cars at 10:00:00 is greater than the maximum distance that car D can travel in one ticking interval, i.e., 1 second. Then, there is no possibility for a newly calculated state value for D being in conflict with a newly calculated state value for any of D's adjacent cars. This means that D can be updated independently of its adjacent cars. Therefore, the update-dependency in the freeway simulation situation is a function of (i) the ticking rate (or the length of the ticking interval), (ii) the speeds of cars, and (iii) the distances between cars, among others. To put it another way, the update-dependency can be "broken" by changing any of those parameters. Of those parameters, the ticking rate is the only basic parameter commonly involved in all simulators.

### ***5.3. Minimization of the Impacts of Update-Dependency among Distributed Simulator Objects***

Suppose now a distributed simulation is attempted by partitioning the simulator (TMO) covering the freeway in Figure 4 into multiple simulator subsystems (TMOs), each covering a segment of the freeway, and running each simulator subsystem on a separate node. Figure 5 illustrates a two-TMO DTS system. As soon as the front edge of a car covered by TMO2 reaches the boundary between the two freeway-segments, the car should be taken over by TMO1.

An important aspect to note here is that the update-dependency among cars does not change by this partitioning of the simulator into multiple simulator subsystems. Suppose that in Figure 5, both TMOs run in parallel and each TMO updates front cars first and rear

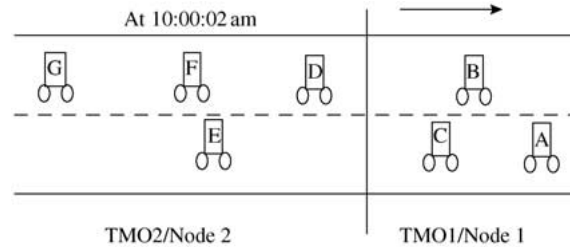


Figure 5. Distributed simulation of cars.

cars later. Cars are to be updated to the states effective at 10:00:03. If car D is update-dependent on car B and car C, then TMO2 needs new state values, B(10:00:03) and C(10:00:03), calculated by TMO1 in order to establish D(10:00:03).

If TMO2 must wait until TMO1 completes updating cars B and C before it can update car D, then almost no parallelism remains between TMO1 and TMO2. Therefore, such distributed simulation is worse than single-node simulation! An innovative approach is needed here to resolve this challenging situation.

One approach is to let TMO2 forward at the end of each simulation step all the cars which are in its territory and update-dependent on the cars in the territory of TMO1 to TMO1. To be more specific, at the end of each simulation step, TMO1 sends to TMO2 information on the cars which form the “end group”, i.e., the group containing the last car on each lane. TMO2 then identifies the cars which are in its territory and update-dependent on the end group in TMO1. Such identified cars are then transferred to TMO1 before the next simulation step begins. This means that the territorial boundaries between TMO1 and TMO2 will dynamically change. One major drawback of this approach is in the increased ticking interval which must cover the aforementioned exchanges between adjacent TMOs. Finding a more efficient approach is a challenging subject for future research.

Therefore, the key factor that determines the efficiency of distributed/parallel RT simulation is the update-dependency.

## 6. An Execution Engine for TMO-Structured DTS

### 6.1. TMO Support Middleware (TMOSM)

A cost-effective way to support execution of TMOs is to realize an execution engine by developing a middleware subsystem running on well-established commercial software/hardware platforms (Kim et al., 1999a; Shokri et al., 1998). An efficient middleware architecture, the TMOSM, has been developed. Then a prototype implementation on Windows NT, TMOSM/NT, has been obtained (Kim et al., 1999a). The most obvious and important requirement that a TMO execution engine must meet is to accurately honor the

timing specifications associated with various application program-segments, in particular, TMO methods. In this section, major characteristics of TMOSM that bring about such capabilities are discussed.

### 6.1.1. Internal Thread Structure

The internal thread structure of TMOSM is shown in Figure 6. 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 *watchdog timer and scheduler thread* (WTST). 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

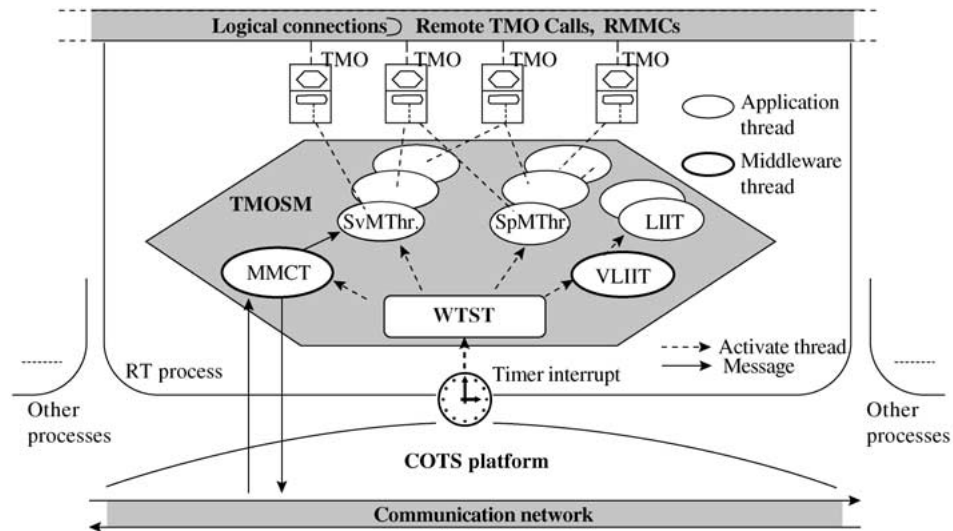


Figure 6. The basic internal thread structure of TMOSM.

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. **Middleware message communication thread (MMCT):** 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. **Virtual local I/O interface thread (VLIIT):** This virtual thread maintains a pool of threads which are called *local I/O interface 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).
3. **Virtual main system thread (VMST):** 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 done by WTST that executes the application scheduler function. Therefore, every time-slice conceptually belonging to the VMST is allocated to a fairly selected application thread.

Several features of this TMOSM architecture contribute to simplifying the analysis of the execution time behavior of application TMOs running on TMOSM. First, the strictly periodic nature of middleware threads and the 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  $\alpha$  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  $\alpha$ , the size of  $\alpha$ , 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.

Second, 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.

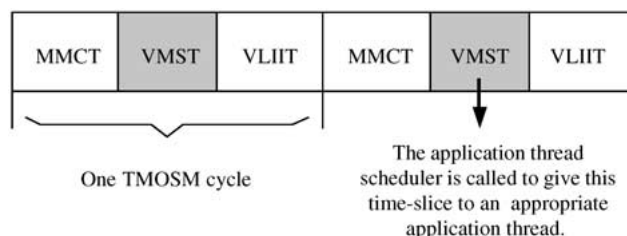


Figure 7. TMOSM scheduling cycle.

### 6.1.2. Two-Level Scheduling

Figure 7 shows the TMOSM scheduling cycle. TMOSM adopts a two-level scheduling approach: middleware thread scheduling and application thread scheduling. WTST first executes the middleware thread scheduler to select the next middleware thread—one among MMCT, VLIIT, and VMST. From the middleware thread scheduler's point of view, all application threads in TMOSM are treated as resource-sharing children of one virtual middleware thread, VMST. When the middleware thread scheduler selects VMST as the next thread to be executed, the application thread scheduler is called to select the next application thread to use that time-slice according to the adopted application scheduling policy. VLIIT behaves similarly.

Other architectural aspects of TMOSM which are useful in realizing the highly predictable behavior of the middleware are not discussed here due to space limit (Kim et al., 1999a). From the validation tests conducted, we have found that our prototype implementation of TMOSM on the Windows NT platform, TMOSM/NT, can accurately enact the time-window for activating a method as small as 10 ms and the method completion deadline as short as 10 ms unless a device driver not preemptible for an excessively long time gets involved.

## 6.2. Ticking Rate and the Simulator Execution Engine

The precision of the RT simulator is proportional to the ticking rate. In addition, for efficient distributed simulation, minimizing the update-dependency is necessary. This update-dependency can also be broken by increasing the ticking rate. Yet the ticking interval must be long enough to accommodate necessary message communications. Therefore, it is desirable to use computing platforms yielding small message delays.

If distributed computing nodes used in DTS are connected via bus LANs such as Ethernet, then it pays off to do some careful scheduling of bus access by the nodes (Kopetz, 1997). This is because without careful scheduling, the probability of a collision occurrence in bus access is very high, especially during the phase for exchanging state descriptors for some moving items in each simulation-step. Such scheduling is the job for

the simulation execution engine consisting of distributed computing nodes. For example, TMOSM currently enforces TDMA access by the nodes to an Ethernet bus.

Use of highly parallel computing platforms is an attractive approach. With such platforms, finding multiple channels that can simultaneously carry messages becomes easier. This means that it is easier to increase the ticking rate with such platforms. DTS using such platforms is considered a timely and potentially fruitful area for future research.

## 7. Conclusion

The DTS scheme is a fundamentally new type of an approach to distributed RT simulation. It requires a global time base which provides consistent RT values to application software running on distributed nodes. Although it has many advantages over other proposed distributed RT simulation approaches, the exploitation of its full potential requires advanced computing platforms such as highly parallel computing platforms. Cost-effective methods for realizing high-quality graphic display of the dynamically changing states of TMOs in DTS are also a meaningful subject for future research.

Therefore, the DTS scheme is a young research subject and much more research, especially, experimental research with advanced distributed and parallel computing platforms, is needed to learn better about its full potential and costs.

## Acknowledgment

The research on the DTS scheme was supported earlier by the US NSWC and the US DARPA under Contract N66001-97-C-8516. The current work is supported in part by the NSF under Grant Numbers 99-75053, 02-04050 (NGS) and 00-86147 (ITR), and in part by the US DARPA (NEST) under Contract F33615-01-C-1902 monitored by AFRL. No part of this paper represents the views and opinions of any of the sponsors mentioned above.

## References

- Ellenberger, R., Ling, R., Buscher, D., Uhde-Lacovara, J., and Shuler, R. 1993. Automatic generation of real-time ada simulations for space station freedom. *Simulation* November: 337–345.
- Fujimoto, R. M. 1990. Parallel discrete event simulation. *Communications of the ACM* 33(10): 30–53.
- Guyse, C., Buscher, D., and Ellenberger, R. 1994. Real-time environment and vehicle dynamics simulations for space station freedom integrated test and verification environment. *Simulation* April: 230–239.

- IEEE CS. 2000. *A Special Issue of Computer (a Magazine of IEEE Computer Society) on Object-Oriented Real-Time Distributed Computing*, June.
- ISORC (*IEEE CS International Symposium on Object-Oriented Real-Time Distributed Computing*) Series; 1st held in April 1998, Kyoto, Japan; 2nd in May 1999, St. Malo, France; 3rd in March 2000, Newport Beach, CA; 4th in May 2001, Magdeburg, Germany. Proceedings are available from IEEE CS Press.
- Kim, K. H. 1997. Object structures for real-time systems and simulators. *IEEE Computer* 30(8): 62–70.
- Kim, K. H. 2000. APIs enabling high-level Real-time distributed object programming. *IEEE Computer* 33(6): 72–80.
- Kim, K. H. et al. 1994. Distinguishing features and potential roles of the RTO.k object model. In *Proceedings of the WORDS'94 (IEEE CS'94 Work on Object-Oriented Real-Time Dependable Systems)* October 1994, Dana Point, pp. 36–45.
- Kim, K. H., Nguyen, C., and Park, C. 1996. Real-time simulation techniques based on the RTO.k object modeling. In *Proceedings COMPSAC'96 (IEEE CS'96 Software and Applications Conference)* Seoul, August 1996, pp. 176–183.
- Kim, K. H., Ishida, M., and Liu, J. Q. 1999a. An efficient middleware architecture supporting time-triggered message-triggered objects and an NT-based implementation. In *Proceedings of the ISORC'99 (2nd IEEE CS Int'l Symp. on Object-Oriented Real-time Distributed Computing)* St. Malo, France, May, 1999, pp. 54–63.
- Kim, K. H., Liu, J. Q., and Ishida, M. 1999b. Distributed object-oriented real-time simulation of ground transportation networks with the TMO structuring scheme. In *Proceedings of the COMPSAC'99 (IEEE CS'99 Computer Software and Applications Conference)* Phoenix, AZ, October 1999, pp. 130–138.
- Kopetz, H. 1997. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Boston: Kluwer Academic Publishers, ISBN: 0-7923-9894-7.
- Mathworks. 2002. Real-Time Workshop 4. Product description in <http://www.mathworks.com/products/rtw/>.
- Misra, J. 1986. Distributed discrete-event simulation. *ACM Computing Survey* 18(1): 39–65.
- OMG. 2000. Collection of slide presentations. *1st OMG Workshop on Real-Time/Embedded Distributed Object Computing*, July, Crystal City, VA.
- Otter, M., and Elmqvist, H. 2001. Modelica—Language, Libraries, Tools, Workshop and EU-Project RealSim. Technical note available from <http://www.modelica.org/documents/ModelicaOverview14.pdf>.
- Ozard, J., and Desira, H. 2000. Simulink model implementation on multi-processors under Windows-NT. In *Proceedings of the Military, Government and Aerospace Simulation Symposium. (A part of Advanced Simulation Technologies Conference (ASTC 2000))* (The SCSI Paper MGASS-197.pdf 518570), pp. 197–202.
- Papini, M., and Baracos, P. 2000. Real-time simulation, control and HIL with COTS computing clusters. In *Proceedings of the AIAA Modeling and Simulation Technologies Conference* Denver, CO, August 14–17, 2000 (AIAA Paper 2000-4593), pp. 1–6.
- Shokri, E., Crane, P., and Kim, K. H. 1998. An Implementation Model for Time-Triggered Message-Triggered Object Support Mechanisms in CORBA-Compliant COTS Platforms. In *Proceedings of the ISORC'98 (IEEE CS 1st International Symposium on Object-Oriented Real-Time Distributed Computing)*, Kyoto, Japan, April 1998, pp. 12–21.
- WORDS (*IEEE CS's Workshop on Object-Oriented Real-Time Dependable Systems*) Series; 1st held in October 1994, Dana Point; 2nd in February 1996, Laguna Beach; 3rd in February 1997, Newport Beach; 4th in January 1999, Santa Barbara; 5th in November 1999, Monterey; 6th in January 2001, Rome, Italy. Proceedings are available from IEEE CS Press.
- Zeigler, B., and Kim, J. 1993. Extending the DEVS-scheme knowledge-based simulation environment for real-time event-based control. *IEEE Transactions on Robotics and Automation* 9(3): 351–356.



**Kane Kim** is Professor of Computer Engineering and Computer Science at the University of California, Irvine, USA. He received an M.A. degree from the Computer Science Department at the University of Texas at Austin in 1972 and a Ph.D. degree from the Computer Science Division at the University of California, Berkeley, in 1974. He has been building up a research laboratory named DREAM (Distributed Real-Time Ever-Available Microcomputing) Lab during the past 17 years. He is the originator of the Distributed Recovery Block (DRB) technique and several other fundamental practical Dependable computing approaches and the principal originator of the TMO (time-triggered message-triggered object, also called RTO.k) programming scheme. Dr. Kim is a recipient of the 1998 IEEE Computer Society's Technical Achievement Award for his contributions to the scientific foundation of both real-time fault-tolerant computing and real-time object-oriented distributed computing.

Dr. Kim is a fellow of IEEE (elected in Fall 1988) and a member of the IFIP WG 10.4 on Dependable Computing. He served as the Chairman of the IEEE Computer Society's Technical Committee on Distributed Processing during 1984–1986 and hosted several IEEE conferences. He was the main initiator of the IEEE Computer Society's WORDS (Workshop on Object-oriented Real-time Dependable Systems) series and the ISORC (Int'l Symp. on Object-oriented Real-time distributed Computing) series. He also served as a member of the founding editorial board for the IEEE Transaction on Parallel and Distributed Systems during 1989–1994.