ROAFTS: A CORBA-Based Real-Time Fault Tolerance Support Middleware

Eltefaat Shokri
SoHaR Inc.

Patrick Crane
SoHaR Inc.

Jerry Dussault
USAF, Rome Lab.

Kane Kim
University of California
Irvine

Chittur Subbaraman
University of California
Irvine

Abstract

*Middleware implementation of various critical services, required by large-scale and complex real-time applications on top of COTS operating systems, has the tremendous advantage of reducing the complexity of the design and implementation by separating the concerns of the application designer for the application functionality from the concerns for application-independent system issues. In addition, such a middleware must preferably be structured in an object-oriented fashion owing to the modularity and natural abstraction benefits object-orientation brings in. This paper presents the results of a research effort on the development of the **R**eal-time **O**bject-oriented **A**daptive **F**ault **T**olerance **S**upport (**ROAFTS**) middleware on top of a COTS operating systems, Solaris, and a COTS CORBA-complaint ORB, Orbix. The supported real-time application consists of a network of Time-triggered Message-triggered Objects (TMO's), the TMO being a major extension of a conventional object for use in real-time applications.*

1. Introduction

Many emerging large-scale safety-critical applications are highly distributed, and must operate in highly dynamic environments with varying timing, functional, and reliability requirements [Kim92]. The robust design and implementation of these complex systems is a difficult challenge and should employ the state of art in both software engineering and distributed systems design methodologies. The following principles may assist system designers in the realization of such complex systems:

- **Middleware Implementation**: In order to reduce the complexity of the design and the implementation of emerging larger-scale applications, it is imperative to maximally separate the concerns for application functionality from the concerns for other system attributes such as "guaranteeing required reliability", and "low-level inter-component communication facilities". In other words, support subsystems for application-independent needs, such as distributed fault tolerance, can be built as a separate layer represented by a minimal set of well-defined application-interfaces such that the application designer can utilize the provided support facilities without concerns on how these supports are built [Sho97].
- **Real-Time Object-oriented Structuring**: Experiences in designing large applications over the past decade have demonstrated that object-oriented design and implementation techniques [Boo94] have good potentials for reducing the design complexity and increasing the robustness and maintainability of the resulting systems. Recently, there have been efforts [Kim94a, Kim94b, Kim97b] to extend the object-oriented structuring to effectively deal with the main concerns of the real-time systems (i.e., timing characteristics for objects). We strongly believe the use of real-time object structuring approaches are essential for design of reliable, reusable, extensible, and maintainable applications.
- **Use of Standards and COTS Components**: It is now a well-known fact that commercial-off-the-shelf (COTS) components, built in conformance with widely-accepted standards, are significantly more reliable than custom-built parts and their ownership costs are dramatically decreasing. One of the well-established software component standards of growing importance is Object Management Group's CORBA, a location-transparent language-independent inter-object communication architecture [OMG95]. CORBA's Object Request Broker (ORB) provides a high-level abstraction for communications among objects residing in different hosts.
- **Adaptability**: Since some emerging applications must operate in highly dynamic environments and the available execution resources may change dynamically, it is highly useful to make the applications, as well as the underlying execution systems, adapt to dynamic changes in the application environments, the available resources, or the application requirements.

This paper presents the architecture and the implementation model for a version of the **R**eal-time **O**bject-oriented **A**daptive **F**ault **T**olerance **S**upport (**ROAFTS**) middleware being developed under the sponsorship of USAF Rome Laboratory. The middleware runs on top of a COTS operating system, Solaris, and a COTS CORBA-complaint ORB, Orbix.

Figure 1 depicts major components of the ROAFTS. Solaris was chosen as the base operating system because it provides some real-time application support with a real-time threads mechanism. Iona's Orbix provides an efficient and lightweight implementation of a multi-threaded CORBA 2.0-compliant ORB. Time-triggered Message-triggered Object (TMO) [Kim94a, Kim97b] is a major extension of conventional objects which enables the object-oriented (OO) system designers to specify timing characteristics of objects and their methods and data elements. Correct and timely execution of the TMO's according to their specifications is the main responsibility of the TMO Management layer. The Adaptation Management layer recognizes any major changes in either the environment or the available execution resources and reconfigures the system to cope with the current state of the system/environment in an optimal execution mode [Sho97].

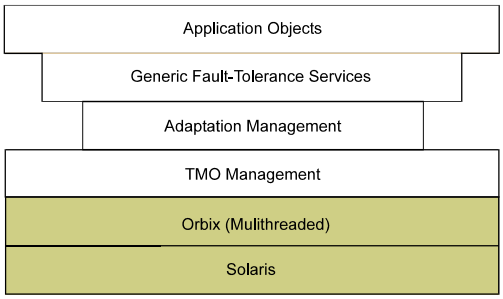

Figure 1. Components of the ROAFTS

A suitable AFTM middleware should enhance system reliability, performance, survivability, and effectiveness through the following essential capabilities:

2. Major design and execution schemes supported

2.1 The TMO structuring scheme

Several years ago, an abstract precursor of the *Time-triggered Message-triggered Object (TMO) model* for cost-effective development of *hard-real-time* systems was proposed [Kop90]. In recent years, the TMO, formerly called the RTO.k object, has been formalized to possess a concrete syntactic structure and execution semantics [Kim94a, Kim94b].

The basic structure of the TMO is shown in Figure 2. It is an extension of the conventional object model(s) in three major ways:

(a) Spontaneous method: The TMO contains a new type of methods, time-triggered (TT-) methods, also called the spontaneous methods (SpM's), which are clearly separated from the conventional service methods (SvM's). The SpM executions are triggered when the real-time clock reaches specific values determined at design time whereas the SvM executions are triggered by service request messages from clients.

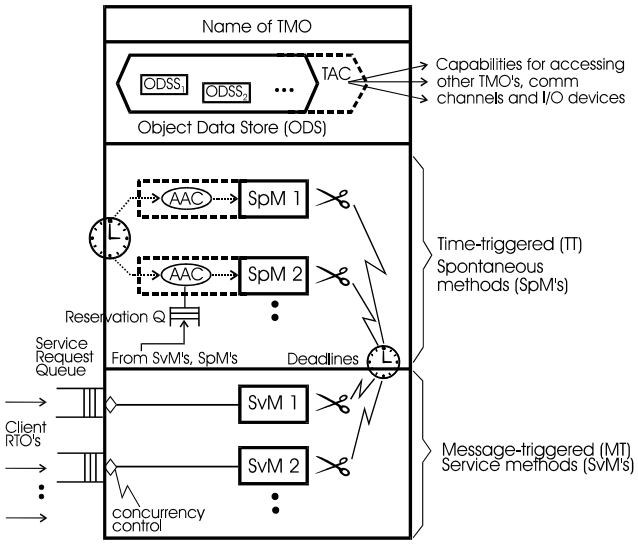

Figure 2. The TMO structure (Adapted from [Kim94a])

Triggering times for SpM's 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 requests future executions of a specific SpM. The AAC specifying candidate triggering times rather than actual triggering times starts with a declaration "if-demanded".

(b) Basic concurrency constraint (BCC): Under this rule, called the basic concurrency constraint (BCC), SvM's cannot disturb the executions of SpM's and the designer's efforts in guaranteeing timely service capabilities of TMO's are greatly simplified. 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 if no SpM that accesses the same portion of the object data store (ODS) to be accessed by this SvM has an execution time window that will overlap with the execution time window of this SvM. However, the BCC does not stand in the way of either concurrent SpM executions or concurrent SvM executions.

(c) A deadline is imposed for each output action and completion of a method of a TMO.

Extensions (a) and (b) are unique to the TMO model in comparison with other object models [Tak92]. Client methods (SpM's or SvM's) may request for service of SvM's in other TMO's.

The designer of each TMO indicates the *deadline for every output* produced by each SvM (and each SpM which may be executed on requests from SvM's) in the specification of the SvM (and some relevant SpM's) and advertises this to the designers of potential client objects. The designer of the server object thus guarantees the timely services of the object. Before determining the deadline specification, the server object designer must convince himself that with the available object execution engine (hardware plus operating system), the server object can be implemented to always execute the SvM such that the output action is performed within the deadline.

2.2 The distributed recovery block scheme

The distributed recovery block (DRB) scheme is an approach for realizing hardware and software fault tolerance in real-time distributed systems. The underlying design philosophy behind the DRB scheme is that a real-time distributed application can take the desirable modular form of an interconnection of *computing stations*, where a computing station refers to a processing node (hardware and software) dedicated to the execution of one or a few application tasks [Kim94c]. Some computing stations perform system resource management functions only.

In its basic configuration, a *DRB computing station* consists of two processing nodes executing two functionally equivalent tasks. The first node is called the primary node and the second node is called the shadow node. The task software in the primary node, as well as that in the shadow node, is constructed by use of the recovery block language construct [Ran95]. A recovery block may consist of multiple versions of a task procedure and an acceptance test (AT) function designed to judge the reasonableness of the results produced by each version. Such versions are called try blocks. A try (i.e., execution of a try block) is always followed by an AT execution. A try not completed within the maximum execution time allowed for each try block, due to hardware faults or excessive looping, is also treated as an AT failure. Therefore, the AT is a combination of both *logic* and *time* AT's.

In most cases a recovery block containing just two try blocks, a *primary try block* and an *alternate try block*, is designed. The roles of the two try blocks are assigned differently in the two partner nodes. The governing rule is that *the primary node tries to execute the primary try block whenever possible, whereas the shadow node tries to execute the alternate try block*. Therefore, the primary node X uses the primary try block A as the first try block initially, whereas the shadow node Y uses the alternate try block B as the initial first try block. Until a fault is detected, both nodes receive the same input data, process the data using two different try blocks, and check the results using the AT concurrently. After the primary node applies its AT, it informs the shadow partner of the AT result. Then, the primary node delivers the data processing results to the successor computing station(s) while the shadow node skips the corresponding output step and waits for an *output success notice* from its partner.

If the primary node fails and the shadow node passes its own AT, the shadow node learns of the failure of the primary partner either from a failure report from the partner or through a time-out for a report. Then the shadow immediately changes its role to that of the primary and delivers its processing results to the successor computing station(s). Meanwhile, the primary node, if alive, will attempt to become a useful shadow without disturbing the (new) primary node; it attempts to roll back and retry with its second try block *B* to bring its application computation state, including its local database, up-to-date.

2.3 The supervisor-based network surveillance (SNS) scheme

Network surveillance schemes facilitate the fast learning by each fault-free node in the system of fault occurrences or rejoin events occurring in other parts of the system. Recently [Kim97c], a supervisor-based network surveillance (SNS) scheme was devised which is applicable to a variety of point-to-point networks. Figure 3 shows the basic operation of the SNS scheme. As shown in the figure, there are two types of nodes that execute the SNS scheme, the *worker* nodes and a *supervisor* node. The worker nodes are mainly responsible for judging their own health status, the health status of their neighbor nodes, and the health status of the links attached to themselves. The supervisor node performs all the duties that a worker normally does. In addition, it is responsible for collecting *fault suspicion reports* from worker nodes, using the collected information to judge whether a fault has indeed occurred, and then sending the *fault occurrence notice* to all the healthy worker nodes in the system. In case the current supervisor is judged to be faulty by the healthy neighbors, they participate in a new supervisor election and the newly elected supervisor informs all the healthy worker nodes about changes in the supervisor node.

2.4 The CORBA Architecture

The underlying philosophy of CORBA is to provide a common architectural framework across heterogeneous hardware platforms, operating systems, and inter-node communication protocols. The core of the CORBA architecture is the Object Request Broker (ORB), a mechanism that facilitates transparency of object location, activation, and communication.

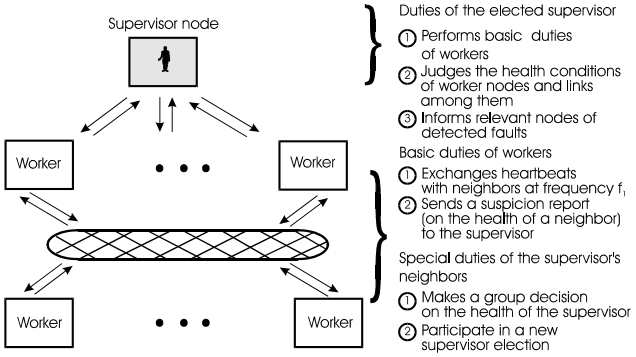

Figure 3: The SNS scheme (Adapted from [Kim97c])

The function of the ORB is to deliver requests of clients to objects and return output values (if any) back to the clients. Clients do not need to know where in the network server objects reside, how they communicate, or how they are implemented.

3. TMO-based implementation of ROAFTS middleware

In this section, major issues in the implementation of various ROAFTS components will be discussed. As shown in Figure 4, ROAFTS is composed of:

- *TMO Support Manager*: This is responsible for supporting the timely execution of TT-methods and message-triggered methods of the registered TMO's. A TMO and its methods are registered with the TMO Support Manager. The registration involves recording essential characteristics, such as activation conditions for TT-methods and method completion deadlines.
- *TMO structured Network Surveillance Manager (NSM)*: NSM is structured as a TMO and is composed of TT-methods for sending heartbeats to the partner nodes and investigating the health-status of the neighbor nodes.
- *TMO structured Generic Fault-Tolerance Service (GFTS)*: GFTS is a component which provides support for different fault tolerance schemes such as the DRB scheme, the sequential recovery block scheme [Ran95], and a forward-recovery oriented exception-handling scheme. GFTS has the capability to receive adaptation commands (from the adaptation management layer or the user) to switch from one fault-tolerance mode to another. GFTS provides user-transparent adaptable fault-tolerance services.

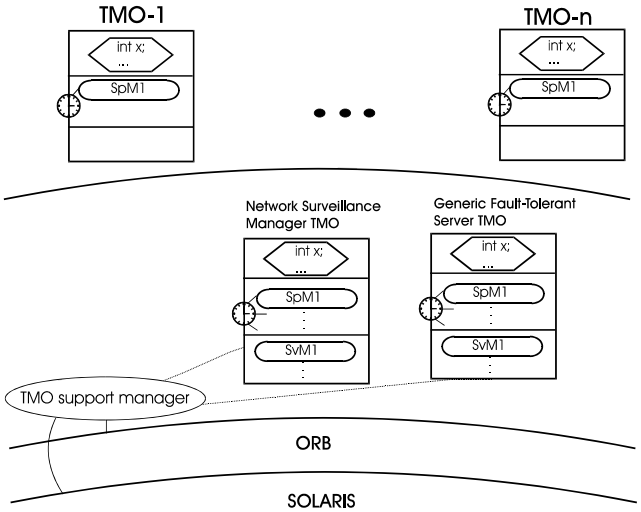

Figure 4: TMO-based middleware implementation

Due to the limitation of the current GFTS, the application class is currently structured as a simple TMO composed of only one TT-method. The main reason for this restriction is that providing full application-

transparent fault tolerance to a conventional TMO requires additional supports [Kim97a] which have not yet been implemented into the GFTS.

3.1 TMO support manager implementation

As mentioned earlier, the TMO Support Manager mainly supports the timely execution of TMO's. A TMO registers its methods with the TMO Support Manager to facilitate a timely activation and execution of the registered methods. As presented in Figure 5, the TMO Support Manager is implemented using the following threads:

- *TMO Manager thread*: The TMO Manager thread handles scheduling and the execution of all ROAFTS threads. It is implemented as a Solaris real-time thread and is periodically activated by a software watchdog timer. Since a Solaris real-time thread has the highest possible priority and is guaranteed to be executed immediately after its activation, the TMO Manager thread provides a degree of predictability for the periodic execution of thread management functionality. However, for several reasons discussed in section 4, complete timing predicability of the application is not possible in the Solaris environment.

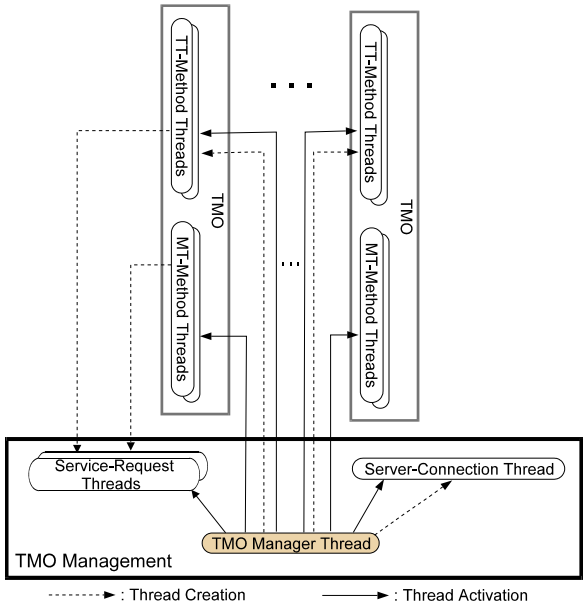

Figure 5: Threads in the TMO Support Manager

- *Server-Connection thread:* A CORBA server requires a thread to be introduced as an interface between the server process and the ORB. The Server-Connection thread supporting this interface must periodically become active to handle all incoming and outgoing CORBA service requests.

- *Service-Request threads*: In order to allow concurrency in both client and server objects, we adopted the approach of creating and dedicating a thread to the chore of initiating a service-request in the client object. Whenever a client TMO attempts to call a remote TMO (or CORBA) method, the TMO support manager creates a Service-Request thread dedicated to making the remote method call and block until the result is returned from the server object.

As briefly discussed in Section 2, (message-triggered) service methods are activated by a call from a client object. The remote activation of an object method must be intercepted by the TMO Support Manager thread to enforce the timely thread management. This interference is made possible by the "filter" facility introduced by Orbix. Orbix allows the application designer to provide additional code, introduced in the form of a filter [Orb95] to be executed before or after the server program is executed.

Under ROAFTS, when the ORB in the server host receives a service request from a remote object, it first executes the filter code to (i) create a thread for executing the service request but keep the thread in the suspended mode, and (ii) register the corresponding service method with the middleware. The execution of the service request will then be initiated (based on its timing characteristics) by the TMO Manager thread at an appropriate point of time. When the service execution is completed, the result will be forwarded back to the client object through the Server-Connection thread.

3.2 TMO-based implementation of SNS scheme

As discussed earlier, the network surveillance manager (NSM) is an important component of the middleware architecture and provides fast fault detection notices to the generic fault-tolerance server. In our current implementation, the NSM has been implemented as a TMO which uses ORB in order to exchange messages with peer NSM's executing in other nodes. Figure 6 shows the TMO structure of the NSM. The NSM has two periodic SpM's, called the *Heartbeat Generator* and the *Heartbeat Analyzer*. The *Heartbeat Generator* SpM is responsible for periodically generating heartbeat signals and using ORB services to send it out to the neighbor nodes. The *Heartbeat Analyzer* SpM is responsible for analyzing the heartbeat signals received from the neighbor nodes and for informing the Generic Fault Tolerance Server (GFTS) of any detected fault occurrences. This TMO also has two SvM's, the *Heartbeat receiver SvM* and the *Heartbeat frequency modifier* SvM. The former SvM is responsible for receiving the heartbeat signals from neighbor nodes and

updating an ODS segment (ODSS). The latter SvM is responsible for receiving a command from an adaptation manager regarding changes to the fault-tolerance execution modes.

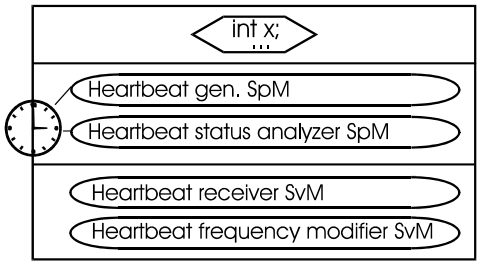

Figure 6: TMO structure of NS manager

3.3 TMO-based implementation of the generic fault-tolerance service

In this section, we wil first discuss the way application-transparent fault-tolerance provisions were realized. Then the TMO structuring of the generic fault-tolerance services will be presented.

It is desirable to separate fault management concerns from the concerns for the functionalities of the applications. However, there are some fault management elements which are application-specific. For example, a non-trivial and realistic acceptance test varies from one application to another. Therefore, it is useful to isolate application-independent elements from application-specific ones. Using object-oriented design and programming technology, it is feasible to capture the application-independent features as a base class such that each application can seamlessly inherit these features and add application-specific features in application-dependent derived classes [Sho96]. In ROAFTS middleware the Generic Fault-Tolerance Service Layer provides fault-tolerance services as a set of abstract classes from which the customized objects can be constructed.

As shown in Figure 7, the Generic Fault Tolerance Server (GFTS) is structured as a TMO with two SpM's and three SvM's. *GFT Executive* SpM is responsible for executing the associated application using the selected fault tolerance scheme (which is given by either the user or the adaptation manager). The selected fault tolerance scheme can be one of the following: (i) the Distributed Recovery Block (DRB) scheme, (ii) the Sequential Recovery Block scheme, or (iii) a forward recovery oriented Exception Handling scheme. *External Command Interpreter* SpM periodically checks for a command from an external adaptation manager for changing the current fault tolerance mode or a command from the NSM for any fault detection notice. If it finds such a command, it notifies the *GFT Executive* SpM via a signal deposited in an ODSS.

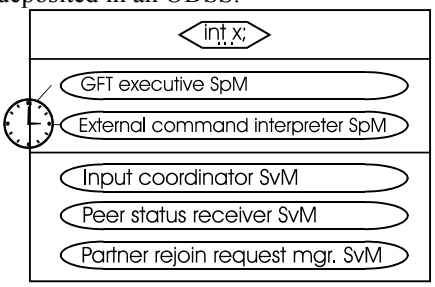

Figure 7: TMO structure of generic fault-tolerant server

Input Coordinator SvM is responsible for receiving the input sent to this GFTS and queues the input in an ODSS to be processed by the *GFT Executive* SpM. In schemes such as the DRB scheme in which different versions of an application task execute in two or more nodes, it is imperative for the participant nodes to coordinate their execution. *Peer Status Receiver* SvM is responsible for doing this. Each GFTS sends its partner GFTS status messages such as data ID, acceptance test result, etc. by remotely calling the partner's *Peer Status Receiver* SvM. Finally, in order to facilitate an orderly joining of a new partner, a newly joined GFTS requests its active partner to send the recent system state, by calling *Partner Rejoin Request* SvM of the partner GFTS.

4. Discussions and conclusions

Our experiences have shown that with the current implementation of ROAFTS, the development of real-time applications with time granularity in the range of 40 - 100 msec can be efficient, if the local area network executing ROAFTS is isolated from outside world (e.g., Internet). The time-slice for the thread scheduling is currently chosen to be 100 msec to satisfy the needs of target applications such as military distributed planing applications. Although ROAFTS's current predictability is weaker than those of the implementations which are conducted in less open environments, such as isolated intra-nets, we believe ROAFTS can be effectively used in a wide range of soft real-time applications, such as military distributed planning applications.

ROAFTS implementation provides us insight into two important issues: (i) Shortcomings and potentials of the Solaris operating system in the real-time setting, and (ii) effectiveness of the current standard CORBA when used in real-time applications. We believe that system designers must consider these important issues carefully in order to effectively use COTS operating systems and CORBA middleware for real-time applications.

As mentioned earlier, Solaris has added multi-threading and real-time thread capabilities to traditional Unix in order to extend Unix-like operating systems for use in real-time applications. While being effective in soft real-time applications, Solaris still suffers from several shortcomings. One major shortcoming is that when a real-time thread issues a "slow" system call (The system calls which cause the caller to be blocked until the requested service is completed such as pipe inter-process communication, sock-based communications, and I/O services), the execution resources are shared by other non-real-time processes and/or threads during the execution of the system call. In other words, if an application running in the real-time mode issues such system calls, its timing behavior cannot be predicted.

Another lesson learned from the ROAFTS implementation is related to the applicability of the CORBA-compliant ORB's for real-time applications. Although the CORBA provides a clean high-level abstraction for location-transparent language-independent inter-object communications, it must be extended in the future to deal with the following two requirements:

- A majority of current implementations are built on top of non-real-time operating systems and non-real-time communication protocols. If the underlying operating system and communication subsystem cannot guarantee predictable services, the CORBA middleware will not be able to guarantee timely interaction among remote objects as well as local objects.
- Even if CORBA is built on top of real-time operating systems, it will not be able to guarantee timely interacton among objects, mainly because the application-designer does not have the means for defining timing characteristics of the applications such as completion deadline for service-requests (method calls).

Object Management Group has recognized the need to define a new standard real-time CORBA for real-time applications [OMG96]. However, it should be noted that the real-time application domain ranges over a wide spectrum of various applications with different levels of needed predictabilities. Therefore, this fact should be reflected in defining a real-time CORBA standard.

7. References

[Boo94] Booch, G., *Object Oriented Analysis and Design with Applications*, Redwood City, CA: Benjamin/Cummings Publishing, 2nd ed., 1994.

[Kim92] Kim, K. H., and Lawrence, T., "Adaptive Fault-Tolerance in Complex Real-Time Distributed Applications", *Computer Communication*, Vol. 15, No. 4, May 1992, pp. 243-251.

[Kim94a] Kim, K.H. et al., "Distinguishing Features and Potential Roles of the RTO.k Object Model", *Proc. 1994 IEEE CS Workshop on Object-oriented Real-time Dependable Systems (WORDS)*, Oct. 94, Dana Point, pp.36-45.

[Kim94b] Kim, K. H., and H. Kopetz, "A Real-Time Object Model RTO.k and an Experimental Investigation of Its Potentials", *Proc. IEEE Computer Society's 1994 International Computer Software and Applications Conference*, November 1994, Taipei, pp. 392-402.

[Kim94c] Kim, K.H., "Action-level Fault Tolerance", Ch. 17 in Sang H. Son ed., *'Advances in Real-Time Systems*', Prentice Hall, 1994, pp. 415-434.

[Kim97a] Kim, K. H., and Subbaraman, C., "Fault-Tolerant Real-Time Objects", *Communications of ACM,* Vol.40, No. 1, Jan. 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.

[Kim97c] Kim, K.H., and Subbaraman, C., "A Supervisor-Based Semi-Centralized Network Surveillance Scheme and the Fault Detection Latency Bound", to appear in Proc. 16th Symposium on Reliable Distributed Systems, October 1997.

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

[OMG95] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, Revision 2.0, 1995.

[OMG96] OMG Real-Time Interest Group, "Real-Time CORBA Issue 1.0", Working Document, OMG, 1996.

[Orb95] Orbix Programming Guide, *IONA Technologies*, Dublin, 1995.

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

[Sho96] Shokri E. and Tso, K. S., "Development of Software Fault-Tolerant Applications with Ada95 Object-Oriented Support", *The National Aerospace and Electronics Conference*, Dayton, OH, May 1996, pp.519-526.

[Sho97] Shokri, E., et al., "An Approach for Adaptive Fault Tolerance in Object-Oriented Open Distributed Systems", Proc. 3rd Workshop on Object-Oriented Real-Time Dependable Systems, Newport Beach, CA, February 1997, pp.

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