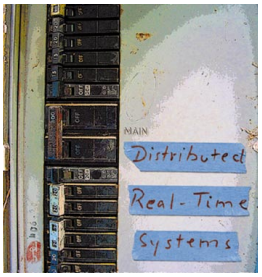
APIs for Real-Time Distributed Object Programming



This article focuses on application programming interfaces (APIs) that take the form of C++ and Java class libraries and support high-level, high-precision, real-time object programming without requiring new language translators.

K.H. (Kane) Kim
University of California, Irvine

An ideal real-time distributed programming method should be based on a general high-level style that could be easily accommodated by application programmers using C++ and Java. If such a method were to exist, these programmers could specify the interactions among distributed components and the timing requirements of various actions without expending much effort. In an ideal world, this kind of programming method would also allow system engineers who deal with safety-critical applications to produce certifiable, real-time distributed computing systems. The research and development efforts on new distributed real-time programming tools have shown rapid growth in recent years—with real-time CORBA, Java, UML, TMO, and others serving as examples—but the industry is still far from achieving these ideals.[1-8]

Consider, for example, complex real-time applications like intelligent ground transportation systems, automated pilots, natural emergency management systems, and widely distributed multisensor-based defense systems. While improved methods for engineering these kinds of real-time systems appear continuously these days, the state of the art remains inadequate for producing applications that offer sufficient reliability. Even so, research efforts geared toward developing innovative approaches are steadily intensifying.

For example, the time-triggered message-triggered object (TMO) developed by this author and his colleagues is a syntactically simple and natural but semantically powerful extension of the conventional object structure.[7-8] As such, its support tools can be based on various well-established OO languages like C++ and Java, and on commercial real-time operating system kernels.

Facilitating high-level, high-precision, real-time object programming by establishing some form of language tools has become a subject of great interest to the embedded systems community. This article focuses on application programming interfaces (APIs) that take the form of C++ and Java class libraries and support high-level, high-precision, real-time object programming without requiring new language translators. These APIs wrap the services of the real-time object execution engines, which consist of hardware, node OSs, and middleware; they enable convenient high-level programming almost to the extent that a new real-time object language can.

FUNDAMENTAL FEATURES

Any practical real-time distributed programming facility must enable efficient specification and execution-control of the following:

- past, present, and future time-referencing;
- uniform method invocation of both local and remote objects;
- deadline imposition for arrival of the results from the invoked object method;
- time-triggered actions;
- concurrent object-method execution; and
- nonblocking invocation of object methods.

Several APIs let you specify and control the execution of each of these fundamental operations.

0018-9162/00/$10.00 © 2000 IEEE

Global time base

To engender efficient, distributed real-time computing, you must establish a global time base that supplies the time to distributed objects within computing nodes. There are several ways of establishing global time bases, each offering varying degrees of precision. The global positioning system (GPS) is one example.[9]

A real-time API must include function "now()", which involves a call to the execution engine to obtain the real-time value. This value must be sufficiently close to the current real-time value kept by the authentic sources such as GPS or the Universal Time Coordinated (UTC) system. Otherwise, interactions among the objects running on geographically dispersed object execution engines, each maintaining a separate time base, can lead to unpredictable results.

The distributed computer system age (DCS_age)—another useful time-related facility—is equivalent to the number of microseconds past the distributed computer system start time (DCS_start_time). Instead of expressing a real-time value based on year, month, day, hour, minute, second, millisecond, and microsecond, you can express a real-time value in the convenient form of DCS_age.

You can establish the DCS_start_time in many ways. For example, the master node first confirms the readiness of all the nodes belonging to the initial DCS configuration and then announces to all worker nodes the DCS_start_time. When the DCS_start_time arrives, all the nodes belonging to the DCS configuration start executing their parts of the program.

Distributable software component

While conventional C++ objects are nondistributable passive units, distributed real-time objects such as TMOs are active. Moreover, some applications must be capable of adapting to dynamic changes in network configurations. The real-time objects that make up such applications need to be dynamically migrated across node boundaries. This means that each real-time object and each of its service methods (SvMs)—methods of the object that can be called from outside it—must have a unique, logical, systemwide name. Cooperating execution engines should be capable of locating the SvM that corresponds to the symbolic ID presented by the client (the main function or a method of another real-time object).

One cost-effective arrangement for avoiding or minimizing the overhead incurred in calling SvMs by their symbolic names works like this: When a real-time object is instantiated, the symbolic name for the object, say "RTO1," and the symbolic name for each SvM in that object, should be registered with the execution engine. Suppose the real-time object has only one SvM for which the registered name is "SvM7." Then a gate object corresponding to "SvM7" should be created in every potential host node of a client real-time object. The gate is an entry point to the efficient call-path leading to the associated SvM. When a gate is instantiated, the symbolic name of the associated SvM is used as a parameter for the constructor. Thereafter, client objects can just use the gate's name in calling the SvM, without using the symbolic name, as shown in Figure 1.

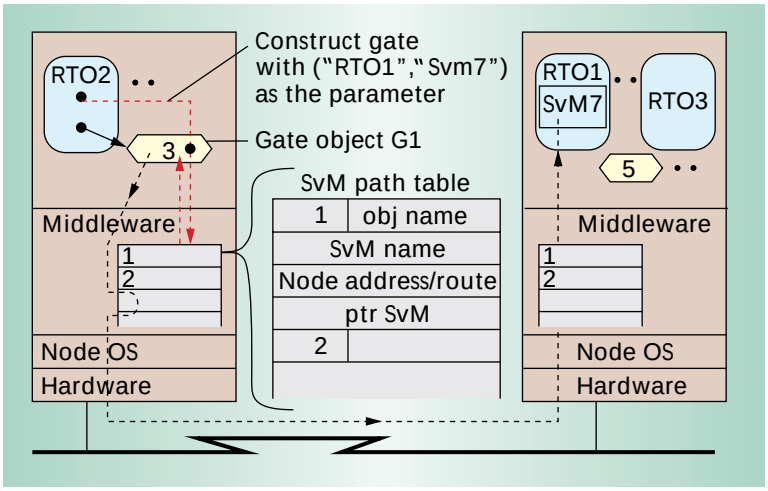


Figure 1. A gate object providing an efficient call path. The gate object G1 is a front-end interface of an automatically created call-path to remote object method RTO1.SvM7.

To enable easy gate creation, the API can contain a class named `GateClass`. Then the gate for the SvM named "SvM7" can be created like this:

```
GateClass G1 (
"RTO1", "SvM7", /*service-start-
  time*/ t_DCS_age (3x1000x1000) ).
```

Gate G1 should be used after the service-start-time, which is three seconds past the DCS_start_time. The sevice_start_time should be chosen so that, by the start time, the execution engine will be able to establish an efficient call-path to SvM7. Establishing the call path may involve locating the node hosting RTO1, identifying the address or the numeric ID of RTO1.SvM7, or identifying an efficient message route to the node. This gate can also be extended to incorporate security-enforcement capabilities.

Structuring of component parameters

It is sensible to provide the parameter structure's definition for the SvM as a companion to the gate-creation statement. For example, the gate-creation statement may have the following companion:

```
struct ParamStruct_RTO1class_SvM7
  { int a; float b; } ;
```

A client real-time object may then contain an instantiation such as `ParamStruct_RTO1class_SvM7 param1`. If you're not going to use a new lan-

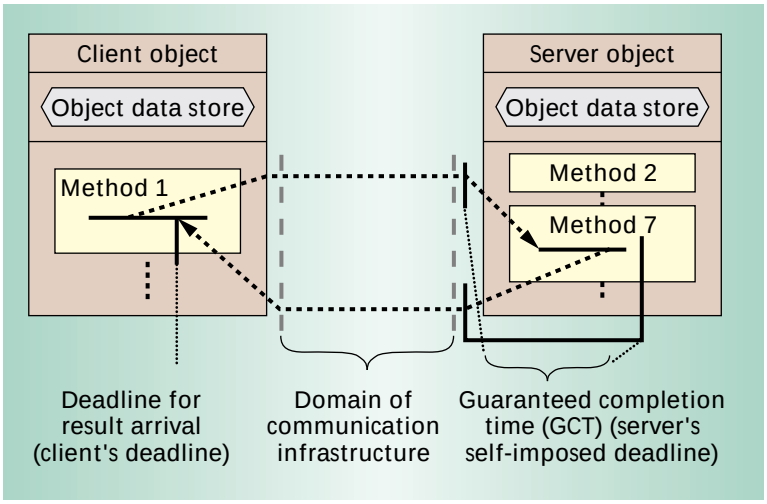


Figure 2. The client's deadline for result arrival is set by the programmer with the understanding of the server's GCT and the transmission times to be consumed by the communication infrastructure.

guage translator, then you must group all parameters into a single structured variable and let the client pass a pointer to the execution engine along with the information on the size of the variable's memory area. The client engine then transfers the structured parameter as a single message across the network.

This restriction can be removed in the CORBA that uses an IDL translator.[6] The programmer of a CORBA object class produces an IDL specification that contains, in addition to the class, the method names and method parameters. An IDL translator then takes the IDL specification as an input and produces two program modules—one called the stub, for use by the client objects, and the other called the skeleton, for use by the server object. The stub-skeleton pair takes care of parameter transfer across the network and may perform multiple message exchanges to handle a large set of parameters.

To facilitate calls to SvMs, you can place various types of call operations as methods inside the GateClass. Among several methods for such calls, the most basic is a blocking service request method, BlockingSR1. For example, you can use `G1.BlockingSR1 (&param1, sizeof(Param-Struct), 50x1000)` to call the SvM named "SvM7," with param1 as the parameter and the deadline for result return set as 50 milliseconds after the calling time.

Concurrency in SvM executions

Within a real-time object, you can facilitate concurrency in executions in several ways. One approach that offers great flexibility in concurrency control while still yielding a relatively easy procedure for analyzing worst-case timing behavior is to explicitly indicate the needs of SvMs for accessing the set of object data variables, collectively called the object data store (ODS).

The ODS segment (ODSS) is a basic unit of data storage—a group of variables—that can be reserved for exclusive access by a real-time object method. If you explicitly indicate the group of ODSSs that may be accessed by each SvM, even the object execution engine can easily check whether two SvM executions may interfere with each other or not. Recall that when a distributed real-time object's SvM is created, its symbolic name must be registered with the execution engine. The ODSS-based concurrency control approach suggests that the set of ODSSs to be accessed by the SvM must also be registered with the execution engine so that the latter may use the registered information in effecting concurrent SvM executions. More specifically, when a distributed real-time object's SvM is created and initialized, the following (at the very least) must be registered with the object execution engine:

- the symbolic name, such as SvM1 or Update_Speed, and
- the set of ODSS-name-access-mode-indicator pairs, such as `{ (ODSS1, read-write), (ODSS2, read-only) }`.

A desirable API must therefore include a class, say `ODSSBaseClass`, that defines basic operations associated with an ODSS, including registration of the ODSS with the execution engine and locking the ODSS for read-only or read-write access. An application-specific ODSS, say ODSS1, can then be defined as a derived class of ODSSBaseClass. When instantiated, it is automatically registered with the engine.

Guaranteed completion times

To enable systematic modular construction of reliable applications, real-time objects must offer guaranteed timely services. By advertising the guaranteed completion time (GCT) of each SvM, the server object's designer guarantees the object's timely services. As shown in Figure 2, the GCT of an SvM is the upper bound on the time duration, from the instant at which the service request message arrives at the node hosting the server object to the instant at which both the SvM execution and the preparation of a message containing the return parameter become complete. Before determining GCTs, the server object designer must make sure that with the available object execution engine the server object can be implemented such that its SvMs are always executed within the GCTs.

On the other hand, as shown in Figure 2, the client imposes a deadline for returning results. The client execution engine receives this deadline as one of the parameters associated with the remote method call; the engine checks whether or not the result comes back within the client's deadline. Three sources can create a fault, causing a client's deadline to be violated:

- the client object's resources, which are basically node facility (hardware plus OS);

- the communication infrastructure; and
- the server object's resources, which include not only node facility but also the object code.

Thus, while the server is responsible for finishing a service within the GCT, the following relationship must hold between the GCT and the client's deadline:

(deadline for result arrival minus call initiation time) > (maximum transmission time imposed on the communication infrastructure plus GCT of SvM)

The GCT of an SvM is inherently subject to the condition that the aggregate arrival rate of calls from all possible clients to the SvM does not exceed the maximum invocation rate (MIR). When an SvM is created and initialized, the GCT and the MIR must also be registered with the engine.

Time-triggered action and method

Time-triggered (TT) computations distinguish real-time programming from non-real-time programming. In principle, the TT computation unit can be any one of the following:

- a simple statement, such as an assignment statement with the right-side expression restricted to an arithmetic logical expression type that involves neither a control flow expression nor a function call;
- a compound statement like if-then-else or while-do;
- a statement block;
- a function and procedure; or
- an object method.

TT actions associated with a computation unit may include TT initiation of the computation unit, timely completion of the computation unit, and periodic execution. In any practical real-time programming language, it is desirable to have the following type of a construct, called the autonomous activation condition (AAC):

```
" for <time-var> = from <activation-
  time> to <deactivation-time>
[every <period>]
start-during (<earliest-start-time>,
  <latest-start-time>)
finish-by <guaranteed completion time> "
```

For example, consider the following case:

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

This case specifies that the associated computation unit must be executed every 30 minutes, starting at 10 AM and running until 10:50 AM, and that each execution must start at any time within the 5-minute interval (t, t +5min) and must be completed by t + 10min. So it has the same effect as

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

The object method is the most important, feasible unit for time-triggered initiations and completion time checks.

Of the five types of computation units mentioned here, the object method is the most important, feasible unit for TT initiations and completion time checks. The only execution engines built so far that fully allow the specifications of TT initiations, completion deadlines, and periodic executions to be associated with object methods are the TMO execution engines.[10-11] This situation holds mainly because developers discovered the importance of TT methods in high-level real-time distributed programming only a few years ago. However, support for TT methods will likely appear in an increasing number of major industrial real-time tools. At the same time, it is important to use TT methods in well-structured and disciplined ways since, otherwise, you'll produce difficult-to-analyze error-prone real-time programs. The TMO programming scheme best illustrates the proper way to use TT methods.

Object structure with time-triggered methods

The basic structure of the TMO model consists of four parts:

- object-data-store section (ODS-sec): the list of object-data-store segments (ODSSs);
- environment-access-capability section (EAC-sec): the list of gate objects, logical communication channels, and I/O device interfaces;
- spontaneous-method section (SpM-sec): the list of spontaneous methods; and
- service-method section (SvM-sec).

Figure 3 depicts TMO, a distributed computing component that offers unique extensions to conventional objects. These extensions include the spontaneous method (clearly separated from the service method) and basic concurrency constraint.

Spontaneous method. The TMO may contain spontaneous methods (SpMs), which are TT methods clearly separated from the conventional SvMs. The SpM executions trigger upon reaching the real-time clock at specific values determined at design time. The SvM executions, on the other hand, are triggered by service request messages from clients. Moreover, actions to be taken at real times, which can be determined at the

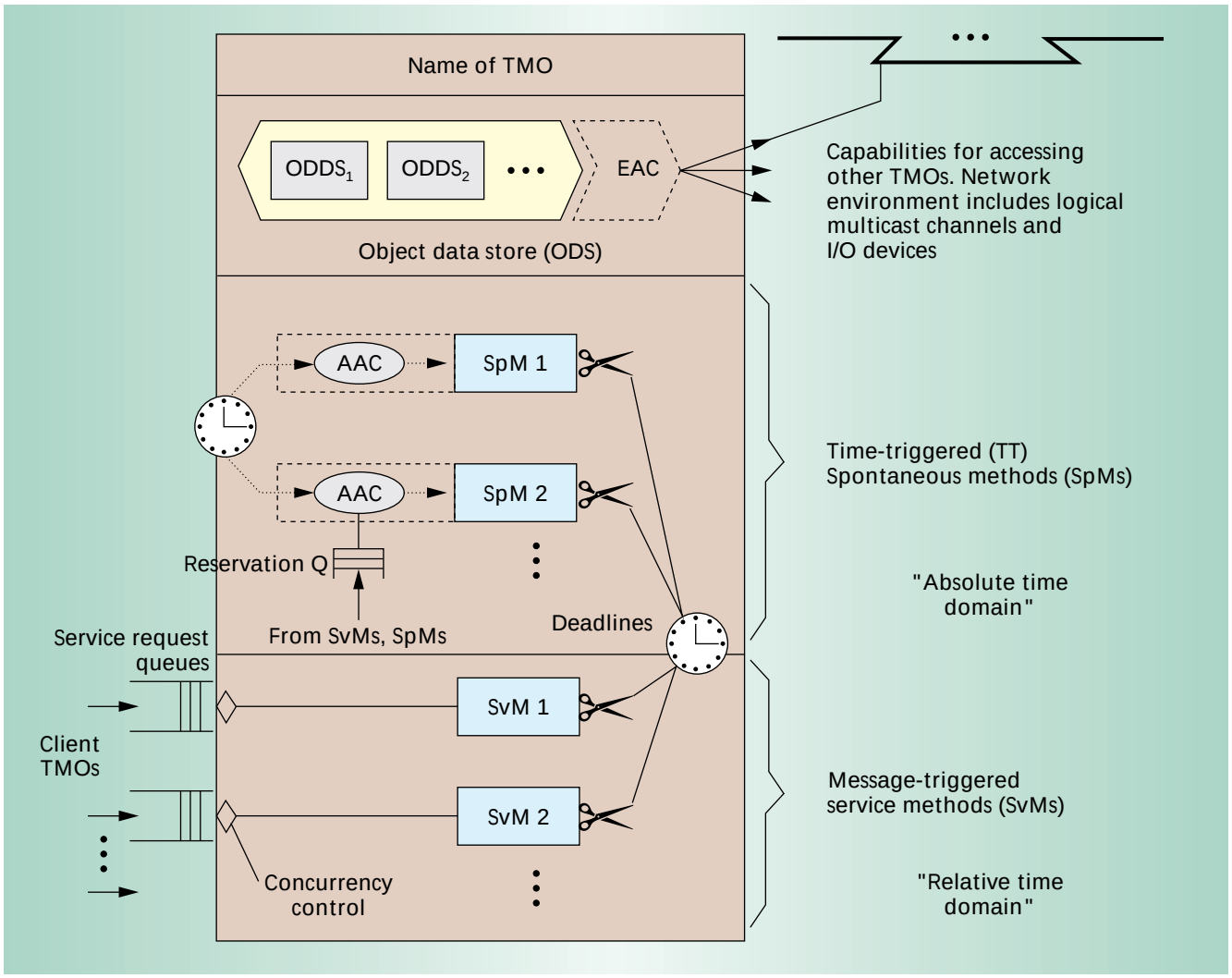


Figure 3. Structure of the time-triggered, message-triggered object (TMO). The TMO structure results from adding time-triggered methods and deadline-imposed remote object call mechanisms to the C++/Java object structure.

design time, can appear only in SpMs. 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, the autonomous activation condition (AAC) section.

Basic concurrency constraint. The BCC 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 may execute only if no SpM that accesses the same ODSSs to be accessed by this SvM has an execution time window that will overlap with the execution time window of this SvM. The BCC does not reduce the programming power of TMO.

APIs for time-triggered actions

Although the TMO execution engines built so far completely support TT methods, they support TT executions of object-method-segments only to a limited extent: Object methods are about the only basic schedulable computation units fully supported so far. This constraint does not seriously limit the programming power and flexibility offered to real-time programmers, and it greatly simplifies the job of constructing reliable and efficient execution engines. Supporting TT method-segments just requires construction of object execution engines capable of accurately scheduling finer-grain real-time computation units.

To support TT executions of method-segments in a limited form, a TT method in a high-level programming language may contain the following statements:

```
"at global-time-constant do S" and
"after global-time-constant do S"
```

Global-time-constant must be a real-time instance preceding the completion deadline of the TT method. Such statements can be supported by the execution

```
int NonBlockingSR ( void *ParamPtr, int ParamSize,  tmsp &Timestamp );
int NonBlockingGetResultOfNonBlockingSR ( tmsp Timestamp );
int BlockingGetResultOfNonBlockingSR1 ( tmsp Timestamp,
  microsec ResponsePeriod );
int BlockingGetResultOfNonBlockingSR2 ( tmsp Timestamp, r_tm RTDeadline );
```

Figure 4. When a client calls an SvM via NonBlockingSR, the execution engine records a time stamp into the variable Timestamp. The time stamp uniquely identifies this particular call for the SvM as distinct from other (past or future) calls for the same SvM from this client.

engine in several ways. API components approximating such constructs include "`wait_for (microsec x1)`" and "`wait_until (r_tm t1)`" where `r_tm` denotes the type "real-time."

In the case of the TMO programming scheme, SvMs deal with the relative time domain only; they use only the elapsed intervals since the method was started by an invocation message from an object client. This use is natural since the arrival time of a service request from an object client cannot be predicted by the SvM's designer in general, especially when that designer did not design the client object. The wait_until function should therefore not be used in SvMs.

A provision should also be made for making the AAC section of a TT method 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 or TT method within the same real-time object makes a reservation for future executions of a specific TT method.

An easy-to-use API-style approximation toward the AAC is the class AACclass, of which the constructor registers with the execution engine the parameters such as AAC activation-time, deactivation-time, period between TT executions, earliest start-time of each TT execution, latest start-time of each execution, and GCT for an execution. One more parameter used here indicates whether the constructed object is a permanent AAC or a candidate AAC with a name.

When a TT method is created, there is no symbolic name to be registered with the execution engine but as in the case of an SvM, the set of ODSSs to be accessed by the TT method must be registered with the execution engine.

INTERACTION AMONG REAL-TIME OBJECTS

Any practical real-time object language must support multiple types of service requests. The API for service requests can be structured most naturally using the gate methods described previously, since the gate provides an access path to the server object methods.

Blocking call

After calling an SvM, the client waits until the SvM returns a result message. For this basic type of service request, the following two API functions (which differ only in the type of a return deadline imposed) are most desirable:

```
int BlockingSR1 ( void *ParamPtr, int
  ParamSize, microsec ResponsePeriod );
int BlockingSR2 ( void *ParamPtr, int
  ParamSize, r_tm RTDeadline );
```

BlockingSR1 is used by a client that imposes a deadline of the type "within 20 milliseconds." Any client can make this type of blocking call. On the other hand, BlockingSR2 imposes a deadline of the type "by 10 AM" and thus should ideally be used only by a client operating in the absolute time domain.

If the result does not return to the client by the deadline, then the execution engine for the client sets the result value of the API call to "1," indicating the deadline miss, and then raises an exception signal. How such an exception signal is handled depends on factors such as

- whether the execution engine was set to handle such an exception signal in a generic manner, and
- whether the client program provided an exception handler to be invoked in response to the signal.

The client program can also check the result value; if the value is "1," it can branch to an appropriate exception handler code segment.[12]

Nonblocking call

After calling an SvM, the client can proceed to follow-on steps and then wait for a result message from the SvM. The four API functions shown in Figure 4 support this method. When the client needs to ensure—by execution of one of three GetResult API functions—the arrival of the results returned from the earlier NonBlockingSR for the SvM, the variable Timestamp, containing the time stamp associated with the subject call, must be supplied as a parameter. The time stamp type, tmsp, is usually implemented as the real-time value suffixed by the node ID. When a client makes multiple nonblocking calls for SvMs before executing a GetResult API function, the time stamp unambiguously indicates to the execution engine which nonblocking call is referenced.

Of the three GetResult functions listed in Figure 4, the first one, NonBlockingGetResultOfNonBlockingSR, is of the nonblocking type, which means that if the results have not been returned, the client can immediately proceed to follow-on steps. The other two GetResult functions are of the blocking type, differing

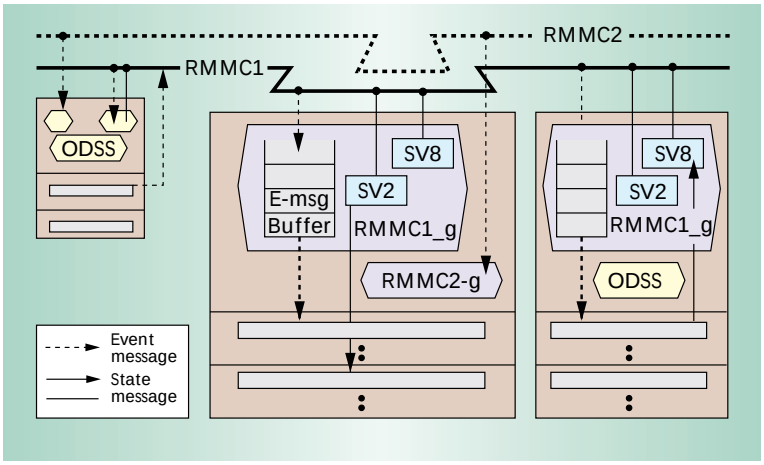


Figure 5. RMMC channels connecting real-time objects. Real-time objects connected to an RMMC can announce via multicasts, read event messages, and globally update and read distributed replicated state message variables.

only in the type of deadlines imposed. If the results have not been returned at the time of executing one of the two BlockingGetResult functions, the execution engine keeps the client in the waiting mode until the arrival of the results is recognized.

A nonblocking call thus creates concurrency between a client method (TT method or SvM) and an SvM in a server object. The concurrency lasts until the execution of the corresponding BlockingGetResult function. Most industrial real-time programming tools will offer support for this kind of concurrency.

Client-transfer call

An SvM in a real-time object can pass a client request to another SvM by using a client-transfer call. The latter SvM can again pass the client request to another SvM. This chaining sequence can repeat until the last SvM in the chain returns the results to the client. The main motivation behind such a client-transfer call stems from BCC, which requires an execution of an SvM to be made only if a sufficiently large time window opens up between executions of TT methods that might conflict with the SvM.

In certain situations a time-consuming SvM may never be executed due to the lack of a wide-enough time window. This problem can be avoided by dividing the SvM into multiple smaller SvMs. A client can then call smaller SvMs in sequence. Calling each smaller SvM incurs the communication overhead of transmitting a request to the smaller SvM and obtaining the results. The substantial reduction of such communication overhead is the motivation behind an arrangement in which the client calls the first SvM and the latter passes the service contract with the client to another SvM. The process repeats until the last SvM of the chain returns the results to the client.

As a part of executing this client-transfer call for an SvM, the execution engine terminates the caller SvM, places a request for execution of the called SvM into the service request queue for the called SvM, and establishes the return connection from the called SvM to the client of the caller SvM just terminated. The port or channel ID through which the initial client of the client-transfer call chain is prepared to receive return results is passed by the execution engine onto the execution support record of the SvM being called. When the system executes the return statement in the called SvM, the results return through the established return connection. Since the initial client of the client-transfer call chain cannot predict from which SvM it will receive returned results, it is implemented to accept results without having to know from which SvM the results originated.

The following API function can facilitate this client-transfer call:

```
int ClientTransferSR ( void
  *OrigParamPtr, int OrigParamSize,
    void *AddedParamPtr, int
    AddedParamSize );
```

Here OrigParam is the parameter structure that the initial client of the client-transfer call chain supplied. The system passes the parameter through the chain. In addition, the caller of each client-transfer call passes an additional parameter structure, AddedParam, which contains intermediate results produced by the caller but not included in OrigParam.

The recipient SvM of a client-transfer call may be an SvM in the same real-time object to which the caller belongs; it can also be an SvM in another real-time object, although it is more often than not an SvM in the same object.

MULTICAST CHANNELS

In addition to the interaction mode based on remote method invocations, distributed real-time objects can use another interaction mode in which messages can be exchanged over logical message channels explicitly specified as data members of involved objects. One of the most advanced types of such channels is called the real-time multicast and memory-replication channel (RMMC), previously called the programmable data field channel. While Figure 3 depicts such data members in abstract forms, Figure 5 depicts RMMCs and real-time object methods that access them in more detailed forms.

For example, access gates for two RMMCs (RMMC1 and RMMC2) can be declared as data members of each of the three remotely cooperating real-time objects (TMO1, TMO2, and TMO3) during the design time. Once TMO1 sends a message over RMMC1, the message will be delivered to the buffer allocated inside the execution engine for each of the three real-time objects. Later during their execution,

certain methods in TMO2 and TMO3 can pick up those messages by sending the requests through their RMMC1 gates to their execution engines. In many applications, this interaction mode leads to better efficiency than the interaction mode based on remote method invocations. An RMMC can be implemented over point-to-point networks as well as over broadcast-enabled bus networks.

The RMMC scheme supports not only conventional event messages but also state messages based on distributed replicated memory semantics. A state message carries information to be stored in a fixed memory location in each receiver corresponding to the ID of the state message.

A state message's ID represents a group of replicated memory units, each capable of holding the information carried in the state message and belonging to a different receiver. A state message producer timestamps the message at message-production time. Each receiver reads the content of its state message memory through a relevant gate at a 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 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 state-varying item, like a car's position.

REAL-TIME MULTICAST API

The API related to RMMCs must include operations both for connecting and disconnecting an RMMC gate (declared in the ODS of a real-time object) to an RMMC. This API can take the form of a class RMMCgateClass, which plays a role similar to that of GateClass in construction of call-paths. As shown in Figure 6, RMMCgateClass provides four basic methods for message communication over an RMMC.

Any method of a real-time object connected to an RMMC can perform an announce operation that results in the delivery of the message to all other real-time objects connected to the RMMC. The official release time is the time at which Msg should become accessible to the methods in all the real-time objects.

A real-time object can perform the receive operation to pick an officially released event message from its buffer inside the execution engine and put it in the memory area Msg. Real-time objects pick event messages one at a time in the order of their official release times.

Any method of a real-time object connected to an RMMC can perform the global update operation SM_update. This operation updates all SM replicas allocated inside execution engines of all the real-time objects (connected to the RMMC) with NewData. During the execution of this SM_update, the execution engine creates a time stamp and attaches it to the SM being replicated over the group of real-time objects.

```
void announce (void *MsgPtr, int MsgSize, r_tm
  Official-release-time, int &Result_code)
int receive (void *MsgPtr, int &MsgSize)
void SM_update (int SM_ID, void *NewDataPtr, r_tm
  Official-release-time, int &Result_code)
int SM_read (int SM_ID, void *NewSMptr, tmsp
  &Timestamp)
```

Figure 6. RMMCgateClass contains as member functions the two basic operations on event messages, announce and receive, and the two basic operations on state messages, SM_update and SM-read.

A real-time object can perform the SM_read operation to read the most recent officially released value contained in its replica of SM into NewSM.

These RMMCs can serve as an alternative mechanism for interaction among distributed real-time objects. They can also complement the remote method call mechanism.

New languages supported by new compilers will continue to emerge in ever more compelling forms, but it will be a while before we have ideal language tools for real-time distributed object programming. Analysis tools are also needed. The analysis tools most highly desired but unavailable at this time are those that would assist the distributed real-time object designer in the process of determining guaranteed response times.

At the minimum, such tools must be capable of making good estimates for worst-case execution times of various object-method segments. An even more powerful tool could provide a useful aid in checking the execution feasibility of real-time objects—checking, for example, whether a group of real-time objects can be loaded onto a network of nodes without introducing the possibility of any timing-constraint violations.

Until ideal language tools arrive, a pragmatic approach to enabling high-level real-time object programming today is to provide abstract APIs. ✸

Acknowledgments

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 program under Grant 99-75053.

References

1. Object Management Group, "Realtime CORBA Joint Revised Submission," OMG Document orbos/99-02-12, Mar. 1999.
2. Java Consortium, "Real-Time Core Extensions for the Java Platform," Specification No. T1-00-01, Rev. 1.0.10, Feb. 2000, http://www.j-consortium.org/.

3. Real-Time for Java Experts Group, "Real-time Specification for Java, Version 0.9.2," 29 Mar. 2000, http://www.rtj.org/public.
4. J. Rumbaugh et al., *The Unified Modeling Language Reference Manual*, Addison Wesley Longman, Reading, Mass., 1999.
5. B. Selic, "Turning Clockwise: Using UML in the Real-Time Domain," *CACM*, Oct. 1999, pp. 46-54.
6. R. Soley, ed., *Object Management Architecture Guide*, John Wiley & Sons, New York, 1995.
7. K.H. Kim, "Object Structures for Real-Time Systems and Simulators," *Computer*, Aug. 1997, pp. 62-70. This and other articles on the TMO scheme are also available from http://dream.eng.uci.edu/TMO/TMO.htm.
8. K.H. Kim, "Real-Time Object-Oriented Distributed Software Engineering and theTMO Scheme," *Int'l J. Software Eng. & Knowledge Eng.*, Apr. 1999, pp. 251-276.
9. H. Kopetz, *Real-Time Systems*, Kluwer Academic, New York, 1997.
10. K.H. Kim and C. Subbaraman, "Principles of Constructing a Timeliness-Guaranteed Kernel and the Time-Triggered Message-Triggered Object Support Mechanism," *Proc. ISORC 98*, IEEE CS Press, Los Alamitos, Calif., 1998, pp. 80-89.
11. K.H. Kim, M. Ishida, and J. Liu, "An Efficient Middleware Architecture Supporting Time-Triggered Message-Triggered Objects and an NT-based Implementation," *Proc. ISORC 99*, IEEE CS Press, Los Alamitos, Calif., 1999, pp. 54-63.
12. K.H. Kim, J. Liu, and M.H. Kim, "Deadline Handling in Real-Time Distributed Objects," *Proc. ISORC 2000*, IEEE CS Press, Los Alamitos, Calif., Mar. 2000, pp. 7-15.

K.H. (Kane) Kim is a professor in the Department of Electrical and Computer Engineering at the University of California, Irvine. He is a recipient of the 1998 IEEE CS Technical Achievement Award for his contributions to the scientific foundation for both real-time fault-tolerant computing and real-time object-oriented distributed computing. He received a PhD from the Computer Science Division of the University of California, Berkeley. Contact him at Khkim@uci.edu.