

Realization of Autonomous Decentralized Computing with the RTO.k Object Structuring Scheme and the HU-DF Inter-Process-Group Communication Scheme

K. H. (Kane) Kim
University of California, Irvine, U.S.A.
Kane@Ece.Uci.Edu

Kinji Mori
Hitachi SDL, Japan

and

Hiroaki Nakanishi
Hitachi Omika Works, Japan

Abstract: The autonomy of various subsystems in complex real-time distributed computer systems (DCS's) is a highly desired attribute. Highly autonomous subsystems enable concurrent and largely independent testing/verification and maintenance of the subsystems. In recent years, we formulated two system structuring techniques which could be used as fundamental tools for enhancing the subsystem autonomy: (1) the RTO.k object structuring scheme and (2) the HU-DF inter-process-group communication scheme. Three approaches of using the above two tools for enhancement of the three different basic types of autonomy in an object, (1) relocation autonomy, (2) service timing autonomy, and (3) data acceptance autonomy, are presented.

Index Terms: autonomy, distributed computing, real-time object, data field, object structure, inter-process communication.

1. Introduction

Enhancing the autonomy of various subsystems (or processing components) in distributed computer systems (DCS's) returns the benefit of the increased cost-effectiveness in system *validation* and *maintenance*. Highly autonomous subsystems enable concurrent and largely independent testing/verification and maintenance of the subsystems. These are issues of great importance in developing and operating large-scale complex real-time DCS's. A challenge in designing DCS's is then to maximize the degree of autonomy of the subsystems while achieving the objective of efficient cooperative computing.

In recent years, we formulated two system structuring techniques which could be used as fundamental tools for enhancing the subsystem autonomy: (1) the RTO.k object structuring scheme [Kim93, Kim94b] and (2) the HU-DF inter-process-group communication scheme.

The *RTO.k object model*, also called the time-triggered real-time object (TT_RTO) model, is a result of the attempt by the first co-author and Hermann Kopetz at the Technical University of Vienna to find a proper extension of the basic object model which is highly cost-effective in development of *hard-real-time* application

systems. Based on the initial framework formulated by Kopetz and Kim in late 1980's [Kop90], we formulated a concrete syntactic structure and execution semantics in recent years [Kim93, Kim94b, Kim94c]. A major attraction of this model is in that it is based on the following futuristic paradigms of real-time computing [Kim94d]:

- (1) *Real-time computing* \supset *Non-real-time computing*: Future real-time computing must be realized in the form of a generalization of the non-real-time computing, rather than in a form looking like an esoteric specialization.
- (2) *Design-time guarantee of timely service capabilities of subsystems/objects*: To meet the demands of the general public on the assured reliability of future real-time computer systems (RTCS's) in safety-critical applications, there is no adequate way but to require the system engineer to produce design-time guarantees for timely service capabilities of various subsystems (which will take the form of objects in object-oriented system designs).

The RTO.k object model is effective not only in the multiple-level abstraction of real-time (computer) control systems under design but also in the accurate representation and simulation of the application environments. A specification and implementation experiment that involved an application of the RTO.k structuring scheme to the development of a defense system together with an application environment simulator was conducted [Kim94b]. This experiment reinforced our belief that the RTO.k model had the necessary representational power and also offered an efficient and rigorous way to develop complex real-time systems.

The *HU-DF (HU data field) scheme* for inter-process-group communication is an extension of the original *data field* scheme developed by the second co-author and other researchers in Hitachi, Ltd. [Mor86, Mor93]. The essence of the data field scheme is to facilitate dynamic creation of logical multicast channels and dynamic connection of processes to the logical channels in such a way that the idiosyncracies of the physical communication networks are transparent to the process designer. If the physical communication facility has the broadcast capability, then a logical multicast

channel is facilitated by making all processing nodes using the channel broadcast (through the physical communication facility) messages with the headers containing the ID of the channel called the content code. The processing nodes connected to the logical channel can see all the messages coming through the physical broadcast facility but will pay attention only to those messages containing relevant content codes. This means that when processes are designed to communicate via the logical multicast channels only, processes can be dynamically relocated without impacting other cooperating processes. The HU-DF scheme differs from the original data field scheme in that the former allows dynamic flexible connection of processes to the logical channels and supports not only conventional event messages but also state messages which are based on the distributed replicated memory semantics.

This paper starts with a brief overview of both the RTO.k object structuring scheme and the HU-DF scheme in Sections 2 and 3, respectively, and then presents three approaches of fundamental nature for enhancement of the *object autonomy* in Section 4. The first approach is to use the HU-DF scheme (event message channels) as a means for interconnecting RTO.k objects. The major direct benefit is the *transparency of object locations* which in turn leads to the enhanced autonomy of the objects.

The second approach is a special form of RTO.k object structuring which is called the master spontaneous method structuring and enables objects to choose their action times of convenience without being dictated by their clients. In this sense, RTO.k objects with spontaneous master methods are highly autonomous and enable relatively easy guarantee of timely services of total DCS's to application environments.

The third approach for enhancement of the autonomy of an RTO.k object is the loose connection between producer RTO.k objects and consumer objects realized by using only the state message channels provided under the HU-DF scheme for inter-object message communication. This loose connection implies elimination of mutual acknowledgment-dependency between producers and consumers at the message delivery protocol level. This kind of connection is particularly beneficial when one communication party (producer or consumer) is much less reliable than the other communication party.

The paper concludes in Section 5.

2. An overview of the RTO.k object structuring scheme

The RTO.k object model formulated in recent years [Kim93, Kim94b, Kim94c] was specifically devised with the following goals directly related to the futuristic paradigms of real-time computing mentioned in Section 1.

(a) Uniform structuring of both RTCS's and their

application environment simulators;

(b) Facilitating design-time guarantee of timely service capabilities of objects.

2.1 Basic structure

The basic structure of an RTO.k object is depicted in Figure 1. It is an extension of the conventional object model(s) in four essential ways:

(a) For some methods of an RTO.k object, a real-time clock serves as the mechanism for triggering the method executions as the clock reaches some values specified at design time and such methods are called time-triggered (TT-) methods, also called the spontaneous methods (SpM's), and *clearly* separated from the conventional service methods (SvM's) triggered by messages from clients;

(b) A concurrency constraint which prevents conflicts between TT-methods and message-triggered methods is incorporated. Basically, *activation of a service method triggered by a message from an external client is allowed only when potentially conflicting TT-method executions are not in place*. To be exact, when a message-triggered service method is not free of conflict in accessing the same portion of the object data space (ODS) with a TT-method, execution of the former (message-triggered) method must not be allowed in a time zone earmarked for a TT-execution of the latter (spontaneous) method. This restriction is called the basic concurrency constraint (BCC). Therefore, TT-methods are given higher priorities for execution over the message-triggered methods. Note that this BCC does not impose any restriction on concurrent execution of TT-methods or concurrent execution of message-triggered methods;

(c) For each execution of a method of an RTO.k object, a deadline is imposed;

(d) Real-time data contained in an RTO.k object become invalid after the interval called the maximum validity duration passes.

Extensions (a) (the clear-cut separation of SpM's from SvM's) and (b) (the execution rule called the basic concurrency constraint) represent the two most distinguishing characteristics of this model relative to other proposed real-time object models [Att91, Bih89, Ish90, Ish92, Shi91, Tak92]. The two types of methods in an RTO.k object are different not only in the way their executions are triggered but also in that

“actions to be taken at real times *which can be determined at the design time* can appear only in SpM's”.

Therefore, actions of the type “at constant-clock-value do S” or the type “sleep-until constant-clock-value” can appear only in SpM's.

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. The AAC may be specified in the following form.

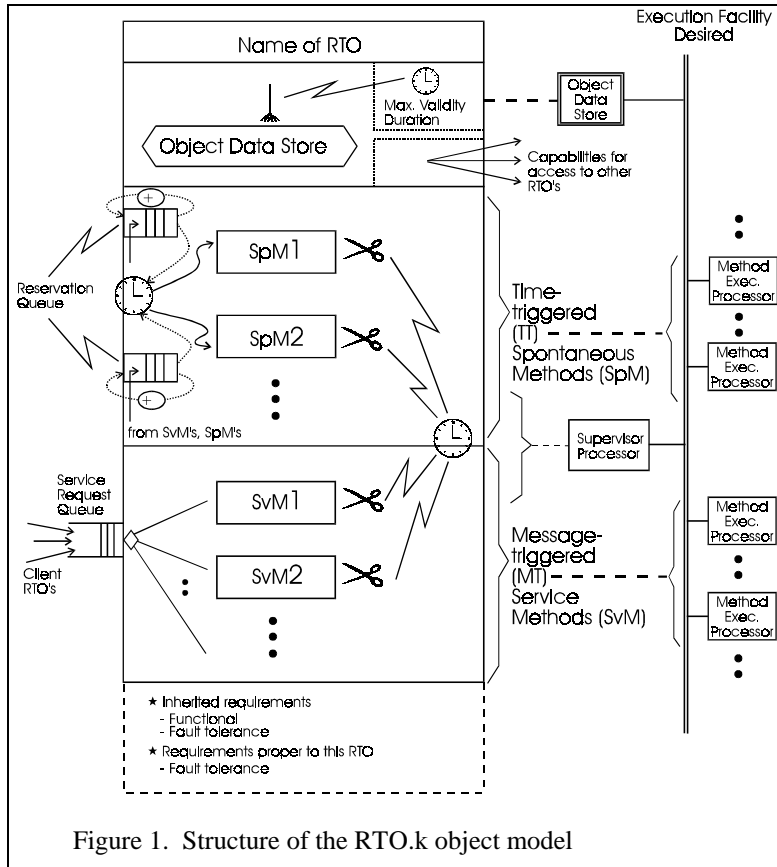


Figure 1. Structure of the RTO.k object model

```

ab      "AAC-begin"
{ [AAC name:]
  for <time-var> = from <activation-time>
                    to <deactivation-time>
                    [every <period>]

  start-during
    (<earliest-start-time>,
     <latest-start-time>)

  finish-by<deadline>
}*
ae      "AAC-end"

```

where the "star" expression x^* or $\{x\}^*$ is a regular expression for the set $\{\text{NULL}, x, xx, xxx, \dots\}$. Note that "for $t = \text{from } 10\text{am to } 10\text{am start-during } (t, t+5\text{min})$ " has the same effect as "start-during (10am, 10:05am)".

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 RTO.k object requests future executions of a specific SpM. The AAC specifying candidate triggering times rather than actual triggering times starts with a declaration

"if-demanded". Therefore, there are two different modes of determining triggering times for SpM's:

- fully determined during the system design, in which case the SpM is said to be statically scheduled, and
- determined during the run time when an SvM requests executions of the SpM and designates a subset of the candidate triggering times prepared during the design time as actual triggering times, in which case the SpM is said to be partially dynamically scheduled.

On the other hand, actions to be taken at real times which cannot be determined at the design time may appear in SvM's, even if the actions may have to be executed periodically. Therefore, an SvM may include a statement of the type "for X seconds every Y seconds do S". However, the start time of this statement execution is not known until the SvM is called by a client.

An underlying design philosophy of the RTO.k object model is that an RTCS will always take the form of a network of RTO.k objects. RTO.k objects 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. In order to facilitate highly concurrent operations of client and server objects, non-blocking (sometimes called asynchronous) types of calls (i.e., service requests) can be made to SvM's.

The following types of concurrency can be exploited in executions of object methods in an RTO.k object:

- Concurrency among SpM executions;
- Concurrency among SvM executions;
- Concurrency between SpM executions and SvM executions.

Concurrency among the SpM's is specified in an implicit but natural manner, e.g., two SpM's designed to be triggered at 10 am. The approach adopted for exploiting concurrency of type-b and type-c is to explicitly declare the portion of the object data space (ODS) used by each method and allow concurrent execution of methods whenever there is no data conflict.

2.2 Design-time guarantee of timely service capabilities of RTO.k objects

The designer of each RTO.k object provides a guarantee of timely service capabilities of the object by indicating 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) advertised to the designers of potential client objects. An output action here may be

- an updating of a portion of the ODS,
- sending a message to either another RTO.k object (which may or may not be the client) or a device shared

by multiple objects, or

(c) placing a reservation into the reservation queue for a certain SpM (where the reservation queue holds future execution triggering schedules for the SpM determined up to the present).

The specification of each SvM which is provided to the designers of potential client RTO.k objects must contain at least the following:

- (a) an *input specification* that consists of
 - (a1) the types of input parameters that the server object can accept and
 - (a2) the maximum rate at which the server object can receive service requests from client objects;
- (b) an *output specification* that indicates the *deadline* (not the exact output time) and the *nature of the output value* for every output produced by the SvM.

Before determining the deadline specification, the server object designer must convince himself/herself that with the object execution engine (hardware plus operating system) available, the server object can be implemented to always execute the SvM such that the output action is performed within the deadline.

On the other hand, a client RTO.k object imposes a deadline on the server RTO.k object for creation of all the intended computational effects (i.e., all intended output actions). The deadline imposed by a client must not be earlier than the deadline adopted and advertised by the designer of the server object for completion of the corresponding SvM.

The specifications of the SpM's which may be executed on requests from the SvM must also be provided to the designers of the client objects which may call the SvM. There is no input specification in an SpM specification but the output specification for an SpM indicates, for every output expected from the execution of an SpM, the exact time *at* or *by* which it will be produced and the nature of every value carried in the output action.

The basic concurrency constraint (BCC) was incorporated into the RTO.k object model to ease the design-time guarantee of timely service capabilities. At least it makes it very easy to analyze the execution time behavior of SpM's. Executions of SpM's are not disturbed by SvM executions and triggering times of SpM's are fixed at the design time. For example, if a statement of the type "at 10am do S" appears in an SpM, its reliable execution can be easily assured.

2.3 Execution engines for RTO.k objects

Figure 1 also shows an idealistic execution engine model in which each method of an RTO.k object is mapped onto a dedicated processor with its local memory. A more economic execution engine will use each processor in execution of multiple RTO.k objects, let alone in execution of multiple methods in one RTO.k object. Such an execution engine can share some parts with the existing real-time operating system kernels but must contain several major unique components. Recently

we designed such an execution engine model called the DREAM (Distributed Real-Time Ever Available Microcomputing) engine and implemented the first prototype, DREAM kernel, on the PC LAN equipped with Intel 80486 processors, DOS-BIOS device drivers, the Packet Ethernet driver, and an inter-process multicast communication manager called the *HU data field subsystem*. Services of the DREAM kernel including process management services can be obtained from within a C++ program (representing an implementation of an RTO.k object) via calls for DREAM library routines. An early version of the prototype DREAM kernel was used in an experimental development of a defense system with its application environment simulator [Kim94b].

2.4 Uniform structuring of control computer systems and application environment simulators

The RTO.k object supports an interesting style of simulation [Kim94b]. The ODS in an RTO.k object which simulates an application environment contains state representations of physical components in the environment, e.g., the airplanes, the ships, and the space (sky + land + sea). Each TT-method, when executed, updates a variable-set in the ODS representing the state of some physical component (i.e., airplane, ship) to reflect the current state of the physical component. Ideally the TT-methods should be *activated continuously* and each of their executions be *completed instantly*. However, the limited power of the simulation engine dictates the activation frequency of any TT-method to be no more than once per every simulator clock tick while allowing each execution to be completed before or by the time of the following activation of the same method. Therefore, TT-methods are the mechanisms for simulating continuous state changes that occur naturally in the environment objects. The natural parallelism that exists among the environment objects is precisely represented by use of multiple TT-methods which may be activated simultaneously. This single RTO.k representation can be expanded into a representation in the form of a network of RTO.k objects, each representing a physical component (e.g., airplane or ship).

3. The HU data field (HU-DF) scheme for inter-process-group communication

The main features of the HU-DF scheme are discussed and the differences between the HU-DF scheme and the original DF scheme developed by the second co-author and others [Mor82, Mor86] are briefly pointed out.

(1) Logical multicast channels and dynamic connection of a process to channels

In order to minimize the impacts of frequently changing hardware configurations on the software, it is useful to design and implement the software such that physical node addresses of each node are used only inside a small kernel module within the node. With such design,

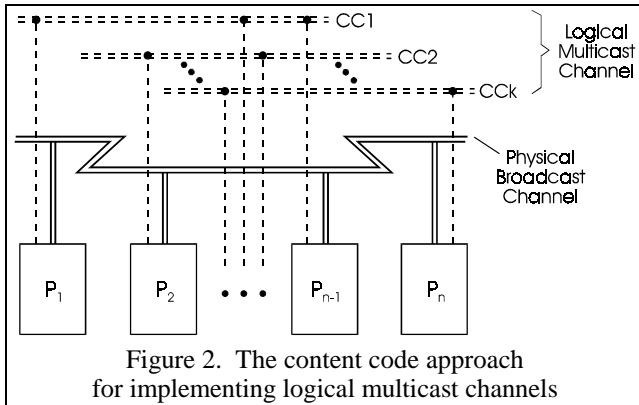


Figure 2. The content code approach for implementing logical multicast channels

interaction among the processes distributed over multiple nodes does not involve use of physical node addresses.

The original DF scheme developed in [Mor82, Mor86] supports establishment of "logical multicast channels" shared among the concurrent distributed processes for their interaction without necessitating the processes to know the identities of other cooperating processes, let alone the identities of the nodes running the processes. The only thing that a group of cooperating processes need to know in advance for a message communication is the name of the logical multicast channel through which the message will be communicated. Each of those processes can simply educate the communication subsystem in the host node about the logical multicast channel to be used.

With popular CSMA bus and token ring architectures, establishing a logical multicast channel to be shared among a group of processes boils down to choosing a logical channel ID number, called *content code* in [Mor86], to be included in the address field of each message broadcasted over a CSMA bus or a token ring. Figure 2 depicts such an arrangement. The processing nodes connected to the logical channel can see all the messages coming through the physical broadcast facility but will pay attention only to those messages containing relevant content codes. This means that when processes are designed to communicate via the logical multicast channels only, processes can be dynamically relocated without impacting other cooperating processes.

Therefore, a process wanting to send a message to other cooperating processes will execute a primitive "Multicast the data D_i over the channel C_x ". Then all other processes set up to share channel C_x will pick up the message.

In the HU-DF scheme, a process can freely connect itself to or disconnect itself from a logical multicast channel, which is somewhat restricted in the original DF scheme. The communication subsystem supports primitives such as "Add content codes $\{CC_i, CC_j, CCK\}$ ", "Delete content codes $\{CC_i, CC_j\}$ ", etc. A process can thus be connected to multiple logical multicast channels, thereby joining multiple communicating groups.

(2) Event messages and state messages

The HU-DF scheme supports not only conventional event messages as the original DF scheme does but also state messages which are based on the distributed replicated memory semantics [Kop89]. The differences between event messages and state messages are depicted in Figure 3. An event message carries information about an event occurrence and every event message should be read by the intended receiver. An event message is not supposed to be overridden by another event message. On the other hand, a state message carries information to be stored in a fixed memory location in each receiver corresponding to the ID of the state message. Therefore, the ID of a state message represents a group of replicated memory units, each capable of holding the information carried in the state message and belonging to a different receiver. The producer of a state message timestamps the message at the message production time. Each receiver will read the content of its state message memory at its convenient time. This means that the producer may update the contents of the state message memory units at a higher frequency than that at which a certain receiver reads the content of its state message memory. A state message is thus typically used to share the periodically observed state information about a dynamic object, e.g., temperature of an oven.

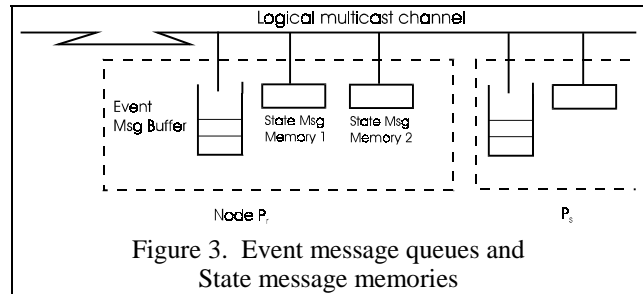


Figure 3. Event message queues and State message memories

In the HU-DF scheme, every logical multicast channel consists of one event message channel and multiple (fixed number) state message channels. More specifically, the message transmitted over a logical multicast channel contains either the sequence number of an event message or the ID of a state message channel in its header. State message channels are particularly beneficial when one communication party (producer or consumer) is much less reliable than the other communication party.

3.3 Blocking and non-blocking read of both event and state messages

The HU-DF scheme supports both *blocking* and *non-blocking read* of event messages. In the case of a non-blocking read of an event message from a logical multicast channel, the reading process picks an event message from its event message buffer allocated for the logical multicast channel (Figure 3) and if the buffer is empty at the time of reading, proceeds to the next step

without waiting for arrival of a new event message. State messages are always read in the non-blocking mode.

4. Types of autonomy of an RTO.k object in a real-time distributed computer system

Three approaches of fundamental nature for enhancement of the object autonomy are discussed in this section.

4.1 Relocation autonomy

In the execution engine model mentioned in Section 2.3, the DREAM engine, every method of an RTO.k object is mapped to a process. Interaction of RTO.k objects is realized only through SvM calls. In order to maximize the relocation autonomy, i.e., autonomy in choosing its residence, of each RTO.k object, the HU-DF scheme is incorporated into the DREAM engine and every SvM is assigned and represented by a dedicated logical multicast channel (i.e., a content code number). The SvM receives service requests via the channel and returns the result parameter to the channel. The event message channel is used for this purpose. To every SvM a unique logical multicast channel is dedicated. With this arrangement, an RTO.k object can be relocated to another site in a network without impacting its client objects.

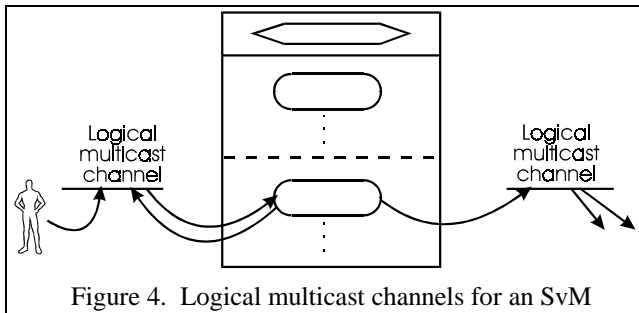


Figure 4. Logical multicast channels for an SvM

In a system where objects are dynamically created and destroyed, information on logical multicast channels representing the SvM's of a newly created object can be announced to all potential client objects by the creator object or the configuration manager responsible for system configuration including the name service and others.

4.2 Service timing autonomy

If an object can freely reject service requests from clients when the object is in an "uncomfortable" situation, e.g., due to an overload condition created, internal faults, etc., then philosophically one can say that the object is autonomous. In a sense, such capability is built into the RTO.k object. As discussed in Section 2.2, an object is advertised with the maximum rate at which the object can receive service requests from client objects, which is one possible form of representing the maximum load that can be handled without violating the timely service guarantee, or with an alternative representation of the maximum load. This was necessary to make timely service guarantees.

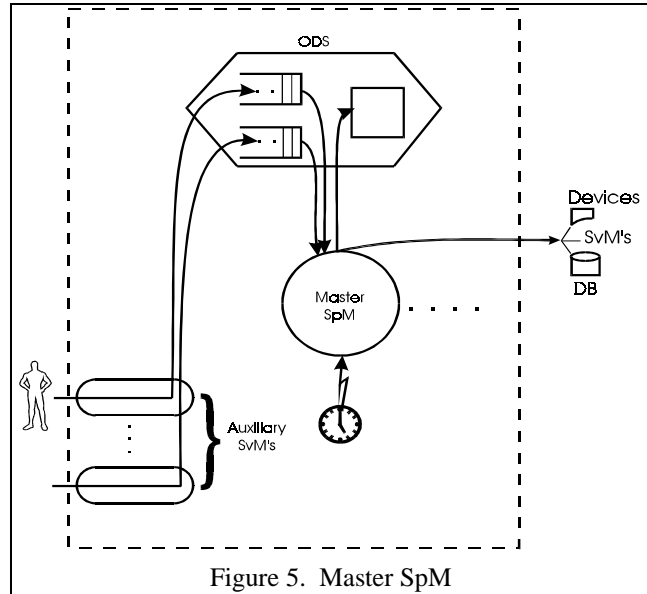


Figure 5. Master SpM

Therefore, when the service request receiver in an RTO.k object receives service requests while the object is in a maximally loaded state, the receiver replies with rejection messages.

More generally, a highly desirable type of object autonomy in a real-time object oriented DCS is the service timing autonomy of an object, i.e., the autonomy in choosing its times for handling requests from clients. This autonomy is valuable in enabling the easy analysis of the timing behavior of the DCS. For example, a typical situation which presents difficulties in analysis of the timing behavior of the DCS is where several long chains of SvM calls (i.e., an SvM call involving another SvM call which in turn involves another SvM call, etc.) are created and intertwined. Basically, the analyst is forced to examine many interference possibilities of numerous objects together to derive a conclusion about the timing behavior of the system.

On the other hand, in the extreme case of an RTO.k object where SvM's are not allowed to call the SvM's of other objects and thus most calls to SvM's of other objects are made by SpM's, the analysis of the timing behavior of the DCS involves multiple rounds of examining two objects in a client-server relationship at a time. This arrangement is called the master spontaneous method structuring and depicted in Figure 5. The SvM's in such an object can play the roles of receptionists which receive service requests and then convert them into work orders to be put in a queue which is examined regularly by a master SpM. Such SvM's are called auxiliary SvM's. The master SpM can reject some work orders when the workload is excessive and inform the clients of such rejection. Note that an RTO.k object can have multiple master SpM's.

The aforementioned approach of allowing a server object to reject some service requests should be

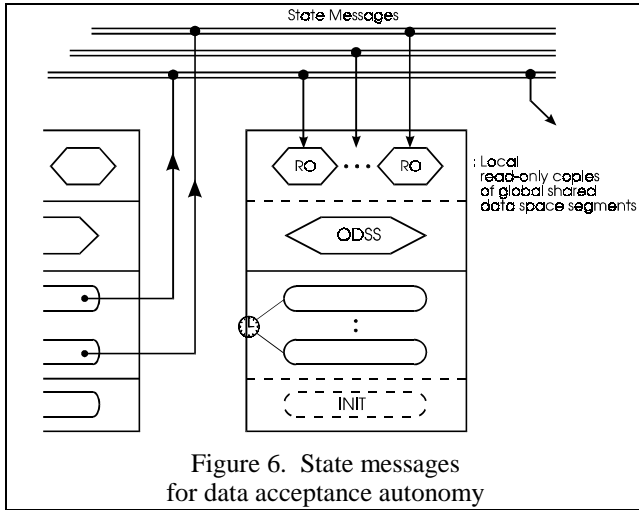


Figure 6. State messages for data acceptance autonomy

accompanied by provision of multiple servers, otherwise, the cooperative distributed computing may collapse.

4.3 Data acceptance autonomy

In order to further loosen the connection between data producer objects and consumer objects, state message channels can be utilized for providing data directly into the ODS's of consumer objects. Figure 6 depicts such a consumer RTO.k object. Special read-only components are included in the ODS and they are connected to state message channels. This means an extension of the RTO.k object model reviewed in Section 2. This facilitates the data acceptance autonomy of an object, i.e., autonomy in choosing its times for accepting real-time input data. Again, if either member of a producer-consumer pair fails, then the partner is not necessarily blocked. The special read-only components can be viewed as read-only local copies of global shared data space segments (or bulletin board postings).

Moreover, with the above arrangement, it is possible to have RTO.k objects without any SvM's but a trivial SvM used for object initialization. SvM calls in normal RTO.k objects are replaced by state messages carrying work orders to be picked by SpM's at their convenient times. The result is then the ultimate in loosening the connection between producer and consumer objects.

5. Conclusion

Three approaches of fundamental nature for enhancement of the object autonomy were presented in this paper. Of the three, the approaches for facilitating the relocation autonomy and the service timing autonomy were tested through an experimental development of a defense system and its application environment simulator. As the research in the RTO.k object structuring of real-time DCS's progresses further to include experimental validation of fault tolerance techniques [Kim94a], the approach for facilitating the data acceptance autonomy

will be tested. Much research and development work remains to be done to adherence to the futuristic paradigms of real-time computing mentioned in Section 1 a relatively painless common practice.

Acknowledgment: The research work reported here was supported in part by US Navy, NSWC Dahlgren Division under Contract No. N60921-92-C-0204, in part by the University of California MICRO Program under Grant No. 93-080, in part by Hitachi, Ltd., and in part by ETRI.

References

- [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.
- [Bih89] Bihari, T., Gopinath, P., and Schwan, K., "Object-Oriented Design of Real-Time Software", *Proc. IEEE CS 10th Real-Time Systems Symp.*, 1989, pp.194-201.
- [Ish90] Ishikawa, Y., Tokuda, H., and Mercer, C. W., "Object-Oriented Real-Time Language Design: Constructs for Timing Constraints", *Proc. ECOOP/OOPSLA '90*, October 1990, pp. 289-298.
- [Ish92] Ishikawa, Y., Tokuda, H., and Mercer, C. W., "An Object-Oriented Real-Time Programming Language", *IEEE Computer*, October 1992, pp. 66-73.
- [Kim93] Kim, K.H. and Bacellar, L.F., "A Real-Time Object Model: A Step toward an Integrated Methodology for Engineering Complex Dependable Systems", *Proc. CSESAW '93 (1993 Complex System Engineering Synthesis and Assessment Technology Workshop)*, US Navy NSWC, July 1993, pp.56-64.
- [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. and Kopetz, H., "A Real-Time Object Model RTO.k and an Experimental Investigation of Its Potentials", *Proc. 1994 IEEE CS Computer Software and Applications Conf. (COMPSAC)*, Nov. 1994, Taipei, pp.392-402.
- [Kim94c] Kim, K.H. et al., "Distinguishing Features and Potential Roles of the RTO.k Object Model", to appear in *Proc. 1994 IEEE CS Workshop on Object-oriented Real-time Dependable Systems (WORDS)*, Oct. 1994, Dana Point, (Proceedings to be published in Mar. 1995, A draft version in the preliminary workshop proceedings).
- [Kim94d] Kim, K.H., "A Utopian View of Future Object-Oriented Real-Time Dependable Computer Systems", (Invited paper) *Proc. 1st Int'l Workshop on Real-Time Computing Systems and Applications*, Seoul, Korea, Dec. 1994, pp. 59-69.
- [Kop89] Kopetz, H., Damm, A., Koza, C., Mulazzani, M., Wolfgang, S., Senft, C., and Zainlinger, R., "Distributed Fault-Tolerant Real-Time Systems: The Mars Approach", *IEEE Micro*, Feb. 1989, pp. 25-39.

[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.

[Mor82] Mori, K. and Ihara, H., "Autonomous Decentralized Loop Network", *Proc. of COMPCON Spring 1982*, pp. 192-195.

[Mor86] Mori, K., et. al., "Autonomous Decentralized Software Structure and Its Application.", *Proc. Fall Joint Computer Conference*, Dallas, Texas, November 1986, pp. 1056-1063.

[Mor93] Mori, K., "Autonomous Decentralized Systems: Concept, Data Field Architecture, and Future Trends", *Proc. IEEE CS Int'l Symp. on Autonomous Decentralized Systems (ISADS 93)*, Mar. 1993, Kawasaki, Japan, pp. 28-34.

[Shi91] Shrivastava, S.K. and Waterworth, A., "Using Objects and Actions to provide Fault Tolerance in Distributed, Real-Time Applications", *Proc. IEEE CS 12th Real-Time Systems Symp.*, 1991, pp.276-285.

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

Proceedings

ISADS 95

**Second International Symposium on
Autonomous Decentralized
Systems**

April 25 – 27, 1995

Phoenix, Arizona, USA

Sponsored by



IEEE Computer Society



Information Processing Society of Japan



The Society of Instrument and Control Engineers of Japan

In cooperation with



International Federation for Information Processing



International Federation for Automatic Control



IEEE Computer Society Press
Los Alamitos, California

Washington

•

Brussels

•

Tokyo