

Object-Oriented Real-Time Distributed Programming and Support Middleware

(Keynote Paper)

K. H. (Kane) Kim
University of California
Irvine, CA 92697, U.S.A.
kane@ece.uci.edu

Abstract: The object-oriented (OO) distributed real-time (RT) programming movement started in 1990's and is growing rapidly at this turn of the century. The motivations are reviewed and then a brief overview is given of the particular programming scheme which this author and his collaborators have been establishing. The scheme is called the *time-triggered message triggered object* (TMO) programming scheme and it is used to make specific illustrations of the issues and potentials of OO RT programming. The desirable features of middleware providing execution support for OO RT distributed programs are then discussed. The issue of fault-tolerant execution of distributed RT objects and that of RT distributed / parallel simulation are also discussed.

Keywords: real time, deadline, guarantee, TMO, time-triggered, message triggered, object, middleware, fault, tolerance, distributed, programming, simulation.

1. Introduction

Among several cutting-edge technology movements initiated in 1990's in software engineering is the *high-precision real-time (RT) object-oriented (OO) programming* movement [Alc99, Att91, IEE00, Ish92, J-C00, Kim94b, Kim97b, Kop90, OMG99a, RJE00, Tak92]. In this author's view, the most important goal of that movement has been to instigate a quantum productivity jump in software engineering for RT computing application systems. Particularly targeted application domains have been those challenging large-scale distributed / parallel computing applications in fields such as factory automation, telecommunication, defense, intelligent transportation, emergency management, etc.

Now that a new century has just begun, it seems to be fitting time to reflect a bit on the state of the art and future directions. The movement is still in its youthful stage and its impact has just started surfacing up. However, its great potential is now much more clearly and widely recognized than it was in mid-1990's.

In the next section, the motivations for pursuing the OO RT programming approach are reviewed and then in Section 3, a brief overview is taken of the particular programming scheme which this author and his collaborators have been establishing. This programming scheme called the *time-triggered message triggered object* (TMO)

programming scheme is used on several occasions in the rest of this paper to make specific illustrations of the issues and potentials of OO RT programming.

The desirable features of middleware providing execution support for OO RT distributed programs are discussed in Section 4. The application programming interfaces (APIs) which wrap the services of middleware and enable convenient OO RT programming are also discussed briefly. Section 5 deals with the immature but important area of middleware supporting fault-tolerant (FT-) OO RT distributed programs.

Another important branch of OO RT distributed computing is distributed RT simulation. Here *RT simulation* refers to the mode of simulation in which the simulation objects show the timing behavior that are the same as or similar to the timing behavior of the simulation targets. This branch is discussed in Section 6. A conclusion of this paper is drawn in Section 7.

2. Motivations for using the OO RT programming approach

2.1 Complexity and costs of RT distributed programming

Starting a few years ago, the field of RT computing applications has been showing a rapid growth pattern. Computer systems in those application domains are generally responsible for *RT control* of physical devices, *RT storage and search* for information, and *RT communication and display* of information. In addition, they are often tasked to perform *RT simulation* of their application environments. The field of computer-embedded communication-equipped system engineering has been growing particularly fast in recent years.

As a result, industry has felt an acute need for RT distributed programming and software engineering methods which are at least *multiple times more effective* than currently widely practiced programming techniques. This new-generation RT distributed software engineering method must be based on a "general high-level programming style" which can be accommodated with minimal efforts by current-generation business application programmers (using C++ and Java) rather than on a style that has been practiced by assembly language or low-level language programmers.

Continuous use of old low-level programming styles is not economically viable for dealing with increasing demands for new RT application systems.

Designers must be required to specify both the interactions among distributed program components and the timing requirements of various actions in natural intuitively appealing forms only. Designers should not be forced to deal directly with notions such as priorities for the sake of meeting application timing requirements since priorities are usually associated with low-level computation units such as processes and threads in manners reflecting the application semantics poorly [Kim99d].

The fact that *distributed objects* represent a higher-level structure for distributed programs than *distributed processes* do have been widely recognized by the industry in the past 10 years, e.g., technology movements such as CORBA [OMG99b, Sol95], DCOM [Ses97], and RMI [Sun98]. Naturally, researchers started searching for extensions of distributed objects that allow unambiguous specification of timing requirements imposed on various computations units [ISO98, ISO99, ISO00, WOR94, WOR96, WOR97, WOR99, WOR99F].

2.2 Should RT programming remain an esoteric branch of computer science and engineering ?

It is fair to say that up to now, RT programming has been treated as an esoteric branch of computer science and engineering. Very few universities have courses on RT programming and even those few existing courses are almost entirely graduate courses.

The main reason is that RT programming has been practiced largely as an ad hoc art in a form looking quite alien to the vast number of business and scientific application programmer. On the other hand, there is no reason why future RT computing cannot be realized in the form of a generalization of the non-RT computing, rather than the other way around. Figure 1 depicts this. If the main-stream (traditional) programming science is viewed as a study of the two-dimensional space, (data X operation), then a proper form of RT programming should be practiced as work within the three-dimensional space, (data X operation X time). Of course, the less the programmer is burdened with the work on the time dimension, the better off. We just need a powerful programming scheme capable of dealing with all practically useful RT and non-RT computing requirements in uniform manners. Under such a properly established RT programming methodology, every practically useful non-RT program must be realizable by simply filling the time constraint specification part with the default value "unconstrained".

2.3 Reliability of RT distributed programs

RT programs have been notoriously difficult to analyze. It is well known that testing alone does not ensure sufficiently high reliability of RT programs. Given rapidly increasing demands for RT application

systems and the fact that complexities of RT distributed programs are far greater than those of single-node programs, the practice of relying solely on testing for

reliability assurance is becoming less and less tolerable.

A new-generation RT distributed software engineering method must allow some system engineers dealing with safety-critical applications to confidently produce certifiable RT distributed computing systems. The general public which has witnessed conspicuous improvements in the reliability of the desk-top computer systems in 1990's will demand in this new century a different level of reliability for the systems in safety-critical applications. They will demand sufficiently trustable certifications of the designs and implementations. *Design-time guaranteeing of response times of computing components / systems* is considered a major technological requirement that must be fulfilled before such certification becomes a common practice.

Moreover, distributed computer systems (DCSs) used in safety-critical RT applications must possess some degrees of *RT fault tolerance* capabilities. The current reality is that widely used node operating systems (OSs) do not show easily analyzable and predictable timing behavior. Whether we like it or not, there are certain *hard deadlines* in human societies and violation of these deadlines have severe consequences. For example, suppose cars are to be driven by automated drivers (robots). If such cars are heading toward a collision course, then the collision can be avoided only if at least one driver detects the danger and takes an avoidance action within a certain hard deadline. Applications subject to hard deadlines are called *hard-real-time (HRT) applications*.

Therefore, both unreliable hardware components and node OSs with erratic timing behavior can lead to violation of hard deadlines. This makes employment of RT fault tolerance mechanisms in safety-critical RT DCSs to be imperative.

Again, whether hardware and node OSs possess RT fault tolerance mechanisms or not, distributed HRT applications must be designed with *response time guarantees*. It cannot be done if application software is structured in undisciplined manners. That is, easily analyzable structuring of application software must be pursued to the maximum extent possible. Research in recent years has made it clear that high-level structuring in the form of distributed RT objects has significant

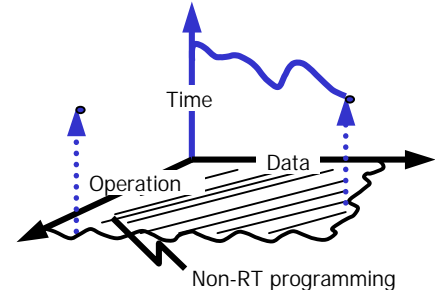


Figure 1. RT programming as a generalization of non-RT programming

advantages in this regard in comparison to somewhat lower-level distributed process structuring.

2.4 HRT program components

From the viewpoint of realizing systematic modular construction of sizable HRT distributed computing applications, one highly desirable approach is to use *HRT program components*, each of which is associated with a *guaranteed service time (GST)*, also called *guaranteed completion time*, for every service method that it provides [Kim00b]. If a program component provides multiple service methods, it is associated with multiple GSTs. If every program component contributing to the computation subject to a hard deadline has a GST associated with it, then meeting the hard deadline becomes the trivial problem of checking whether the sum of the GSTs of the contributing components is indeed less than the hard deadline. The real problem then is to ensure that every GST associated with every program component is credible. In a sense, each GST is a hard deadline that the designer of the program component has decided to impose on the program component. Therefore, implementation of such a program component may involve the use of fault tolerance mechanisms.

If a program component fails to meet a GST, then a number of other program components designed to be dependent on the former component may also be treated as failed components unless the latter were also designed to handle reports about the failure of the former component. Any attempt to replace a GST of a program component by a "soft" deadline to be imposed on the component is expected to lead to a complicated methodology which does not enable modular systematic construction of HRT systems. Special situations where it might be worth augmenting HRT program components with statistical performance indicators were discussed in [Kim00b].

Therefore, the HRT program component possessing GST attributes is a key to cost-effective systematic construction of HRT distributed computing applications.

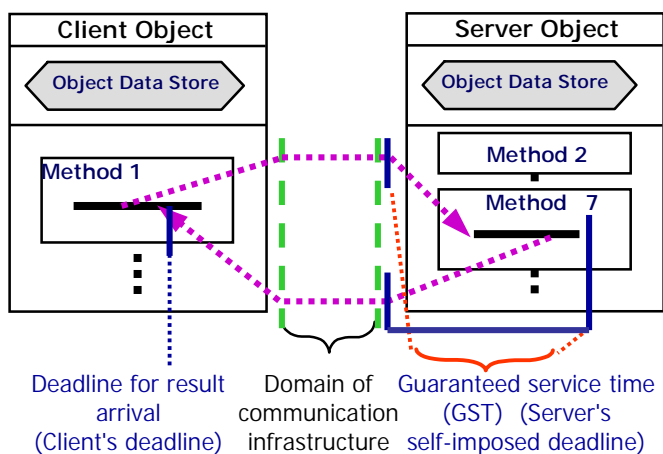


Figure 2. Client's deadline vs. Server's GST (adapted from [Kim00b])

This *HRT component based construction approach* is conservative in nature but highly cost-effective due to its systematic and modular characteristics. Here the designer/implementer of an HRT component announces its GST to all potential designers of client components. We expect that this HRT component based construction approach will meet increasing acceptance in the practicing field in the future.

Figure 2 depicts the relationship between a client and a server component in a system composed of HRT components which are structured as distributed computing objects. The client object in the middle of executing its method, Method 1, calls for a service, Method 7 service, from the server object. In order to complete its execution of Method 1 within a certain target amount of time, the client must obtain the service result from the server within a certain deadline. During the design of this client object, the designer searches for a server object with a GST acceptable to him/her. The designer must also consider the time to be consumed by the communication infrastructure in judging the acceptability of the GST of a candidate server object.

3. An overview of a high-level approach to programming real-time distributed systems: the TMO scheme

As a concrete example of a high-level OO RT distributed programming approach that has been based on the philosophy discussed in the preceding section, the *time-triggered message-triggered object (TMO)* programming scheme developed by the author and his collaborators is briefly summarized in this section. The TMO scheme was established in early 1990's [Kim94b, Kim97b, Kim99b, Kim00d] with a concrete syntactic structure and execution semantics for economical reliable design and implementation of RT systems. The TMO scheme is a general-style component structuring scheme and supports design of all types of components including distributable HRT objects and distributable non-RT objects within one general structure.

Calling the TMO scheme a high-level distributed programming scheme is justified by the following characteristics of the scheme:

- (1) No manipulation of processes and threads: Concurrency is specified in an abstract form at the level of object methods. Since processes and threads are transparent to TMO programmers, the priorities assigned to them, if any, are not visible, either.
- (2) No manipulation of hardware-dependent features in programming interactions among objects: TMO programmers are not burdened with any direct use of low-level network protocols and any direct manipulation of physical channels and physical node addresses / names.
- (3) 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*: TMOs are devised to contain only high-level intuitive and yet precise expressions of timing requirements. 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 [Kim99d]. Therefore, no such indirect and inaccurate styles of expressing timing requirements are associated with objects and methods [Kim97b].

At the same time the TMO scheme is aimed for 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 [Kim97a, Kim99b].

3.1 TMO structure and design paradigms

TMO is a natural and syntactically minor but semantically powerful extension of the conventional object(s) [Kim97b, Kim99e, Kim00b]. As depicted in Figure 3, the basic TMO structure consists of four parts:

- ODS-sec** = object-data-store section: list of *object-data-store segments* (ODSS's);
- EAC-sec** = *environment access-capability* section: list of *gate* objects (to be discussed later) providing efficient call-paths 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.

(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-*

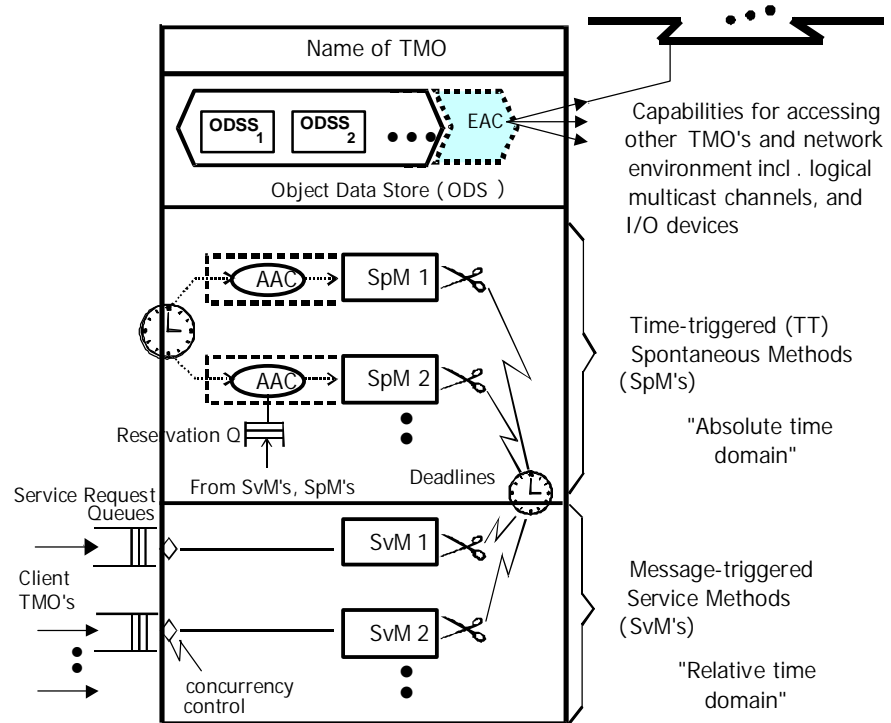


Figure 3. The basic structure of TMO (Adapted from [Kim97b])

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 RT clock at specific values determined at the design time whereas the SvM executions are triggered by service request messages from clients. Moreover, actions to be taken at real times *which can be determined at the design time* can appear only in SpMs.

(c) Basic concurrency constraint (BCC):

This rule prevents potential conflicts between SpMs and SvMs and reduces the designer's efforts in guaranteeing timely service capabilities of TMOs. Basically, *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 big enough for the SvM that does not overlap with the execution time-window of any SpM that 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 and deadline:

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 times and completion times.

Triggering times for SpMs must be fully specified as constants during the design time. Those real-time constants appear in the first clause of an SpM specification called the *autonomous activation condition* (AAC) section. An example of an AAC is

```
"for t = from 10am to 10:50am every 30min
  start-during (t, t+5min) finish-by t+10min"
```

which has the same effect as

```
{ "start-during (10am, 10:05am)
  finish-by 10:10am",
  "start-during (10:30am, 10:35am)
  finish-by 10:40am" }
```

A provision is also made for making the AAC section of an SpM contain only *candidate* triggering times, not actual triggering times, so that a subset of the candidate triggering times indicated in the AAC section may be dynamically chosen for actual triggering. Such a dynamic selection occurs when an SvM within the same TMO object requests future executions of a specific SpM. Each AAC specifying candidate triggering times rather than actual triggering times has a name.

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.

Middleware which together with node OSs and hardware make up TMO execution engines, have been developed [Kim96a, Kim99c, Kim99f]. These and other middleware supporting OO RT programs will be discussed in Section 4.

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 [Kim97b]. 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 TMO's executing on engines. This top-down process can also produce a RT simulator of the application environment, again in the form of a TMO network. In fact, the TMO scheme facilitates an attractively simple approach to parallel and distributed real-time simulation, called the *distributed time-triggered simulation (DTS)* [Kim96b, Kim97b, Kim99e]. This systematic approach has been shown to be practical via several experiments which dealt with RT distributed applications such as a missile defense command-control application [Kim97b, Sho98a], a freeway car traffic control application [Kim99e], and a steel mill factory control application [Kim00c].

4. Middleware supporting OO RT programs

A cost-effective way to support execution of OO RT distributed programs is to realize an execution engine by developing middleware running on well established commercial software / hardware platforms. RT CORBA object request brokers (ORBs) under development are examples of such middleware [Ete99, Sch98].

To support TMOs, an efficient middleware architecture, named *TMO Support Middleware* (TMOSM), has been developed. Then a prototype implementation on Windows NT, TMOSM/NT, has been obtained [Kim99c]. Our experiences indicate that even this middleware extension of a general-purpose OS (Windows NT) can support application actions with the 10ms-level timing accuracy. Also, another prototype implementation in the form of a CORBA service, TMOSM/AnyORB/NT, that runs on platforms equipped with Windows NT and a basic ORB and supports CORBA-compliant application TMOs has been obtained [Kim99f].

TMOSM implementations on the Windows NT platforms have been operational in the author's laboratory along with several non-trivial applications structured as TMO Networks. A friendly application programming interface (API) wrapping the services of TMOSM has also been developed and named the *TMO Support Library* (TMOSL) [Kim99c, Kim00d]. It consists of a number of C++ classes. TMOSL empowers C++ programmers with powerful and natural mechanisms for specification of unique and essential features of RT distributed programs.

4.1 TMOSM

In this section, a brief overview of the structure of TMOSM is provided as an illustration of functionalities of middleware supporting OO RT distributed programs.

The internal thread structure of TMOSM is shown in Figure 4. TMOSM consists 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. This author believes that

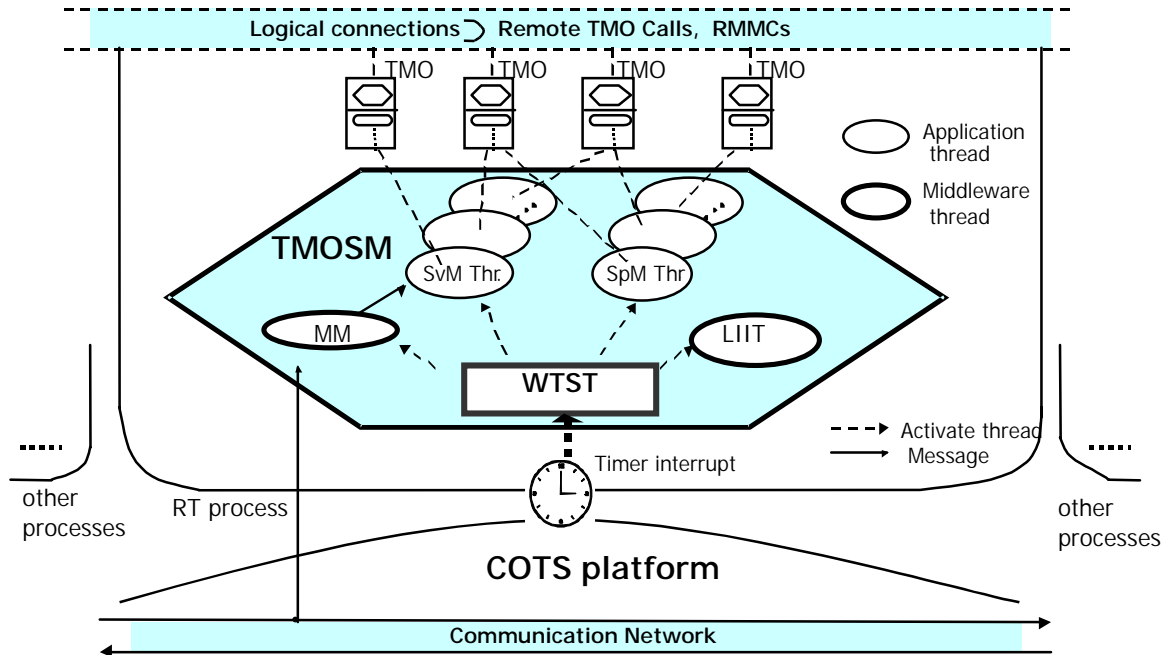


Figure 4. The basic internal thread structure of TMOSM (adapted from [Kim99c])

structuring of middleware threads as periodic threads is a fundamentally sound approach which leads to easier analysis of the worst-case time behavior of the object execution engine without incurring any significant performance drawback. The super-micro thread is called the WTST (*Watchdog Timer & Scheduler Thread*). It is a "super-thread" in that it runs at the highest possible priority level. It is also a "micro-thread" in that it manages the scheduling / activation of all other threads in TMOSM. Even those threads created by the node OS before TMOSM starts are executed only if WTST allocates some time-slices to them. Therefore, WTST is activated whenever thread switching needs to be performed, e.g., upon expiration of a time-slice. Also, WTST checks for any deadline violations and if a violation is found, it provides an exception signal to the user.

The set of middleware threads is *fixed at the TMOSM start time*. The following three middleware threads handle the core functions:

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

(2) LIIT (Local I/O Interface Thread): This periodic thread executes functions that utilize 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.

(3) VMST (Virtual Main System Thread): Every time-slice not used by the above two middleware threads is conceptually given to this virtual thread which merely represents all application threads running TMO methods. The actual time-slice allocations are taken by WTST which executes the application scheduler function. In summary, every time-slice conceptually belonging to the VMST is allocated to a fairly selected application thread.

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 [Kim99c]. From the validation tests conducted, we have found that TMOSM/NT can accurately enact the time-window for activating a method as small as 10ms and the method completion deadline as short as 20ms unless a device driver not preemptible for an excessively long time gets involved.

The service request communications in TMOSM/NT are based on UDP/IP and TCP/IP, while those in TMOSM /AnyORB/NT take place through the ORB infrastructure.

4.2 TMOSL: An API-style approximation of an abstract TMO programming language

Creating a new language and its support tools which do not share much with C++ and/or Java tools is a highly expensive endeavor which are unlikely to have near-term impacts on the RT application system community. Ideally, TMO programming can be supported by an extension of C++ or Java that requires a new extended compiler. Even such a language extension is an expensive endeavor of which potential rewards are not

fully clear. Therefore, TMOSL (TMO Support Library) has been created as an API that enables TMO programming to the extent of convenience that goes quite close to that which can offered by a TMO language [Kim00d]. Some major features of TMOSL are briefly summarized below.

(1) Remote SvM call:

When a TMO is instantiated, the symbolic name for the object, say "TMO1", and the symbolic name for each SvM in that object, say "SvM7", should be registered with the execution engine. Then a *gate* object corresponding to SvM7 should be created in every potential host node of a client TMO. The gate is an entry point to the efficient call-path leading to the associated SvM. When a gate is instantiated, the symbolic name of the associated SvM is used as a parameter for the constructor but thereafter, client TMOs can just use the name of the gate in calling the SvM without using the symbolic name. To enable easy creation of a gate, TMOSL contains a class named *GateClass*. Then, the gate for the SvM named "SvM7" can be created by

```
"GateClass G1 ( "TMO1", "SvM7" )".
```

During the construction of G1, TMOSM establishes a call-path from G1 to TMO1.SvM7 on another node.

The definition of the parameter structure for SvM7 may also be listed as a companion of the above gate creation statement.

```
"struct ParamStruct_TMO1_SvM7 { int a; float b; }".
```

A client TMO may then contain an instantiation such as "ParamStruct param1". Since no new language translator is used in programming with the basic TMOSL/NT, it is necessary to group all parameters into a single structured variable and let the client pass a pointer for the variable along with the information on the size of the memory area of the variable onto the execution engine. The client engine then transfers the structured parameter to the engine executing TMO1.SvM7 as a single message across the network.

Both blocking and non-blocking types of remote SvM call operations are included as methods inside *GateClass*.

(2) Object data store segment (ODSS) as a data storage unit accessed by atomic computation-segments

For controlling concurrency in executions of TMO methods, TMOSL requires the programmer to explicitly indicate for each TMO method the set of object data variable groups that needs to be accessed by the TMO method.. The *ODS segment* (ODSS) is a basic unit of data storage (i.e., a group of variables) which can be reserved for exclusive access by a TMO method. Given such specifications, even the object execution engine can easily check whether two TMO method executions may interfere with each other or not. Therefore, TMOSL provides a class, *ODSSBaseClass*, containing basic operations applicable to an ODSS including registration of the ODSS with the execution engine.

(3) GCT of an SvM

The GCT of an SvM is inherently subject to the condition that the aggregate arrival rate of calls from all possible clients to the SvM does not exceed the threshold called the *maximum invocation rate* (MIR). Therefore, when an SvM is created and initialized, not only the symbolic name of the SvM and the set of ODSSs to be accessed but also the GCT and the MIR must also be registered with the engine. TMOSL supports this by providing the class *SvMbaseClass*.

(4) SpM

TMOSL contains the class *AACclass* of which the constructor registers with the execution engine the parameters such as AAC activation-time, deactivation-time, period between TT executions, earliest start-time of each TT execution, latest start-time of each execution, and GCT for an execution. One more parameter used here indicates whether the constructed object is a *permanent AAC* or a *candidate AAC* with a name. When an SpM is created, there is no symbolic name to be registered with the execution engine but as in the case of an SvM, the set of ODSSs to be accessed by the SpM and the set of associated AACs must be registered.

To support TT executions of SpM-segments (not just entire SpMs) in a limited form, TMOSL provides "wait_for (microsec x1)" and "wait_until (r_tm t1)" when r_tm denotes the type "real-time".

(5) TMO interaction via multicast channels

In addition to the interaction mode based on remote method invocations, TMOs can use another interaction mode where messages may be exchanged over logical message channels explicitly specified as data members of involved objects. Such channels are called *RT multicast and memory-replication channels* (RMMCs) (previously called the programmable data field channels) [Kim99g].

4.3 Other middleware supporting OO RT distributed programs

OMG has been developing the RT Corba specifications in the past few years. They decided to proceed in two steps. First, they produced the version based on static priority assignment [Alc99]. An ORB meeting this specification has also been produced [Sch98]. This step has been justified on the grounds that many current-generation RT programmers who deal with safety-critical applications and use low-level programming languages may have easier time adapting to such styles of programming distributed objects. However, this author feels that static priority assignment does not match well with distributed RT object programming [Kim99d]. It defeats the goal of adopting the high-level OO style into distributed RT programming. Recognizing the limitation of the first version, OMG started the second step of developing a specification based on dynamic scheduling two years ago [OMG99a, Ete99].

It was mentioned earlier that since no new language translators were to be used in programming with the basic

TMOSL/NT, it was necessary to group all parameters into a single structured variable and let the client pass a pointer for the variable along with the information on the size of the memory area of the variable onto the execution engine. This restriction is removed in the CORBA which uses an IDL (*interface definition language*) translator [OMG99b, Sol95]. The programmer of a CORBA object class produces an IDL specification, which contains the method names and method parameters, in addition to the class. An IDL translator then takes the IDL specification as an input and produces two program-modules, one called the *stub* for use by the client objects and the other called the *skeleton* for use by the server object. The stub-skeleton pair takes care of parameter transfer across the network and may perform multiple message exchanges to handle a large set of parameters.

A CORBA service named *TMO execution support* (TMOES) that supports CORBA-compliant application TMOs has been defined and a prototype implementation, TMOSM/AnyORB/NT, that runs on platforms equipped with Windows NT and a basic ORB has been obtained [Kim99f]. Of course, the API wrapping this TMOES involves the use of IDL and no restriction on parameter structuring is imposed.

Middleware supporting distributed RT Java objects are expected to become available in the near future [J-C00, Jen00, RJE00]. So are the middleware supporting distributed RT objects based on the DCOM architecture.

5. Middleware supporting fault-tolerant OO RT distributed computing

Fault tolerance capabilities are required in many distributed RT computing applications. The architecture and a prototype implementation of a middleware which supports reliable fault-tolerant execution of TMO-structured distributed applications, have been developed by this author and his collaborators. This middleware has been named ROAFTS (*Real-time Object-oriented Adaptive Fault Tolerance Support*) [Kim98b, Sho98b]. The ROAFTS middleware is reviewed in this section and other middleware developments efforts under way are also briefly mentioned.

5.1 Essence of ROAFTS

The ROAFTS middleware is capable of

- (i) *network surveillance* for cooperative distributed detection of resource and computational failures [Kim94a, Kim99a]; and
- (ii) *adaptive configuration management* including allocation of available resources to the active applications such that the minimum required quality of services (QoS's) of critical components may be maintained for longest possible periods.

ROAFTS uses TMOSM as an integral component and supports fault-tolerant execution of TMOs and conventional passive objects *with tightly bounded time*

costs. ROAFTS is aimed for minimizing the efforts of the budget-constrained system engineers to ensure that the risk of the system failing to accomplish its critical missions is minimized.

Two prototype implementations have been produced along with corresponding versions of non-trivial RT network application software [Kim98c, Sho98b]. These prototypes are implementations of the core part of ROAFTS and one is based on the in-house kernel (called the DREAM kernel [Kim96a, Kim98a]) while the other is on the Sun Microsystems Solaris platform. A prototype implementation of the core part of ROAFTS on Windows NT platforms is being tested at the time of this writing.

The reconfiguration and adaptation manager in ROAFTS is designed to accept application-specific policy parameters and reflect those parameters in the RT adaptation decisions.

5.2 RT fault tolerance techniques incorporated into ROAFTS

In devising ROAFTS, we considered only those fault tolerance schemes for which *recovery time bounds* can be easily established. It is our judgment that as the basic OS and communication subsystem technologies become mature, rapidly growing demands for quantitatively guaranteed RT fault tolerance will be generated from the customers of distributed applications, not just from the customers of single-node computing system applications.

A natural and modular way of constructing fault-tolerant distributed and/or parallel computer systems is to

- (1) construct various subsystems in the rugged fault-tolerant forms,
- (2) interconnect them by use of rugged fault-tolerant communication subsystems, and
- (3) operating them under the supervision of a *network surveillance and reconfiguration* (NSR) manager which detects the needs for system reconfiguration, performs or coordinates reconfiguration, and invokes detailed system diagnosis tasks periodically or as needs arise.

The most basic type of subsystems are *computing stations* each of which consists of a processing node (hardware and software) dedicated to the execution of one or a few application processes or objects [Kim94a]. Every complex subsystem can be constructed with such computing stations.

Over the years this author and his collaborators have formulated several basic RT fault tolerance techniques yielding small recovery time bounds. These include

- (1) the *pair of self-checking processing nodes* (PSP) scheme which is in essence to use two copies of a self-checking computing component in the form of a primary-shadow pair [Kim94a, Kim98b],
- (2) the *distributed recovery block* (DRB) scheme [Kim94a, Kim95] which is an extension of the PSP scheme for tolerance of both hardware and software faults in process-structured RT distributed systems, and

(3) optimal versions of the *periodic reception history broadcast* (PRHB) scheme for RT network surveillance in bus-LAN based systems [Kim93, Kop89], etc.

The PSP / DRB scheme above is a powerful approach to constructing computing stations in fault-tolerant forms. More recently formulated techniques are:

- (4) the *primary-shadow TMO replication* (PSTR) scheme [Kim97a, Kim00a] which is an extension of the DRB scheme for fault tolerance in RT object-based schemes,
- (5) the *supervisor-based network surveillance* (SNS) scheme [Kim99a] for RT network surveillance in point-to-point network based systems,
- (6) the *release-time based fault-tolerant real-time multicast* (RFRM) scheme [Kim99g].

During prototype implementations of ROAFTS, basic techniques for fault-tolerant computing station structuring such as the PSP / DRB scheme, the PSTR scheme, the PRHB scheme, and the SNS scheme were incorporated first. Then the ROAFTS architecture was extended to incorporate integrations of the aforementioned basic techniques and an adaptation manager, which are the following two adaptive fault tolerance schemes.

- (1) the *adaptable DRB* (ADRB) scheme [Kim98b, Sho98c] combining
 - the self-checking node / sequential recovery block scheme [Ran95],
 - the PSP / DRB scheme,
 - the SNS scheme,
 - the PRHB scheme, and
 - the adaptation manager;
- (2) the *adaptable PSTR* scheme combining
 - the TMO rollback scheme,
 - the PSTR scheme,
 - the SNS scheme,
 - the PRHB scheme, and
 - the adaptation manager.

ROAFTS can easily incorporate additional fault tolerance techniques, e.g., TMR, and multicast techniques [Tac98].

5.3 Other middleware

Developments of middleware supporting fault tolerant execution of some types of distributed RT objects are under way in a number of research organizations besides the author's laboratory [Nar99, Ren99]. It is a timely research subject. The main differences between the ROAFTS approach and other approaches are in the tight recovery time bounds emphasized in ROAFTS. There are also some differences in the range of the types of faults handled while other approaches optimize different aspects, e.g., less overhead incurred for less fault coverage, and thus they are complimentary to ROAFTS.

6. Distributed time-triggered simulation

As mentioned in Section 3.2, the TMO scheme facilitates an attractively simple approach to parallel and distributed real-time simulation, called the *distributed time-triggered simulation* (DTS) [Kim96b, Kim97b,

Kim99e]. The essence of this approach is briefly summarized and the advantages of using parallel computing platforms to perform DTS are discussed in this section.

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 running the simulation program). The ticking rate of the simulator clock in an RT simulator must be chosen such that during any ticking interval, all (real-time) events which should be simulated during that interval may be transparent to the user of the RT simulator and only the resulting state of the simulator at the end of the ticking interval may be seen by the user. Therefore, the microscopic order in which events are simulated during a ticking interval should be of no concern to the simulator user. This requirement is called *the simulator clock atomicity requirement* [Kim99e]. All computational activities taking place during a ticking interval of the simulator clock may be viewed as one *simulation-step*.

In distributed RT simulation, simulator objects are distributed among multiple nodes. *Synchronization of the simulation-steps of distributed simulator objects* is then a key challenge. In other words, a simulation-step executed by the distributed nodes as a group must include the activities necessary to keep the executions of the simulation-step by the nodes synchronized. Here the DTS approach which uses distributed TMOs of which SpMs execute simulation-steps has major advantages over other distributed simulation approaches, even if we assume that the latter approaches can be adapted somehow to enable RT simulation. This is because synchronization of SpMs does not require message exchanges among the host nodes (not counting the message exchanges which may be needed at a certain low frequency for resynchronizing 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 TMO to the territory covered by another TMO are inevitable. Therefore, the duration of a simulation-step must be long enough to cover this kind of message exchanges.

Since the precision of the RT simulator is inversely proportional to the duration of the simulation-step, it is desirable to use computing platforms yielding small message delays. Therefore, use of highly parallel computing platforms is an attractive approach. DTS using such platforms is considered a timely and potentially fruitful area for research in the future.

7. Conclusion

OO RT programming is a technology expected to flourish in this quarter of the 21st century. Currently, its

youthfulness is indicated by the insufficient availability of the support middleware and the associated API, let alone language compilers. The middleware providing fault-tolerant execution support is in its infancy. The advances in OO RT distributed programming will also enable large-scale RT simulations. The research community dealing with this technology area is expected to grow continuously for foreseeable future and consequent accelerations of the technology advances will in turn accelerate the development of many new types of sophisticated RT distributed computing applications.

Acknowledgements: The research work reported here was supported in part by the US Defense Advanced Research Project Agency under Contract N66001-97-C-8516 monitored by SPAWAR, and in part by the NSF Next-Generation Software (NGS) Program under Grant 99-75053.

References

- [Alc99] Alcatel et al., 'Realtime CORBA Joint Revised Submission', OMG Document Orbos/99-02-12, March 1, 1999, available from www.omg.org.
- [Att91] Attoui, A. and Schneider, M., "An Object Oriented Model for Parallel and Reactive Systems", *Proc. IEEE CS 12th Real-Time Systems Symp.*, 1991, pp. 84-93.
- [Ete99] Eternal Systems, et al, 'Dynamic Scheduling - Initial Submission,' OMG Document: orbos/99-10-06 ed., Oct. 25, 1999, available from www.omg.org.
- [IEE00] 'A special issue of Computer (a magazine of IEEE Computer Society) on Object-oriented Real-time distributed Computing', to appear in June 2000.
- [Ish92] Ishikawa, Y., Tokuda, H., and Mercer, C. W., "An Object-Oriented Real-Time Programming Language", *IEEE Computer*, October 1992, pp. 66-73.
- [ISO98] *Proc. ISORC '98 (IEEE CS 1st Int'l Symp. on Object-oriented Real-time distributed Computing, Kyoto)*, IEEE CS Press, April 1998.
- [ISO99] *Proc. ISORC '99 (IEEE CS 2nd Int'l Symp. on Object-oriented Real-time distributed Computing, St. Malo)*, IEEE CS Press, May 1999.
- [ISO00] *Proc. ISORC 2000 (IEEE CS 3rd Int'l Symp. on Object-oriented Real-time distributed Computing, Newport Beach)*, IEEE CS Press, March 2000.
- [J-C00] J Consortium, "Real-Time Core Extensions for the Java Platform", Specification No. T1-00-01, Rev. 1.0.10, available from www.j-consortium.org, Feb. 3, 2000.
- [Jen00] Jensen, E.D., "Distributed Real-Time Java: A Proposal", *Proc. ISORC 2000*, Newport Beach, March 2000, pp.2-6.
- [Kim93] Kim, K.H. and Shokri, E.H., "Minimal-Delay Decentralized Maintenance of Processor-Group Membership in TDMA-Bus LAN Systems", *Proc. IEEE CS 13th Int'l Conf. on Distributed Computing Systems*, Pittsburg, May 1993, pp. 410-419.
- [Kim94a] Kim, K.H., "Action-Level Fault Tolerance", Ch. 17 in Sang H. Son, 'Advances in Real-Time Systems', Prentice Hall, 1994, pp.415-434.
- [Kim94b] Kim, K.H. et al., "Distinguishing Features and Potential Roles of the RTO.k Object Model", *Proc. WORDS '94 (IEEE CS '94 Work. on Object-Oriented Real-Time Dependable Systems)*, Oct. 1994, Dana Point, pp.36-45.
- [Kim95] Kim, K.H., "The Distributed Recovery Block Scheme", Ch. 8 in Michael R. Lyu, ed., 'Software Fault Tolerance', 1995, pp. 189-209.
- [Kim96a] Kim, K.H. et al., "The DREAM Library Support for PCD and RTO.k programming in C++", *Proc. WORDS '96*, Laguna Beach, Feb. 96, pp. 59-68.
- [Kim96b] Kim, K.H., Nguyen, C., and Park, C., "Real-Time Simulation Techniques Based on the RTO.k Object Modeling", *Proc. COMPSAC '96 (IEEE CS '96 Software & Applications Conf.)*, Seoul, August 1996, pp. 176-183.
- [Kim97a] Kim, K.H., and Subbaraman, C., "Fault-Tolerant Real-Time Objects", *Communications of the ACM*, January 1997, pp. 75-82.
- [Kim97b] Kim, K.H., "Object Structures for Real-Time Systems and Simulators", *IEEE Computer*, Vol. 30, No.8, August 1997, pp. 62-70.
- [Kim98a] Kim, K.H. and Subbaraman, C., "Principles of Constructing a Timeliness-Guaranteed Kernel and the Time-triggered Message-triggered Object Support Mechanism", *Proc. ISORC '98 (IEEE CS 1st Int'l Symp. on Object-oriented Real-time distributed Computing)*, Kyoto, Japan, April 1998, pp. 80-89.
- [Kim98b] Kim, K.H., "ROAFTS: A Middleware Architecture for Real-time Object-oriented Adaptive Fault Tolerance Support", *Proc. HASE '98 (IEEE CS 1998 High-Assurance Systems Engineering Symp.)*, Washington, D.C., Nov. 1998, pp.50-57.
- [Kim99a] Kim, K.H. and Subbaraman, C., "Dynamic Configuration Management in Reliable Distributed Real-Time Information Systems", *IEEE Trans. on Knowledge and Data Engr.*, Vol.11, No.1, Jan./Feb. 1999, pp.239-254.
- [Kim99b] Kim, K.H., "Real-Time Object-Oriented Distributed Software Engineering and the TMO Scheme", *Int'l Jour. of Software Engineering & Knowledge Engineering*, Vol. 9, No.2, April 1999, pp.251-276.
- [Kim99c] Kim, K.H. Ishida, Masaki, Liu, Juqiang, "An Efficient Middleware Architecture Supporting Time-Triggered Message-Triggered Objects and an NT-based Implementation", *Proc. ISORC '99 (2nd IEEE CS Int'l Symp. on Object-Oriented Real-time Distributed Computing)*, St. Malo, France, May, 1999, pp.54-63.
- [Kim99d] Kim, K.H., "Questionable Relevancy of Fixed Priority Assignment in Distributed Object Design", *Proc. ISORC '99*, May 1999, pp. 283-285 (Position paper).
- [Kim99e] Kim, K.H., Liu, J., and Ishida, M., "Distributed Object-Oriented Real-Time Simulation of Ground

Transportation Networks with the TMO Structuring Scheme", *Proc. COMPSAC '99* (IEEE CS Computer Software & Applications Conf.), Phoenix, AZ, Oct. 1999, pp.130-138.

[Kim99f] Kim, K.H., Liu, J.Q., Miyazaki, H., and Shokri, E.H., "A CORBA Service Enabling Programmer-Friendly Object-Oriented Real-Time Distributed Computing", *Proc. of WORDS '99F (IEEE CS 5th Workshop on Object-oriented Real-time Dependable Systems)*, Monterey, Nov. 1999, pp.101-107.

[Kim99g] Kim, K.H., "Group Communication in Real-Time Computing Systems: Issues and Directions", *Proc. FTDCS '99 (7th IEEE Workshop on Future Trends of Distributed Computing Systems)*, Cape Town, South Africa, Dec. 1999, pp.252-258.

[Kim00a] Kim, K.H. and Subbaraman, C., "The PSTR/SNS Scheme for Real-Time Fault Tolerance via Active Object Replication and Network Surveillance", to appear in *IEEE Trans. on Knowledge and Data Engr.*, Jan./Feb. 2000.

[Kim00b] Kim, K.H., and Liu, J. "Deadline Handling in Real-Time Distributed Objects", *Proc. ISORC 2000*, Newport Beach, CA, March 2000, pp.7-15.

[Kim00c] Kim, K.H., "Real-time Object-oriented Distributed Computing", in John G. Webster ed., *'Encyclopedia of Electrical & Electronics Engineering'*, Supplementary Volume, John Wiley & Sons, 2000.

[Kim00d] Kim, K.H., "APIs Enabling High-Level Real-Time Distributed Object Programming", to appear in *IEEE Computer*, June 2000.

[Kop89] Kopetz, H., et al., "Fault-Tolerant Membership Service in a Synchronous Distributed Real-Time System.", *Proc. IFIP WG 10.4 Conf. on Dependable Computing for Critical Appl.*, Santa Barbara, Aug. 1989, pp.167-174.

[Kop90] Kopetz, H. and Kim, K.H., "Temporal Uncertainties in Interactions among Real-Time Objects", *Proc. IEEE CS 9th Symp. on Reliable Distributed Systems*, Oct. 1990, pp.165-174.

[Nar99] Narasimhan, P., Moser, L. E., and Melliar-Smith, P. M., "Using Interceptors to Enhance CORBA," *IEEE Computer*, July 1999, pp. 62-68.

[OMG99a] Object Management Group, "Dynamic Scheduling Request For Proposal", OMG Document: Orbos/99-03-32, March 26, 1999.

[OMG99b] Object Management Group, *'The Common Object Request Broker: Architecture and Specification'*, Revision 2.3.1, Oct. 1999, available from www.omg.org.

[Ran95] Randell, B., and Xu, Jie, "The Evolution of the Recovery Block Concept", Chap. 2 in *'Software Fault Tolerance'*, Michael R. Lyu, Ed., 1995, pp. 1-21.

[Ren99] Ren, J. et al., "Building Dependable Distributed Applications Using AQUA", *Proc. 4th IEEE Symp. on*

High Assurance Systems Engineering (HASE'99), Washington D.C., Nov. 17-19, 1999, pp. 189-196.

[RJE00] Real Time Specification for Java Experts Group, "Real-time Specification for Java, Version 0.9.2", available from www.rtfj.org/public, March 29, 2000.

[Sch98] Schmidt, D.C., Levine, D.L., and Mungee, S., "The Design and Performance of Real-Time Object Request Brokers," *Computer Communications*, vol.21, pp.294-324, Apr. 1998.

[Ses97] Sessions, R., *'COM & DCOM; Microsoft's Vision for Distributed Objects'*, John Wiley & Sons, Inc., New York, Oct. 1997.

[Sho98a] Shokri, E., Crane, P., and Kim, K.H., "An Implementation Model for Time-Triggered Message-Triggered Object Support Mechanisms in CORBA-Compliant COTS Platforms", *Proc. ISORC '98 (IEEE CS 1st Int'l Symp. on Object-oriented Real-time distributed Computing)*, Kyoto, Japan, April 1998, pp. 12-21.

[Sho98b] Shokri E, Crane, P., and Kim, K.H., and Subbaraman, C., "Architecture of ROAFTS/Solaris: A Solaris-based Middleware for Real-Time Object-Oriented Adaptive Fault Tolerance Support", *Proc. COMPSAC '98 (IEEE CS 22nd Int'l Computer Software & Applications Conf.)*, Vienna, Austria, August 1998, pp.90-98.

[Sho98c] Shokri, E., Hecht, H., Crane, P., Dussault, J., and Kim, K.H., "An Approach for Adaptive Fault-Tolerance in Object-Oriented Open Distributed Systems", *Int'l Jour. of Software Engineering & Knowledge Engineering*, Vol.8, No.3, pp.333-346.

[Sol95] Soley, R. ed., *'Object Management Architecture Guide'*, 3rd edition, John Wiley & Sons, New York, 1995.

[Sun98] Sun Microsystems, "Java Remote Method Invocation Specification", Revision 1.50, JDK 1.2, Oct. 1998, available from web2.java.sun.com/products/jdk/1.2/docs/guide/rmi/spec/rmi-title.doc.html.

[Tac98] Tachikawa, T., Higaki, H., and Takizawa, M., "Group Communication Protocols for Realtime Applications", *Proc. 18th IEEE ICDCS*, 1998, pp.40-47.

[Tak92] Takashio, K., and Tokoro, M., "DROL: An Object-Oriented Programming Language for Distributed Real-Time Systems", *Proc. OOPSLA*, 1992, pp. 276-294.

[WOR94] *'Proc. WORDS '94 (IEEE CS's 1st Workshop on Object-oriented Real-time Dependable Systems. Oct. '94, Dana Point)'*, IEEE CS Press, 1995.

[WOR96] *'Proc. WORDS '96 (Laguna Beach, Feb. '96)'*, IEEE CS Press, 1996.

[WOR97] *'Proc. WORDS '97 (Newport Beach, Feb. '97)'*, IEEE CS Press, 1997.

[WOR99] *'Proc. WORDS '99 (Santa Barbara, Jan. '99)'*, IEEE CS Press, 1999.

[WOR99F] *'Proc. WORDS '99F (Monterey, Nov. '99)'*, IEEE CS Press, 1999.

Proceedings

Seventh International Conference on Parallel and Distributed Systems

4-7 July 2000

Iwate, Japan

Co-Sponsored by

Iwate Prefectural University, Japan

In cooperation with

Takizawa Village, Japan

Morioka City, Japan

Iwate Prefecture, Japan

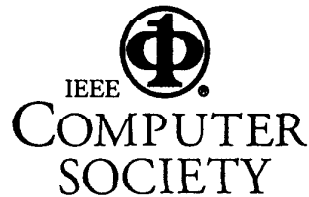
Communications Research Laboratory (CRL) of Ministry of Post Office, Japan

IEEE Taipei Section, Taiwan

Information Processing Society of Japan (IPSJ), Japan

Edited by

Professor Makoto Takizawa



Los Alamitos, California

Washington . Brussels . Tokyo
