

# Interconnection Schemes for RTO.k Objects in Loosely Coupled Real-Time Distributed Computer Systems

K. H. (Kane) Kim and Chittur Subbaraman

Dept. of Electrical & Computer Engineering  
University of California  
Irvine, CA, 92697, U.S.A.

**Abstract:** A highly desired property in complex real-time distributed computer systems (DCS's) is the high degree of autonomy of the various subsystems that compose the DCS. In recent years, we have formulated two system structuring techniques which could be used as basic tools for enhancing the subsystem autonomy: (1) the RTO.k object structuring scheme and (2) the Hitachi-UCI (HU) Data Field (DF) inter-process-group communication scheme. The RTO.k object structuring scheme has been devised to enable the realization of real-time computing in the form of a generalization of non-real-time computing and yet allow system engineers to confidently produce certifiable real-time DCS's. The HU-DF scheme enables the enhancement of the object autonomy, especially the relocation autonomy and the data acceptance autonomy. Schemes for interconnecting RTO.k objects using the HU-DF scheme with the aim for maximizing the subsystem autonomy and easing the design-time guarantee of timely service capabilities of the objects are the main theme of this paper. A new approach for interconnecting RTO.k objects is presented. This and the earlier established basic interconnection approach are qualitatively analyzed mainly with respect to their contribution to autonomy and service timing predictability of the subsystems. Both of these approaches have been incorporated into an operating system kernel model named the DREAM kernel and its prototype implementation supporting both conventional real-time processes and RTO.k objects with guaranteed timely kernel services.

**Keywords:** autonomy, distributed computing, real-time object, RTO.k, data field, HU-DF, inter-RTO method communication, DREAM kernel.

## 1. Introduction

In order to meet the high expectations of the user community with regard to the ease of expansion, testing, verification, and on-line maintenance of complex real-time distributed computer systems (DCS's), the designers must construct the systems by using highly autonomous subsystems. On an equal level of importance, any design approach chosen must be able to guarantee the timely service capabilities of the system at the system design time in order for the designers to confidently certify the reliability of their designs. In recent years, we have formulated two basic techniques aimed for constructing highly predictable and yet cost-effective autonomous real-

time object-based DCS's: (1) the *RTO.k object structuring scheme* for modular structuring of complex real-time DCS's [Kim94a, Kim94b, Kim97], and (2) the *Hitachi-UCI-Data Field (HU-DF) scheme* for location-transparent inter-process-group communication in DCS's [Kim95a]. Various schemes for interconnecting RTO.k objects using the HU-DF scheme with the aim for maximizing the subsystem autonomy and easing the design-time guarantee of timely service capabilities of the objects are the main theme of this paper.

An abstract skeleton of a *real-time object model* was proposed a few years ago by the first co-author and Hermann Kopetz in [Kop90] as an approach for modular structuring of *hard-real-time* application systems. Recently, a concrete model with unambiguous syntactic structure and execution semantics was developed and the new object model was named the *RTO.k object model* (also called the time-triggered real-time object (TT-RTO) model) [Kim94a, Kim94b]. One of the most distinguishing features of the RTO.k object structuring scheme is that it is based on the following paradigms of real-time computing which may be called the GT (General-form Timeliness-guaranteed) design paradigms:

- (1) *General-form design style*: Future real-time computing must be realized in the form of a generalization of the non-real-time computing, as opposed to a form looking like an esoteric specialization.
- (2) *Design-time guarantee of timely service capabilities of subsystems*: To meet the demands of the general public on the assured reliability of future real-time computing systems (RTCS's) in safety-critical applications, it is inevitable 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).

In order for the RTO.k objects to provide guaranteed timely services to external clients, the *execution engine*, which consists of the hardware and the operating system and supports the execution of RTO.k objects, should itself provide guaranteed timely responses. Existing commercial operating systems lack the capability for either supporting a general-form design style or providing guaranteed timely services to application software. A model of an operating system kernel called the DREAM (Distributed Real-time Ever Available Micro-computing) kernel which can support real-time processes with

guaranteed timely services was formulated by the first co-author [Kim95b, Kim96]. The DREAM kernel can support both RTO.k object structured application software as well as conventional process-structured real-time application software. Several prototype implementations of the DREAM kernel have been produced by the authors and their research collaborators to run on local area networks (LAN's) of PC's connected by Ethernet. A prototype kernel (v.D3) has been used to run an RTO.k structured non-trivial defense command-control application, together with an RTO.k structured real-time simulator of the application environment. Several experiments of this kind dealing with different applications such as factory automation and intelligent traffic control have produced evidences that the RTO.k object structuring scheme was effective in the development of timeliness-guaranteed complex real-time DCS's.

The HU-DF scheme is an extension of the original data field (DF) scheme developed by Mori and other researchers at Hitachi, Ltd. [Mor93]. The essence of the DF 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 idiosyncrasies of the physical communication networks are transparent to the process designer. The HU-DF scheme differs from the original DF scheme in that the former allows dynamic flexible connection of processes to the logical channels and supports not only conventional *event messages* supported by the latter but also *state messages* which are based on the distributed replicated memory semantics [Kim95a, Kop89]. The logical multicast channels are called *data field channels* (DFC's).

The event message channel based approach for interconnecting RTO.k objects was discussed in [Kim95b] but it was restricted to the transparent mode of use in which the presence of event message channels was transparent to the RTO.k object programmer while the channels provided connections between client objects and remote service methods. A validated implementation technique for this connection approach is presented in this paper.

In addition, a new approach for interconnecting RTO.k objects is presented in this paper. In the new approach, the RTO.k object programmer explicitly creates a DFC to be used by an explicitly specified group of RTO.k objects in the multicast mode. In a sense, the programming tool and the kernel support associated with this interconnection mode can be viewed as an approach for implementing the state message communication outlined in [Kim95a].

This paper also analyzes the two interconnection modes mainly with regard to their contribution to autonomy and service timing predictability of the subsystems. We believe that the newly presented implementation approaches and the qualitative analysis add considerably to the useful guides available to future

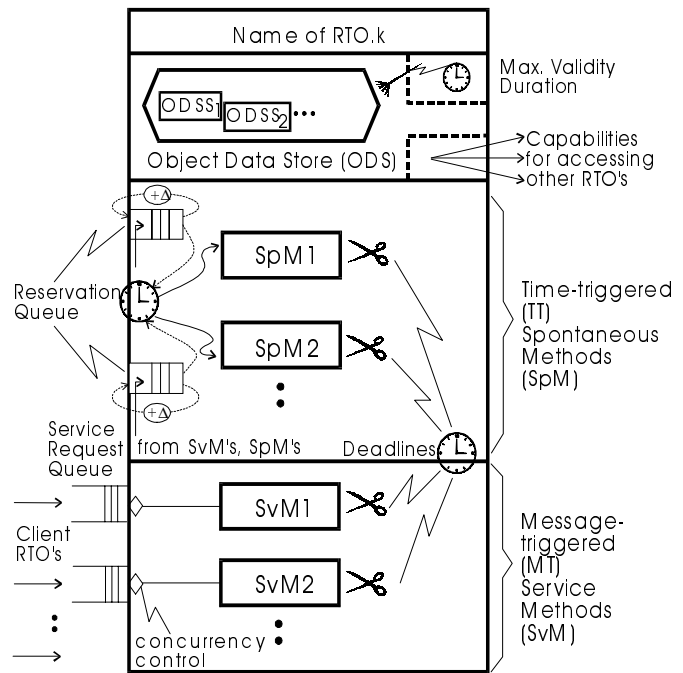


Figure 1. The RTO.k object structure (Adapted from [Kim94a])

object-oriented (OO) real-time DCS designers and would further contribute to efficient development of autonomous predictable real-time DCS's.

The paper starts in Section 2 with a discussion on the essence of the RTO.k object structuring scheme. Section 3 discusses the salient features of the HU-DF scheme. Sections 4 and 5 present two different implementation approaches adopted in the DREAM kernel to connect RTO.k objects. The next section, Section 6, qualitatively analyzes these two approaches. The paper concludes in Section 7 with discussions on the issues to be resolved via future research.

## 2. The RTO.k object structuring scheme

The basic structure of an RTO.k object [Kim94a, Kim94b, Kim97] is depicted in Figure 1. It is an extension of the conventional object structure and three most important extensions are the following:

(a) *Two clearly separated groups of methods:*

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

Incorporation of SpM's means introducing the potential for the following two new types of concurrent executions of object methods in addition to the potential for concurrent executions of SvM's that exist in conventional objects:

(Type I) Concurrency among SpM executions: This concurrency is specified in an implicit but natural manner, e.g., two SpM's designed to be triggered at 10 am.

(Type II) Concurrency between SpM executions and SvM executions.

(b) *Basic concurrency constraint* (BCC):

In order to dramatically reduce the designer's efforts in guaranteeing timely service capabilities of RTO.k objects, the execution rule which prevents conflicts between SpM's and SvM's is incorporated. 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*. To be exact, when a message-triggered SvM is not free of conflict with an SpM in accessing the same portion of the *object data store* (ODS), execution of the former method (SvM) must not be allowed in a time zone earmarked for a TT-execution of the latter method (SpM). This restriction is called the basic concurrency constraint (BCC). Therefore, SpM's are given higher priorities for execution over the SvM's. Executions of SpM's are not disturbed by SvM executions and triggering times of SpM's are fixed at the design time. At least this makes it very easy to analyze the execution time behavior of SpM's. For example, if a statement of the type "at 10am do S" appears in an SpM, its reliable execution can be easily assured.

(c) For each execution of a method of an RTO.k object (to be exact, for each output action to occur during a method execution), a *deadline* is imposed.

The first two features (a) and (b) mentioned above make the RTO.k object model clearly distinguished from other proposed real-time object models [Att91, Bih89, Ish92, Tak92, Yau96].

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 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".

An underlying design philosophy of the RTO.k object structuring 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 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. 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. Again, the BCC contributes to major reduction of these burdens imposed on the designer. The DREAM kernel [Kim95b, Kim96] has been formulated as a model of an operating system kernel that can support guaranteed timely services not only for RTO.k object structured real-time applications but also for conventional process-structured real-time applications.

### 3. Overview of the HU data field (HU-DF) scheme for inter-process-group communication

The main features of the HU data field (HU-DF) scheme are discussed in this section.

#### 3.1 Logical multicast channels and dynamic connection of a process to channels

The original DF scheme supports establishment of logical multicast channels called *data field channels* (DFC's) shared among concurrent distributed processes for their interaction without necessitating the processes to know the identities of other cooperating processes or the physical node addresses of the nodes executing the processes [Mor93]. The only thing that the group of cooperating processes need to know in advance for message communication is the name of the DFC through which the message will be communicated. Each of those

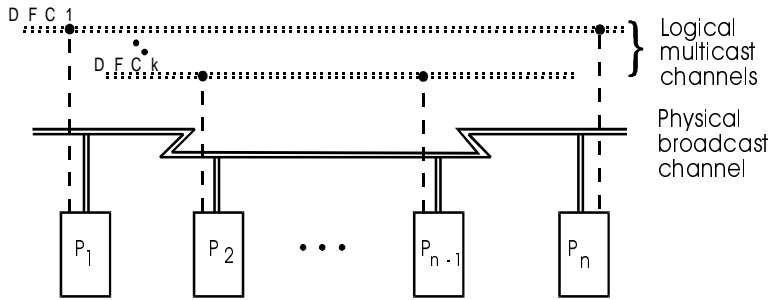


Figure 2. DFC's vs. physical channels (Adapted from [Kim95a])

processes can simply educate the communication subsystem in the host node about the DFC to be used.

With popular CSMA and token ring architectures, establishing a DFC to be shared among a group of processes boils down to choosing a logical channel ID number, called *content code* in [Mor93] and also called DFCID here, to be included in the address field of each message broadcast over the CSMA bus or token ring. Figure 2 depicts such an arrangement. The processing nodes connected to the DFC can see all the messages coming through the physical broadcast facility but will pay attention only to those messages containing relevant DFCID's. This means that when processes are designed to communicate via DFC's 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 DFC X". Then all other processes set up to share DFC X will pick up the message.

In the HU-DF scheme, a process can freely connect itself to or disconnect itself from a DFC, which is somewhat restricted in the original DF scheme [Kim95a]. The communication subsystem supports primitives such as "Connect to DFC X", "Disconnect from DFC Y", etc. A process can also be connected to multiple DFC's at the same time, thereby joining multiple communicating groups.

### 3.2 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, Kim95a]. An event message carries information about an event occurrence and every event message should be read by the intended receiver. An event message is supposed not to be overwritten 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 the frequency at which a receiver reads the contents of its state message memory. A state message is thus used to share periodically observed state information about a dynamic object, e.g., temperature of a room.

## 4. Automated creation of DFC's to support service methods (SvM's)

In this section, we discuss the salient features of the basic inter-RTO communication approach of using SvM calls alone and its implementation approach adopted as a part of the DREAM kernel. The client methods make SvM calls by sending request messages on specific DFC's, each associated with a particular SvM. DFC-based communication contributes in a major way to enhancing the object autonomy.

This mode of inter-RTO communication is illustrated in Figure 3. An external client first sends a service request over DFC<sub>1</sub> to SvM1 of RTO1. In response, the SvM1 updates an ODS segment (ODSS) appropriately. The sibling SpM, SpM1, which periodically polls this ODSS for any work orders, reads this ODSS, processes the service request partially, and sends the partial results over DFC<sub>2</sub> as a service request message to SvM1 of RTO2. Similar to the SvM1 of RTO1, the SvM1 of RTO2 also updates an ODSS and the sibling SpM1 reads the ODSS, finishes the processing of the request, and sends the results back to the external client over the DFC<sub>3</sub>. As we mentioned before, the creation of these DFC's as well as the connection of the object methods to these DFC's are done automatically by the execution engine (e.g., DREAM kernel).

In the following subsections, we discuss the major implementation approaches adopted in the DREAM kernel for supporting inter-RTO method communication using SvM calls alone. The manner in which the DREAM kernel executes RTO.k object methods is discussed first for the sake of identifying the unique characteristics of the communicating parties involved here.

### 4.1 Composing an RTO.k object using PCD program components

The DREAM kernel as a *process execution engine* supports the following three types of concurrent and distributed program components:

(1) Processes: In this paper, these are also called *application processes* (AP's) although in reality, some of these processes may play the roles of system management processes, e.g., processes managing I/O. The processes

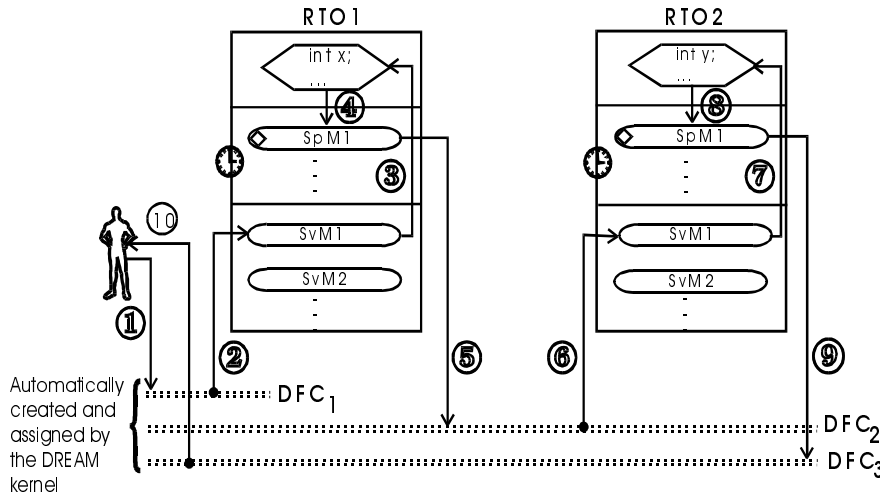


Figure 3. Inter-RTO communication through SvM calls

may frequently impose deadlines on the kernel for execution of their next computation-segments.

(2) **CREW (concurrent-read-&-exclusive-write) monitors:** The CREW monitor is a shared data structure monitor and possesses the *readers-writers* semantics (i.e., *concurrent-read-&-exclusive-write* semantics) instead of the exclusive-read-&-exclusive-write semantics associated with the conventional monitor.

(3) DFC's in the HU-DF scheme.

These three basic components represent fully general facilities for concurrent and distributed program structuring. Therefore, any operating system kernel that supports these three components along with various I/O operations can be viewed as a general purpose kernel. Programs composed of the three types of components are called the **PCD (process-CREW-DFC) programs**.

The DREAM kernel treats components of RTO.k objects as special types of PCD components, i.e., special types of processes, CREW monitors, and DFC's. Thus, the components of an RTO.k object would be mapped to the corresponding components of an equivalent PCD-program as follows:

- (1) both SpM's and SvM's are treated as application-specific program bodies of processes (called method execution processes or MEP's),
- (2) access paths (i.e., service request paths) to SvM's as DFC's created by the SvM's, and
- (3) result-return paths to clients as DFC's created by the clients,
- (4) ODSS's as special CREW monitors,
- (5) capabilities for accessing SvM's in other RTO.k objects as ID's of DFC's.

The advantages of this type of mapping include supporting both PCD and RTO.k programs with a small kernel, facilitating the transparency of object locations, facilitating concurrent access by multiple reader methods to an ODSS, etc. On the other hand, a limitation of this

mapping approach is in that efficient implementation of DFC's is possible only if the underlying communication network provides efficient broadcast capability.

#### 4.2 Creation of DFC's and connection of RTO.k methods to the DFC's

One of the main issues in the design of schemes for inter-RTO method communication stems from the fact that the communicating parties are object methods which may execute on different MEP's during different invocations. Moreover, multiple object methods may execute on the same MEP. This means that the types of communicating parties which can

use DFC's need to be expanded to include object methods in addition to processes.

One of the most essential services that must be provided by the execution engine is to establish for an SvM in a server RTO.k object a service request (SR) path (to the SvM) from each potential client RTO.k object. The approach adopted in the DREAM kernel is as follows. When an SvM is registered in the DREAM kernel in a node, the kernel first assigns a new SR DFC (which has an ID unique throughout the DCS) for the SvM, and connects the SvM to this DFC. The kernel then broadcasts the SR DFC ID of the SvM and the symbolic name of the SvM to all other nodes that may host potential client RTO.k objects. This broadcast is made over a special global broadcast DFC to which every node participating in a distributed computing application is connected to. Then the kernels in the receiving nodes record the broadcasted SR DFC ID along with the SvM name. Therefore, when a client RTO.k object asks the DREAM kernel for establishing an access path to an "external" SvM, chances are that the local supporting kernel already has the record on the SR DFC ID for the SvM. In the special occasions where the local supporting kernel does not have the needed SR DFC ID record, the kernel must obtain the DFC ID over the special global broadcast DFC through cooperation of peer kernels in other nodes.

Ordinarily, only *processes* could be connected to a DFC. Whenever, a process is connected to a DFC, the kernel executing in the node delivers all messages received over the DFC to the connected process. On the other hand, if no process in a node is connected to a certain DFC, the kernel in the node ignores all messages arriving over the DFC. In the case of an SvM, it is practically impossible to connect the MEP which will be assigned to the SvM to an SR DFC. This is because the MEP which will be assigned to the SvM is not known until the SvM is invoked, i.e., until the service request for

the SvM arrives. (An MEP is assigned upon each invocation of an SvM to allow pipelining, i.e., multiple invocations of the same SvM proceeding concurrently.) This problem is solved in the DREAM kernel as follows. When an SvM is registered with the kernel, the kernel assigns a *unique virtual process ID* (VPID) to the SvM and connects this virtual process to the SR DFC of the SvM. Consequently, any message received over this DFC is kept in the appropriate DFC buffer until the RTO.k support module of the DREAM kernel picks the (service request) message. In a sense, each SvM is connected to its SR DFC.

In addition to being connected to an SR DFC, the SvM needs to be connected to the result-return (RR) DFC which the SvM's client has provided for obtaining the results from the SvM. Since there could be multiple invocations of the same SvM occurring concurrently, each invoking client should receive the return results over its own unique DFC. Hence, instead of connecting an SvM to an RR DFC, the MEP which is assigned to an SvM invocation is connected to an RR DFC. On the other hand, each client (method execution) may have just one unique RR DFC and may reuse it for calling different SvM's. In a fashion similar to connecting an SvM to its SR DFC at the registration time, an SvM's client is connected to its RR DFC at the registration time.

A major advantage of the above implementation technique is in that the process of determining the DFC ID's is fully handled by the kernel in a manner transparent to the RTO.k designer.

## 5. Multicast path

In the preceding section we discussed the major implementation approach adopted in the DREAM kernel for the facilities for communication among RTO.k object methods via SvM calls. However, in some applications, using the multicast (or one-to-many) form of

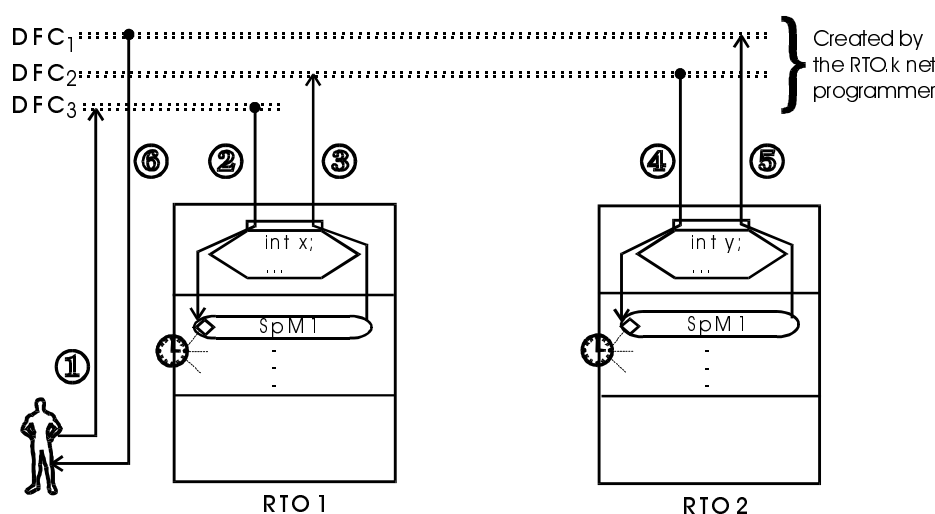


Figure 4. Interaction among RTO's through the DFC's declared in the RAC section

communication among object methods is highly beneficial. Multicast is particularly important for efficient implementation of fault tolerance schemes in which redundant replicated objects that execute on different nodes need copies of the same client request messages.

### 5.1 Basic approach

The basic approach developed is as follows. The RTO.k object programmer explicitly designs such that when an object method is registered with the DREAM kernel, the object method can inform the kernel that it "wishes to be connected to a multicast DFC with ID X". Similar requests could be made by other object methods which may or may not be resident in the same node hosting this object method. This ID "X" should preferably be a symbolic name chosen by the application programmer and the kernels executing in different nodes in the system should cooperate with each other to convert this symbolic name into a DFC ID number.

This mode of inter-RTO communication is illustrated in Figure 4. An external client sends a service request over a DFC, say DFC<sub>3</sub>, to which RTO1 is connected to. Consequently, SpM1 of RTO1, which periodically polls the message buffer for DFC<sub>3</sub> in the local host node, picks the request and processes it. After partially processing it, SpM1 sends the partial results over DFC<sub>2</sub> from which SpM1 of RTO2 will receive the message through periodic polling. The latter SpM will then complete the processing and return the final results back to the external client over DFC<sub>1</sub>. From this example, one can envision an extreme case where no SvM's are used in a network of RTO.k objects.

### 5.2 Communication via state messages

Any process which is connected to a DFC can read all the state message variables that are associated with that DFC. A state message is never deleted upon reading and is always overwritten whenever a new one arrives.

RTO.k object methods may communicate via state messages in the following manner. Globally shared data space segments may be constructed by declaring DFC's including state message variables as global variables and establishing connection to the programmer-specified DFC's in the RTO access capability sections of RTO.k objects. A state message variable may be updated by a producer method in one RTO.k object and the most recently available values of the variable in other objects may be read by their resident consumer method(s). The rate of production and the rate of consumption could be vastly different.

In the above arrangement, it is possible to have RTO.k objects without any SvM's but a trivial SvM used for object initialization. Then a direct call from a client RTO.k object for an SvM in a server RTO.k object is replaced by a state message carrying a work order from a new-style client RTO.k object which is to be picked up by an SpM in a new-style server RTO.k object at its convenient time. In this way, the data acceptance autonomy is realized. The result is also the ultimate loosening of the connection between producer and consumer objects.

## 6. Comparisons of the two basic types of RTO.k object interconnections with respect to object autonomy and timing predictability

The implementation approaches presented in sections 4 and 5 suggest two basic types of interconnections among RTO.k objects; (1) approach C1: interconnecting RTO.k objects using SvM calls alone, and (2) approach C2: interconnecting RTO.k objects via accessible DFC declarations in the RAC section of each RTO.k object. In this section we present a qualitative analysis of these two major implementation approaches and compare them mainly with regard to their contribution to object autonomy and timing predictability.

(1) Relocation autonomy of objects: It is highly desirable to allow RTO.k objects in a DCS to have maximum autonomy in choosing their residence, i.e., maximum relocation autonomy. This is particularly important for purpose of dynamic reconfiguration for load balancing, fault tolerance, etc. Both approaches C1 and C2 utilize DFC's for inter-method communication and equally facilitate relocation autonomy for an RTO.k object.

(2) Data acceptance autonomy of objects: Approach C2 facilitates data acceptance autonomy of objects, i.e., autonomy in choosing its times for accepting real-time input data, unlike approach C1. This is because, using approach C2, one can connect an RTO to one or more DFC's and this RTO can receive not only event messages but also state messages over the DFC. Such a state-message based interconnection makes the connection between producer RTO.k objects and consumer objects very loose. 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.

(3) Timing predictability: Approach C2 would lead to the requirement of less efforts for ensuring service timeliness of an RTO than approach C1 would. This is because of the following major reasons:

(a) Generally, the timing aspects of SpM executions are more accurately predicted than those of SvM executions.

(b) If two SpM's need to access common ODSS's in an exclusive mode, the designer may choose to make their executions sequential to prevent predictability-damaging ODSS locking. This is one of several possible approaches for ensuring that the timing aspects of both SpM executions are highly predictable. On the other hand, if we use approach C1, then an analysis of possible conflicts among different SvM's for ODSS access requires much bigger efforts.

(c) In approach C1, a client can receive "the services of an SpM" only by making a service request to an SvM affiliated with the SpM in the same RTO. In approach C2, a client can directly send an application-level service request message to an SpM over the DFC to which the RTO containing the SpM is connected. This makes the service request path shorter than that in the former approach. The following simple example illustrates this point.

Let us first consider approach C1. Consider an RTO with one SpM and one SvM. The AAC expression for the SpM is defined as follows:

"for t = from 10am to 5:45pm every 30min  
start-during (t, t+1min) finish-by t+25min".

The function of the SvM here is merely to place a work order for the SpM. The SvM completion deadline is 4 min. Also, the SpM and the SvM access a common ODSS, ODSS<sub>7</sub>, in the exclusive mode.

Suppose a client request arrives at the node hosting the RTO at 10:59 am. Then, the kernel executing in the node will find out that the SvM cannot be activated (without violating the BCC) since there is a scheduled execution for an SpM (that shares ODSS<sub>7</sub> with the SvM) at 11:00 am. The kernel would then postpone the SvM activation until the SpM execution is over, i.e., until 11:25 am in the worst case. At 11:25am, the SvM will be activated and it will process the client request and thus place a work order for the SpM. Since the next activation for the SpM is at 11:30am and the execution deadline is 11:55 am, the result for the client will be generated at the latest by 11:55am.

Now, let us consider approach C2. In this case, the RTO needs to have only one SpM with the same AAC expression as shown above. So, if the client request arrives at the node at 10:59am, the next SpM execution starting at 11:00 am can process the client request. The result for the client will be generated at the latest by 11:25am. Therefore, the service request path using approach C2 is shorter than the service request path using approach C1.

(4) Replication of objects: Under C2, the application programmer can create active replicas of an RTO.k object without necessitating new supports from the kernel whereas under C1, it requires new mechanisms in the kernel [Kim97].

(5) Programmer's creation of DFC's: In the case of approach C2, the application programmer determines the DFC's to which an RTO needs to be connected to. On the other hand, in approach C1, the application designer is removed of the burden of determining the DFC ID's.

(6) Execution overhead: If the demands from clients are light, approach C2 might incur more overhead than approach C1. This is because, in approach C2, the SpM's function as servers and they typically execute periodically, whereas in approach C1, the SvM's respond to service requests at their earliest possible times. A consequence of this is that, it may not be easy to execute a lot of SpM's in one node if we use approach C2.

## 7. Conclusion

The RTO.k object structuring scheme and the HU-DF scheme have been demonstrated to some extent as cost-effective techniques for the construction of autonomous and predictable real-time DCS's. This paper presented two major approaches for interconnecting RTO.k objects that have been implemented as a part of the DREAM kernel and qualitatively analyzed these approaches mainly with regard to their contribution to object autonomy and timing predictability. The implementation approaches presented here are believed to be amenable to easy adaptation for a variety of system environments aimed to facilitate highly decentralized real-time computer applications. One desirable direction for enhancing the implementation model of the HU-DF scheme in the near future is to incorporate the time division multiplexed access control and real-time network surveillance and reconfiguration capabilities. Another remaining challenge is to adapt the HU-DF scheme to point-to-point networks.

**Acknowledgments:** 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 US DARPA under Contract No. N66001-97-C-8516 monitored by NRaD, in part by the University of California MICRO Program under Grant No. 96-169, in part by Hitachi, Ltd, in part by Postech, and in part by LG CIT.

## References

[Att91] Attoui, A. "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.

[Ish92] Ishikawa, Y., Tokuda, H., and Mercer, C. W., "An Object-Oriented Real-Time Programming Language", *IEEE Computer*, October 1992, pp. 66-73.

[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 Kopetz, H., "A Real-Time Object Model RTO.k and an Experimental Investigation of Its Potentials", *Proc. 1994 IEEE Computer Society's Computer Software and Applications Conf. (COMPSAC)*, Nov. 1994, Taipei, pp. 392-402.

[Kim95a] Kim, K.H., Mori, K., and Nakanishi, H., "Realization of Autonomous Decentralized Computing with the RTO.k Object Structuring Scheme and the HU-DF Inter-Process-Group Communication Scheme", *Proc. IEEE CS Int'l Symp. on Autonomous Decentralized Systems (ISADS 95)*, Mar. 1995, Arizona, pp. 305-312.

[Kim95b] Kim, K.H. et al., "A Timeliness-Guaranteed Kernel Model - DREAM Kernel and Implementation Techniques", *Proc. 1995 Int'l Workshop on Real-Time Computing Systems and Applications (RTCSA 95)*, Tokyo, Japan, Oct. 1995, pp.80-87.

[Kim96] Kim, K.H. et.al., "The DREAM Library Support for PCD and RTO.k programming in C++", *Proc. 1996 IEEE CS Workshop on Object-oriented Real-time Dependable Systems (WORDS)*, Feb. 96, Laguna Beach, pp. 59-68.

[Kim97] K.H. Kim, and C. Subbaraman, "Fault-Tolerant Real-Time Objects", *Communications of the ACM*, January 1997, pp. 75-82.

[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 Computer Society's 9th Symp. on Reliable Distributed Systems*, Huntsville, AL, Oct. 1990, pp.165-174.

[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.

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

[Yau96] Yau, S., et.al., "Object-oriented Software Development with Fault Tolerance for Distributed Real-time Systems", *Proc. 1996 IEEE CS Workshop on Object-oriented Real-time Dependable Systems (WORDS)*, Feb. 96, Laguna Beach, pp. 160-167.