Proc. ISORC 2000 (IEEE CS 3rd Int'l Symp. on Object-oriented Real-time distributed Computing), Newport Beach, CA, March 2000, pp.7-15.

Deadline Handling in Real-Time Distributed Objects

K. H. (Kane) Kim
kane@ece.uci.edu

Juqiang Liu
jqliu@ece.uci.edu

Moon-Hae Kim
moonkim@dream.eng.uci.edu

University of California
Irvine, CA 92697, U.S.A.

Abstract: Deadline handling is a fundamental part of real-time computing but has been practiced in ad hoc forms for decades. A general framework for systematic deadline handling in real-time distributed computer systems is proposed in this paper. The notions of hard deadlines and hard-real-time program components are discussed along with the advantages of a hard-real-time component based construction approach. To present approaches for implementation of systematic deadline handling in concrete forms, we use the *time-triggered message triggered object* (TMO) network structuring as the basic design framework in which deadline handling approaches are incorporated. The TMO structuring scheme is a general-style component structuring scheme and supports design of all types of components including hard-real-time objects and non-real-time objects within one general structure. An augmentation of the TMO structure with statistical performance indicators is also proposed.

Keywords: real time, deadline, service time, guarantee, TMO, time-triggered, message triggered, object.

1. Introduction

Deadline handling has been practiced in ad hoc forms for several decades but has not been established in a general form [Kop97, Son94]. This has been the case even for single-node systems . In real-time (RT) distributed computer systems (DCSs), deadline handling is a much more complicated issue [J-C99, Kim97, OMG99, RJG99, Sel99]. First, deadlines often need to be imposed on computations involving actions by multiple nodes and the communication facilities. Secondly, large-scale systems must be constructed in a modular systematic manner and thus deadlines must also be imposed in a modular form.

In this paper, we propose one general broadly applicable framework for systematic deadline handling including specification and design of capabilities for detecting and responding to deadline violations. To make discussion on implementation approaches to be concrete, we use the *time-triggered message triggered object* (TMO) network structuring [Kim97] as the basic design framework in which deadline handling approaches are incorporated. The TMO structuring scheme has been established to remove the major limitations of conventional object structuring techniques in developing RT applications. It is a natural and syntactically small but semantically powerful extension of the conventional object-oriented (OO) design and implementation techniques. It allows the system designer to abstractly and yet accurately specify timing characteristics of data and function components of distributed computing objects.

The rest of the paper is organized as follows. Section 2 introduces some basic concepts such as hard deadline, guaranteed service time, and the client's deadline. A general framework for systematic deadline handling is then presented. The TMO scheme is reviewed briefly in Section 3. In Section 4, an implementation model for deadline handling in TMO networks is presented. Section 5 is a conclusion of the paper.

2. General framework for systematic deadline handling

2.1 Hard deadline and fault tolerance

An attitude frequently taken by system engineers has been that:

(1) Deadlines can be violated if a violation does not occur too many times for a certain period, and

(2) when a violation occurs, the system can continue, restart from the beginning, or roll back to a recovery point.

This assumes that the costs of a deadline violation are not that high and certainly not as high as the loss of large economic properties or human lives.

Certain deadlines cannot be treated so lightly. 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 deadline. Such a deadline is called a *hard deadline*.

Therefore, in principle, failing to meet hard deadlines is to fail in the critical application mission and thus it must be avoided via all available means. In this type of situations where system engineers do their utmost best, the main unavoidable sources of deadline violations are hardware faults or unreliable operating systems (OSs). To avoid the dangers presented by unreliable OSs, system

0-7695-0607-0/00 $10.00 © 2000 IEEE

engineers dealing with *hard-real-time (HRT) applications*, i.e., applications subject to hard deadlines, have tended to use minimal-service OSs which can be simply tested and whose timing behavior can be predicted with a high degree of precision. Let us consider for a while such environments where use of minimal-service OSs is acceptable. Hardware faults must still be dealt with and thus use of fault tolerance mechanisms which make up for the weaknesses of hardware has been a popular practice except where the reliability of hardware components is trusted to be sufficiently high.

To be useful in HRT applications, fault tolerance mechanisms must be of the type that incurs "small bounded overhead" during normal fault-free operational periods and recovers both the system and the on-going application within a small bounded period after a fault occurrence. Such an *RT fault tolerance* technology applicable to distributed HRT systems has advanced slowly and only in recent years practically effective technologies have started appearing [Kim98b, Kop97].

On the other hand, the trend is that the scale of newly emerging HRT applications is continuously growing and as a result the use of minimal-service OSs is becoming less and less tolerable. A problem here is that currently available large-service OSs (e.g., Windows NT or OSs providing at least 1/3 of the services offered by Windows NT) [Ric97, Tim97] have not shown easily analyzable timing behavior and thus have not been reliable for use in supporting sizable HRT applications. Here again, there have been some attempts to use special types of fault tolerance approaches to make up for the weaknesses of OSs [Kim98a, Kim98b, Kop97, Ran95]. For example, if a set of program components runs on a primary-shadow pair of nodes, then there is reasonable chance that the weaknesses of the OS will not show up in both nodes during a short interval [Kim98a].

Therefore, use of fault tolerance mechanisms to make up for the weaknesses of not only hardware but also OSs has become an increasingly popular practice in the field of HRT system engineering.

2.2 Guaranteed service time vs client's deadline

On the other hand, from the viewpoint of realizing systematic 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 completion time*, also called *guaranteed service time*, for every service method that it provides. If a program component provides multiple service methods, it is associated with multiple guaranteed service times. If every program component contributing to the computation subject to a hard deadline has a guaranteed service time associated with it, then meeting the hard deadline becomes the trivial problem of checking whether the sum of the guaranteed service times of the contributing components is indeed less than the hard deadline. The real problem then is to ensure that every guaranteed service time associated with every program component is credible. In a sense, each guaranteed service time 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 guaranteed service time, then a number of other program segments may also be treated as failed components. First of all, the client (program component) to which the failed component has been trying to provide a service is treated as having failed. To be more precise, an exception signal indicating the failure of the server component is sent by the execution engine(s) to the client component. If the client is not *fully dependent* upon the server component, i.e., if the client was designed to handle such an exception signal, then the client can continue its operation. Otherwise, the client component is treated as a failed component.

In addition, those program components which issued previous service requests or will issue future service requests to the failed component, may be treated as failed components. This is mainly because the state of that HRT program component which has failed to meet the guaranteed service time, is highly likely to remain inconsistent with the states of those past, current, and future client components. The inconsistency problem is highly likely to be there even if the failed program component tries to either cleanse itself of any remaining effects of the failed service execution or complete the service execution after the guaranteed completion time. There are some special cases where the inconsistency can be removed, but such case occurrences are a small fraction of all occurrences of guaranteed service time violations.

Therefore, the conservative system operation under which all past and future clients of a failed program component are treated as failed components, is a justified operation in many applications, especially considering that it enables systematic modular construction of HRT distributed computing applications. In environments where the probability of a HRT program component failing to meet a guaranteed service time is non-negligible, the short life expectancy of an application may be a concern to the system designer. Then such a designer must consider embedding effective RT fault tolerance capabilities into HRT program components, thereby reducing the probability of missing guaranteed service times. Any attempt to replace a guaranteed service time 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 are discussed in Section 2.4.

Therefore, HRT program component possessing guaranteed service time attributes is a key to cost-effective systematic construction of HRT distributed computing applications. 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 guaranteed service time 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. In the rest of the paper, this construction approach is assumed.

Figure 1 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. This client's deadline is thus set without consideration of the speed of the server. During the design of the client object, the designer searches for a server object with a guaranteed service time acceptable to him/her.

Actually the designer must also consider the time to be consumed by the communication infrastructure in judging the acceptability of the guaranteed service time of a candidate server object. In general, the following relationship must be maintained:

Time consumed by communication infrastructure + Guaranteed service time

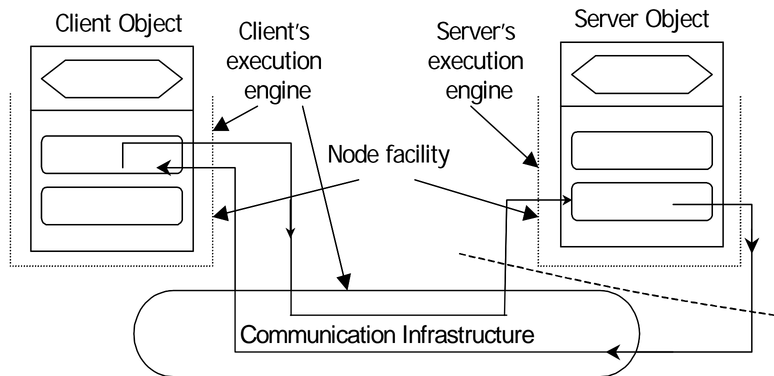


Figure 2. The client's and server's execution engines

< *Maximum transmission times imposed on communication infrastructure + Guaranteed service time*

< *Deadline for result arrival – Call initiation instant*

where both the deadline imposed by the client for result arrival and the initiation instant of the client's remote service call are expressed in terms of absolute real time, e.g., 10am.

In some environments, the maximum transmission times imposed on the communication infrastructure may be violated at a non-trivial frequency but then the system designer must understand the limit on the reliability of the application systems that can be attained in such environments.

2.3 Client's execution engine

There are three sources from which a fault may arise to cause a client's deadline to be violated. They are

(s1) the client object's resources which are basically node facility (hardware + OS),
(s2) the communication infrastructure, and
(s3) the server object's resources which include not only node facility but also the object code.

The server is responsible to finish a service within the guaranteed service time, while the client is responsible for checking if the result comes back within the client's deadline. Therefore, the client object is responsible for checking the results of the actions by all the resources involved, i.e., (s1), (s2) and (s3), whereas the server object is responsible for checking the results of the actions of (s3) only. This means that *the*

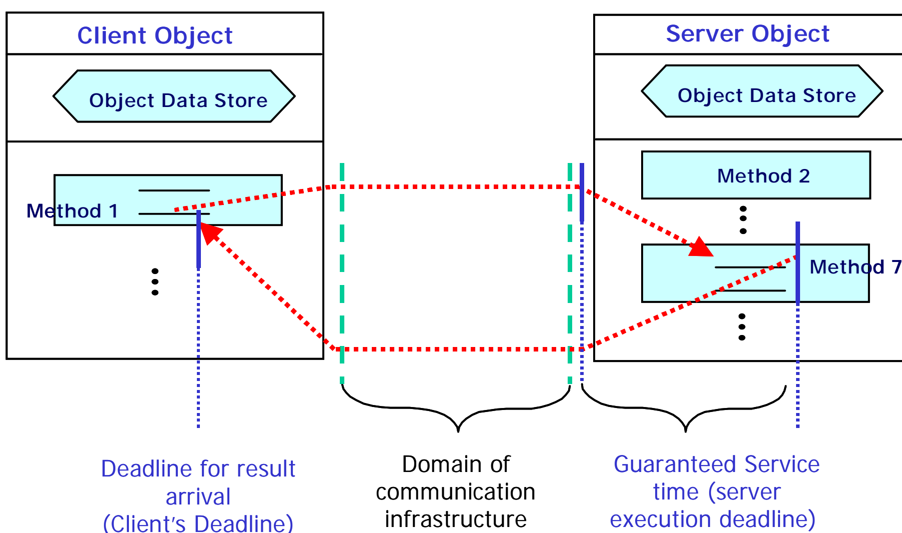


Figure 1. Client's deadline vs Server's guaranteed service time

client's execution engine can be viewed as consisting of not just the client's node facility but also the communication infrastructure, as depicted in Figure 2.

2.4 Augmentations of HRT components with statistical performance indicators

More often than not, HRT program components implemented with currently available tools and execution engines, are associated with guaranteed service times of rather poor qualities, i.e., large guaranteed service times, but exhibit much better service times during most of their service executions. This is due to many factors, including fluctuations in the service times of an OS in supporting application program components, etc.

The communication infrastructure in many application environments has the same characteristics, i.e., guaranteed service time of poor quality and typical service times of much better quality.

Therefore, the designer of a client program component can often run into a situation where available server components do not provide guaranteed service times of acceptable qualities but their typical service times are far better than the client's requirements. A pragmatic compromise in such a situation is to augment each server component with a *statistical performance indicator*. For example, a server component can be put through a number, say 1000, of test-runs while the service time is measured in each test-run. Then the statistical distribution of the service time can be attached to the server program component as a statistical performance indicator of the component.

The designer of a client component may use candidate server components whose statistical performance indicators are good even if the guaranteed service times of the candidate components are not acceptable. This means that the client component must be designed to impose on both the server component and the client's execution engine a deadline for result arrival (DRA) which is smaller than maximum transmission times (imposed on the communication infrastructure) + guaranteed service time (of the server component). In other words, the requirement discussed in Section 2.2 on maintenance of the relationship,

> *(Maximum transmission times + Guaranteed service time) < (Deadline for result arrival – Call initiation instant)",*

is relaxed. Moreover, the client component must be equipped with an alternate logic which will be invoked to accomplish the application objective of the component in time, in case a DRA violation results from the call for a service from the server component. This way the client component can satisfy its own guaranteed service time. Of course, the client component can also offer a statistical performance indicator.

The need to provide an alternate logic is a price paid for relying on the statistical performance indicators of server components, rather than relying on their guaranteed service times.

3. An overview of a general approach to structuring real-time distributed systems: the TMO structuring scheme

In order to deal with the issues in implementation of the deadline mechanisms discussed in the preceding section in concrete terms, the *time-triggered message-triggered object* (TMO) structuring of HRT components will be considered in the rest of this paper. The TMO scheme was established in early 1990's [Kim94, Kim97, Kim99a, Kim99b] with a concrete syntactic structure and execution semantics for economical reliable design and implementation of RT systems. The TMO structuring 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. The basic TMO structure is depicted in Figure 1.

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

(a) Distributed computing component:

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

(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 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 if no SpM that accesses the same object data store segments (ODSSs) to be accessed by this SvM has an execution time-window that will overlap with the execution time-window of this SvM. However, the

BCC does not stand in the way of either concurrent SpM executions or concurrent SvM executions.

(d) *Guaranteed completion time and deadline:*

As in other RT object models, the TMO incorporates deadlines and it does in the most general form. Basically, for output actions and method completions of a TMO, the designer guarantees and advertises execution time-windows bounded by 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" }

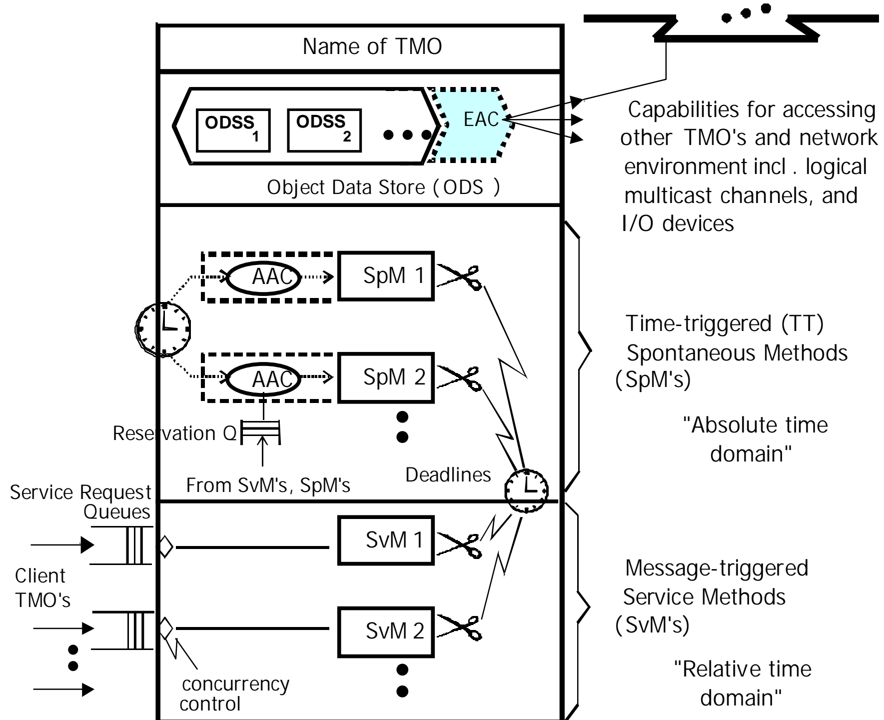


Figure 3. The basic structure of TMO (Adapted from [Kim97]

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. TMOs interact via calls by client objects for service methods in server objects. The caller may be an SpM or an SvM in the client object. To maximize concurrency in the execution of client and server methods, client methods are allowed to make non-blocking (sometimes called asynchronous) types of service requests to SvMs.

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* 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* (hardware plus OS) 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.

A cost-effective way to support TMO-structured distributed RT programming is to build a TMO execution engine as middleware running on well established commercial software/hardware platforms. An efficient middleware model, named TMO Support Middleware (TMOSM), was developed and its implementations on the Windows NT platforms have been operational along with several non-trivial applications structured as TMO Networks [Kim99b].

4. An implementation model for deadline handling in TMO networks

4.1 Multi-transaction structuring of TMO methods

When a violation of a TMO method completion deadline occurs, it could have been caused by any combination of possible sources, e.g. hardware, OS, or

design fault lurking in the TMO. At that point, the ODS of the TMO is in general in a contaminated state.

If the failed method execution is an SvM execution, say SvM2 of TMO2, provisions should be made for letting the client, say SpM1 execution of TMO1, learn of this SvM2 failure. If the client has been prepared to deal with this type of SvM failure, then the business can continue on the client side. Even in such a case, one remaining question is what should be done on the failed TMO side (TMO2). The simplest approach is to discard the TMO (TMO2) altogether and thus any other method executions in progress within that TMO will be discarded, i.e., treated as having failed. This may then involve letting other clients learn of the failures of some of those method executions which are SvM executions.

If the client, TMO1.SpM1, can respond to the failure of the TMO2.SvM2 execution by choosing an alternate course of actions, e.g., requesting a similar service from another TMO, say TMO3, then a less costly action by the failed server TMO (TMO2) is more attractive. That is, TMO1.SpM1 can notify TMO2 that the former is canceling the last request that led to the failed execution of TMO2.SvM2. Meanwhile, TMO2 can attempt to remove any remaining effect of the failed SvM2 execution while waiting for a cancellation of the previous request from TMO1.SpM1. If both are done, i.e., the effect created on the ODS of TMO2 by TMO2.SvM2 is undone and TMO1.SpM1 cancels the request (after learning the failure of the TMO2.SvM2 execution), then there is no reason why TMO2 cannot continue. Therefore, it is worth making provision for a rollback of a TMO to remove the effect of a failed SvM execution.

The rollback capability maintained during a method execution is useful in other respects as well. If a fault is detected by the hardware, the OS, or an application-dependent detection routine before the method completion deadline is reached, then the method execution can roll back to the beginning (or to a checkpoint) and restart.

However, it is practical to exercise the above rollback option only when there is no possibility of the rollback being propagated to other SvM executions in any TMOs. In other words, if the failed TMO2.SvM2 execution passed some information to other SvM executions in different TMOs via method parameters or to other SvM executions within the same TMO (TMO2) via ODS, then rollback of TMO2.SvM2 will dictate rollbacks of those other SvM executions, and thus it is not practical to exercise the rollback option.

In order to limit such information propagation somewhat, we consider it worthwhile adopting the following rule:

> (R1) Prohibit the passage of information from a method execution to other method executions via ODS before the completion of the former method execution.

This means that the ODSSs updated by a TMO method execution must be locked for exclusive access by the method execution until the end of the method execution.

Once the rule R1 is adopted, the only possible way to pass information from a method execution to other SvM executions is to pass via parameters along with SvM calls. In general, an SvM execution (and also an SpM execution for that matter) cannot cancel an SvM call that it has made. This reason was given above, that is, it will incur the needs for chains of rollbacks. The only exceptional case is where the called SvM, say TMO4.SvM7, execution fails and the calling SvM execution, TMO2.SvM2, is designed to invoke an alternate course of actions upon failure of the called SvM (TMO4.SvM7) execution. Therefore, even if an SvM execution (say TMO2.SvM2) runs into a fault and thus its rollback is the only thing worth trying, the call to another SvM (say TMO4.SvM7) that it made earlier cannot be cancelled. We consider it worthwhile adopting the following:

> (R2) If a method execution passed some information to other SvM executions via method parameters before the execution runs into a fault, then the rollback of the method execution should not reach a point prior to the last point of calling an SvM.

This means that the state of the method execution must be saved, i.e., a checkpoint must be established, before every call for an SvM.

Suppose TMO1.SpM1 calls for a service from TMO2.SvM2 which in turn calls for a service form TMO4.SvM7. After receiving a service from TMO4.SvM7, the TMO2.SvM2 execution fails to meet its guaranteed completion time for some temporary faults of the execution engine. Then TMO1.SpM1 cancels its last request to TMO2.SvM2 and the TMO2 execution engine tries to remove any remaining effect of the failed TMO2.SvM2 execution. However, this rollback of the TMO2.SvM2 execution cannot be done since the call issued for an execution of TMO4.SvM7 cannot be cancelled. Therefore, in a case like this, the only thing that the designer of TMO2.SvM2 could have done is to design RT fault tolerance capabilities into the TMO2 execution engine, e.g., the *primary-shadow TMO replication* (PSTR) scheme [Kim98a], so that a TMO2.SvM2 execution may meet the guaranteed completion time even if some temporary faults of the execution engine arise.

Adoption of both R1 and R2 implies the execution of each TMO method as a set of chained *transactions* depicted in Figure 4. The method-segment between two external outputs (e.g., SvM calls) can be viewed as a transaction. By an external output action, the SpM or SvM execution *commits* its current transaction. Before such a commitment, the method execution may involve a check on the reasonableness of the computation state and if it is found to be unacceptable, it can abort its current

transaction, roll back and try again until it successfully reaches an acceptable computation state and commits. By doing this, the fault can be mostly isolated and prevented from being propagated across TMOs or TMO methods.

Immediately before such a commitment, the acceptable computation state must be copied and saved, i.e., a *checkpoint* must be established as shown in Figure 4. This is because if a fault occurs after the commit point, the method execution cannot roll back to a point preceding the commit point. So, each commit point becomes the latest *recovery point* to which the method execution can roll back. In general, establishing a checkpoint involves creating and saving a copy of every ODSS locked by the current TMO method execution, a copy of the local variables, and a copy of the state of a thread assigned to the current method execution. It also involves recording the commit action to be executed immediately. The TMO execution engine manages the memory space for holding these copies.

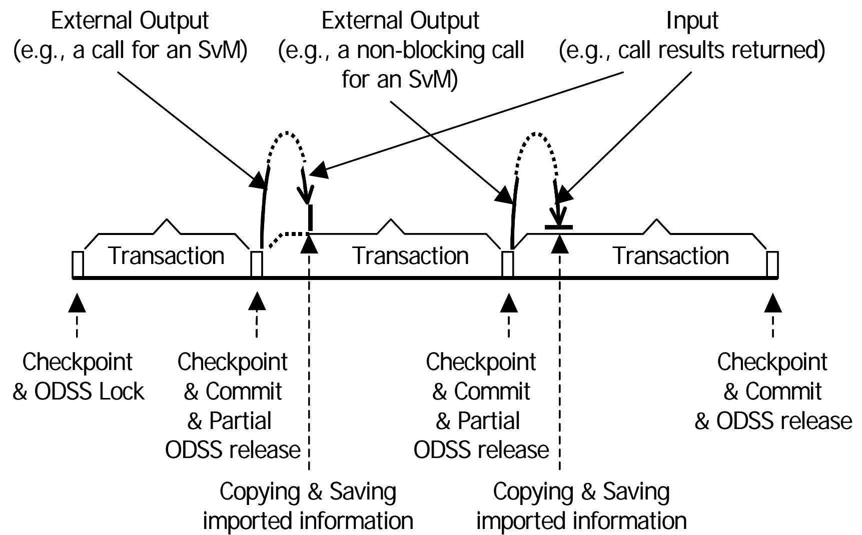


Figure 4. The multi-transaction model

In addition, each input data acquired during a transaction is copied and saved by the TMO execution engine so that the saved copy may be replayed during a retry of the transaction following a rollback.

4.2 Deadline handling in a pure server

An RT object such as TMO can be (1) a *pure server* object, (2) a *pure client* object, or (3) a *client+server* object. A pure server object is a server object which is not a client for any other server objects. A pure client object is a client object which is not a server for any other client objects. A client+server object is an object which plays both the client role and the server role.

Normally, a pure server will complete a requested SvM execution and send the requested return results to the client before the associated guaranteed completion time. Let us now consider the cases where the SvM execution fails to complete by the guaranteed completion time. The server's execution engine tries to do the following:

(1) Send a failure notice to the client: This notice should help the client avoid unnecessary waste of time when the deadline for result arrival imposed by the client is still far away.

(2) Abort the failed SvM execution, i.e., remove any remaining effect of the failed SvM execution: This will be successful only if the latest commit point is the one set at the beginning of the failed SvM execution. This condition is met in the case of a pure server. If the abortion is successful and a cancellation notice is received from the client, then this pure server TMO can continue its operation. Otherwise, the pure server TMO should crash (i.e., shut down by the TMO execution engine) except when it is in a debugging mode.

4.3 Deadline handling in a pure client

An exception signal may be sent to a pure client in two ways:

(1) Type-I: The client's execution engine detects a violation of the deadline imposed by the client for result arrival, and the execution engine generates an exception signal.

(2) Type-II: The server's execution engine sends to the client a notice which is an exception signal notifying that the server failed to complete the requested service by the guaranteed completion time.

Designing the pure client's responses to these exception signals may involve selection of one of the following approaches:

(1) No application-level programming and full dependence on automated exception handling by the execution engine;

(2) Programming of an alternative action per call: An exception signal is embedded in a result value of the SvM call and then the statements for checking the exception signal and taking an exception handling action sequence are produced by the application programmer;

(3) Programming of an exception handler per method: An exception handler function for a method of the client is provided by the application programmer and it is designed to be invoked whenever the execution engine detects an exception signal at any point during the execution of the client method.

4.3.1 Full dependence on automated exception handling by the execution engine

(1) Type-I exception signal

In this case, the client's execution engine tries to abort the current execution of the client method since it has become hopeless for the client to proceed correctly. The abortion may or may not succeed, depending upon whether the SvM call that led to this exception signal has been the only SvM call made by the client method or not. If it is not the only SvM call made, it is too late and the abortion cannot be done unless a *compensation method* is available. A compensation method is designed together with a TMO method to make application-specific output actions to nullify the effects of the earlier output actions of the TMO method (taken before the need for abortion arose). It is not always possible to design a compensation method. Also, the success of this client's abortion depends on whether the failed server can abort the failed SvM execution or not.

Even if the abortion is successful and the pure client TMO may continue its operation, the client TMO may be a failed TMO since one method execution was skipped. Therefore, shutting the TMO down is the only option in many application situations.

(2) Type-II exception signal

The client's execution engine checks if the deadline for result arrival is over or not. If it is over, then the same actions taken in the above case (1) are taken. If not, then the execution engine notifies the server TMO that the previous SvM call is cancelled but it is reissued.

4.3.2 Programming of an alternative action per call

The client's execution engine embeds the exception signal in a result value of the SvM call. A basic SvM call is programmed in a form, "if SvM_call (TMO_Name, SvM_Name, …, deadline) ≠ 0, then …AA…" where "0" indicates normal successful completion and AA represents an alternative action. The first step in the alternative action is to order the execution engine to notify the server that the last SvM call which led to the failed execution is cancelled. An alternative action to be taken by the client method is designed to perform meaningfully regardless of whether the SvM call issued by the client method reached the execution engine for the called SvM correctly or not.

For other types of SvM calls, e.g., non-blocking calls, programming of exception checks and alternative actions is done in forms which are relatively minor variations of the above form. This programming approach is better justified in situations where server TMOs with statistical performance indicators are used as discussed in Section 2.4.

4.3.3 Programming of an exception handler per method

Whenever the client's execution engine detects an exception signal, it stops the client method and invokes the exception handler function designed as a companion to the client method. A sound generic type of an exception handler is one that realizes the semantics of the recovery block [Ran95]. Such an exception handler consists of a step for aborting the current execution of the client method and a step for executing an alternative version of the client method. The alternative version is aimed for producing the same result that the primary version is aimed for. Again this kind of approaches which place burdens on the application programmer are better justified in situations where server TMOs with statistical performance indicators are used.

4.5 Deadline handling in a client+server TMO

Here an SvM may involve calls for other SvMs. Basically, the deadline handling approaches applicable to such an SvM are a combination of the deadline handling approaches applicable to a pure server and those applicable to a pure client. Abortion of such an SvM is less likely to succeed than the abortion of a pure server is.

5. Conclusion

Deadline handling is a fundamental part of real-time computing. This paper has proposed a general broadly applicable framework for systematic deadline handling in RT DCSs. A prototype implementation of the basic middleware support for the proposed deadline handling scheme has been completed recently. However, the cases where advanced RT fault tolerance techniques such as those for active replication of TMO method executions [Ran95] are used, have not yet been dealt with and remain a subject for future study. Systematic deadline handling is an area where much more experimental research is needed. Especially, design experiments involving the use of HRT objects with statistical performance indicators and the design of alternative actions, are considered to be highly meaningful subjects for future research.

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

[J-C99] J Consortium, "Draft International J Consortium Specification", available from http://www.j-consortium.org/, Nov. 1999.

[Kim94] 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.

[Kim97] 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., "An Integration of the Primary-Shadow TMO Replication (PSTR) Scheme with a Supervisor-based Network Surveillance Scheme and its Recovery Time Bound Analysis", *Proc. SRDS '98 (IEEE CS 17th Symp. on Reliable Distributed Systems)*, West Lafayette, IN, Oct. 1998, pp.168-176.

[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., "Real-Time Object-Oriented Distributed Software Engineering and the TMO Scheme", *Int'l Jour. of Software Engineering & Knowledge Engineering*, Vol. No.2, April 1999, pp.251-276.

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

[Kop97] Kopetz, H., '*Real-Time Systems: Design Principles for Distributed Embedded Applications*', Kluwer Academic Publishers, ISBN: 0-7923-9894-7, Boston, 1997

[OMG99] Object Management Group, '*Realtime CORBA Joint Revised Submission*,' OMG Document orbos/99-02-12 ed., March 1999.

[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.

[Ric97] Richter, J., '*Advanced Windows*', 3rd Edition, *Microsoft Press*, 1997.

[RJG99] Real Time Specification for Java Experts Group, "JSR-000001 Real-time Extension Specification", available from www.rtj.org.

[Sel99] Selic B., "Turning clockwise: using UML in the real-time domain", *CACM*, Vol. 42, No. 10, Oct. 1999, pp. 46-54.

[Son94] Son, Sang H., '*Advances in Real-Time Systems*', Prentice Hall, 1994.

[Tim97] Timmerman. M., Monfret. J.C., "Windows NT as Real-Time OS?", *Real-Time Magazine*, Issue 97/2.

Proceedings

Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2000)

March 15-17, 2000

Newport Beach, California, USA

Sponsored by

IEEE Computer Society

In Cooperation with

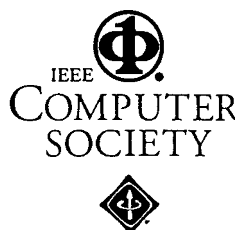
IFIP WG 10.

OMG

IBM

Financial Contributions from

IBM



Los Alamitos, California

Washington • Brussels • Tokyo