Proc. ISORC 2005 (8th IEEE CS Int'l Symp. on Object-Oriented Real-Time Distributed Computing),Seattle, May, 2005, pp. 34-43

RMMC Programming Model and Support Execution Engine in the TMO Programming Scheme

K.H. (Kane) Kim, Yuqing Li, Sheng Liu
University of California
Irvine, CA 92697, USA
{ khkim, yuqingl, shengl }@uci.edu

and

Moon H. Kim and Doo-Hyun Kim
Konkuk Univesity
Seoul, Korea
{ mhkim, doohyun }@konkuk.ac.kr

Abstract: While the conventional remote method invocation mechanism has been considered for a long time as the primary approach for facilitating interactions among real-time objects, a multicast support mechanism has been recognized in recent years as an attractive supplement, if not a favored approach, in many applications. One highly promising concrete formulation of a multicast mechanism and an associated programming model is the real-time multicast and memory-replication channel (RMMC). This paper presents an API set designed for facilitating easy use of RMMCs and a middleware subsystem devised to support RMMCs. An experimental study performed to validate the RMMC support facility and check the performance of a TMO-structured multimedia application which uses an RMMC, is also discussed.

Keywords: multicast, real time, RMMC, event message, state message, multimedia, TMO, fairness, streaming

1. Introduction

Structuring real-time distributed computing systems in the form of networks of real-time objects has become a research and development field of steady growth in the past decade [Bol00, Kim97, Kim00, OMG02, OMG04, Sel04]. In addition to the conventional remote method invocation mechanism for facilitating interactions among real-time objects, a multicast support mechanism has been recognized in recent years as an attractive supplement, if not a favored approach, in many applications [Gor04, Kai99, Kim99b, Kim00, Kop97, Raj96]. An example of such application is a high-quality multi-party video-conferencing system. To support those applications, the underlying execution engine must possess capabilities which can overcome many challenging issues such as the assurance of stable real-time performance in both packet delivery and globally synchronous distributed computing, the guarantee of fairness in treating each distributed user site, etc [Bla96, Cel00, KDH02].

One highly promising concrete formulation of a multicast mechanism and an associated programming model that has been developed recently is the *Real-time Multicast and Memory-replication Channel* (RMMC) [Kim99b, Kim00], of which an earlier version was called the HU data field channel [Kim95]. The RMMC scheme facilitates RT publish-subscribe channel in one of the most powerful forms. It supports not only conventional *event messages* but also *state messages* based on distributed replicated memory semantics [Kop97].

The RMMC scheme has been incorporated into a powerful high-level real-time distributed computing (DC) object programming scheme, the *Time-triggered Message-triggered Object* (TMO) scheme [Kim97, Kim00, Kim02]. The TMO, developed by the first co-author and his collaborators, is a syntactically simple and natural but semantically powerful extension of the conventional object structure. It has been devised to facilitate high-level, high-precision, and distributed object-structured real-time programming. To support TMO execution, an efficient middleware architecture named the TMO Support Middleware (TMOSM), which can be adapted to many commercial-off-the-shelf (COTS) platforms, has been established [Gim01, Kim99a, Kim01, KHJ02].

In this paper, a concrete RMMC-based real-time DC object programming model, more specifically, how RMMCs have been integrated into the TMO scheme, is presented. The execution support provided by TMOSM for RMMCs is also discussed. An experiment which involved development of a simple audio-streaming application was conducted. The experiment was an effort for validating the RMMC support capabilities of TMOSM. These results are also presented.

The paper first gives a brief overview of the TMO programming scheme and the TMOSM in Section 2. Then an API set devised for using RMMCs and a middleware subsystem devised to support RMMCs are presented in Section 3. RMMC2SvM, a specialized extension of the standard RMMC, is described in Section 4, which provides a convenient way to support the invocation of service methods via multicast channels. The performance of a TMO-structured multimedia application which uses RMMCs is discussed in Section 5. The paper concludes in Section 6.

2. Background

2.1 TMO

TMO is a natural, syntactically minor, and semantically powerful extension of conventional object structure. As depicted in Figure 1, the basic TMO structure consists of four parts:

ODS-sec: Object-data-store section. This section contains the data-container variables shared between methods of a TMO. Variables are grouped into *ODS segments* (ODSSs) which are the units that can be locked for exclusive use by a TMO method in execution. Scheduling and access control protect mutual exclusion.

EAC-sec: Environment access capability section. These "gate objects" provide efficient call-paths to remote object methods, real-time multicast and memory replication channels (RMMCs), and I/O device interfaces.

SpM-sec: Spontaneous method section. These are *time-triggered methods* that become alive at specified times.

SvM-sec: Service method section. These provide service methods which can be called by other TMOs.

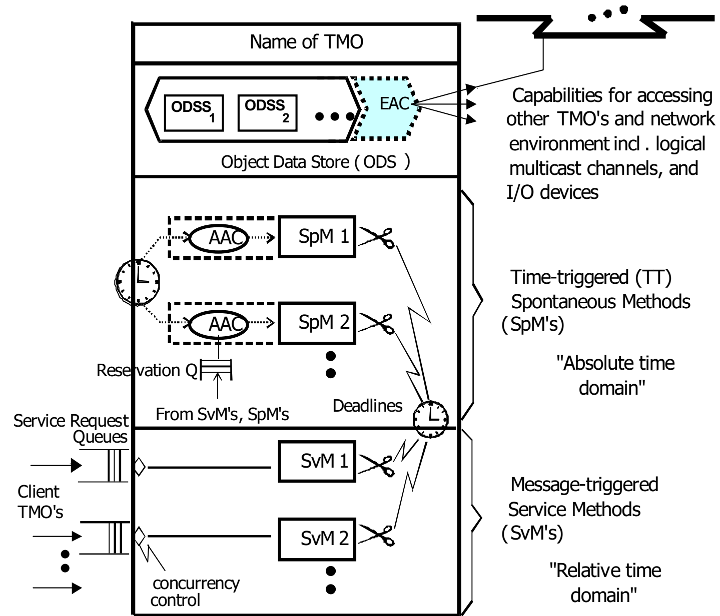

Figure 1. Basic TMO structure (from [Kim97])

Major features are summarized below.

(1) Distributed computing component: The TMO is a distributed computing component and thus TMOs distributed over multiple nodes may interact via remote method calls. To maximize the concurrency in execution of client methods in one node and server methods in the same node or different nodes, client methods are allowed to make non-blocking service requests to service methods. In addition, TMOs can interact by exchange of messages over RMMCs.

(2) Clear separation between two types of methods: The TMO may contain two types of methods, time triggered (TT) methods (*spontaneous methods* or SpMs), which are clearly separated from the conventional *service methods* (SvMs). The SpM executions are triggered when the RT clock reaches time values determined at the design time. On the contrary, SvM executions are triggered by calls from clients that are transmitted by the execution engine in the form of service request messages.

Triggering times for SpMs must be fully specified as constants during the design time. Those RT constants as well as related *guaranteed completion times* (GCTs) of the SpM 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"}.

(3) *Basic concurrency constraint* (BCC): This rule prevents potential conflicts between SpMs and SvMs and reduces the designer's efforts in guaranteeing timely service capabilities of TMOs. Basically, activation of an SvM triggered by a message from an external client is allowed only when potentially conflicting SpM executions are not active. An SvM is allowed to execute only when an execution time-window big enough for the SvM exists and does not overlap with the execution time-window of any SpM that accesses the same ODSS data as the SvM. However, the BCC does not stand in the way of either concurrent SpM executions or concurrent SvM executions.

(4) *Guaranteed completion time* (GCT) of the server (i.e., an SvM of a server TMO) and the *result return deadline* imposed by the client: The TMO incorporates deadlines in the most general form. Basically, for output actions and method completions of a TMO, the designer guarantees and advertises execution time-windows bounded by start times and completion times. In addition, deadlines can be specified in the client's calls for service methods for the return of the service results.

2.2 TMOSM

The TMO Support Middleware (TMOSM) basically consists of a number of virtual machines (VMs), each managing a set of threads and using them to perform certain specialized functions as parts of executing TMOs. See Figure 2. To make VMs co-exist on top of a commodity kernel, TMOSM contains one more component, which can be viewed as the innermost core and is a *super-micro* thread called the WTST (*Watchdog*

Timer & Scheduler Thread). It is a "super-thread" in that it runs at the highest possible priority level. It is also a "micro-thread" in that it manages the scheduling / activation of all VMs which in turn operate other threads in TMOSM. Even those threads created by the node OS kernel before TMOSM starts are executed only if WTST allocates some time-slices to them. Therefore, WTST is in control of the processor and memory resources with the cooperation of the node OS kernel.

WTST leases processor and memory resources to three VMs in a time-sliced and periodic manner. Each VM can be viewed conceptually as being periodically activated to run for a time-slice. Each VM is responsible for a major part of the functionality of TMOSM. Each VM maintains a number of application threads. In fact, whenever WTST assigns a time-slice to a VM, the VM in turn passes the time-slice onto one of the application threads that belong to it. The component in each VM that handles this “time-slice relay” is the application thread scheduler. For example, VM-A has the application-thread-scheduler VM-A-Scheduler. The application thread scheduler is actually executed by WTST. To be more precise, at the beginning of each time-slice, a timer-interrupt results in WTST being awakened. WTST then determines which VM should get this new time-slice. If VM-A is chosen, WTST executes VM-A-Scheduler and as a result, an application thread belonging to VM-A is activated to run for a time-slice as WTST enters into the event-waiting mode.

The set of VMs is fixed at the TMOSM start time. One iteration of the execution of a specified set of VMs is called a *TMOSM cycle*. For example, one TMOSM cycle may be: VCT VMAT VAT VMAT. The following three VMs handle the core functions:

(1) VCT (*VM for Communication Threads*): The application threads maintained by this VM are those dedicated to handling the sending and receiving of middleware messages. Middleware messages are exchanged through the communication network among the middleware instantiations running on different DC nodes to support interaction among TMOs. Therefore, these application threads are called communication threads and denoted as CTs in Figure 2. A communication thread also distributes middleware messages coming through the network to their destination threads, typically belonging to another VM discussed below.

(2) VMAT (*VM for Main Application Threads*): The application threads maintained by this VM are those dedicated to executing methods of TMOs with maximal exploitation of concurrency. Those application threads are called main application threads and denoted as MATs in Figure 2. Normally to each execution of a method of an application TMO is dedicated a main application thread. In principle, TMO method executions may proceed concurrently whenever there are no data conflicts among the method executions. Every time-slice not used by the other VMs is normally given to this VM. In every one of our prototype implementations of TMOSM, the application thread scheduler in VMAT uses a kind of a deadline-driven policy for choosing a main application thread to receive the next time-slice.

(3) VAT (*VM for Auxiliary Threads*): This VM maintains a pool of threads which are called auxiliary threads and denoted as ATs in Figure 2. Some auxiliary threads are designed to be devoted to controlling certain peripherals under orders from TMO methods (executed by main application threads). Others wait for orders for executing certain application program-segments and such orders come from main application threads in execution of TMO methods. Use of this VAT has been motivated partly by the consideration that it should be easier to analyze the temporal predictability of the application computations handled by each VM, i.e., those handled by VMAT and those by VAT, than to analyze the temporal predictability of the application computations when there is no VAT and thus VMAT alone handles the combined set of application computations.

Also, WTST provides the services of checking for any deadline violations and if a violation is found, it provides an exception signal to the user.

We believe that structuring of VMs as periodic VMs is a fundamentally sound approach which leads to easier

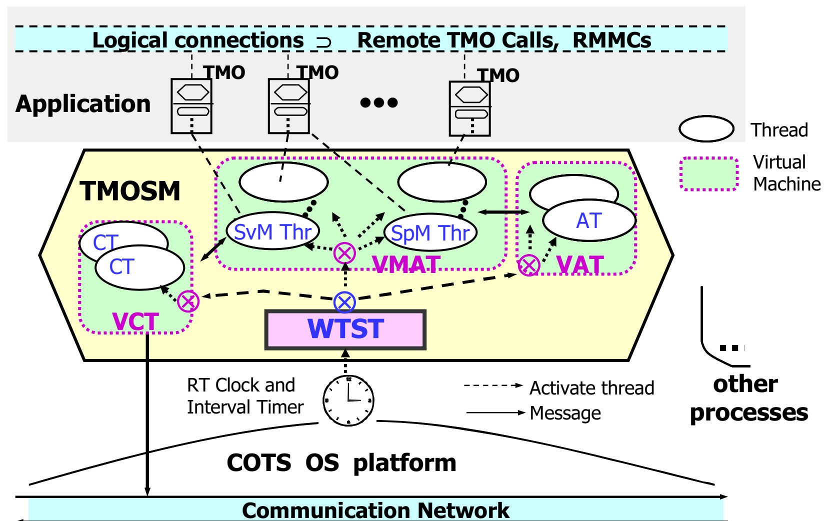

Figure 2. TMOSM architecture

analysis of the worst-cast time behavior of the middleware without incurring any significant performance drawback.

A friendly programming interface wrapping the execution support services of TMOSM has also been developed and named the *TMO Support Library* (TMOSL) [Kim99a, Kim00]. It consists of a number of C++ classes. TMOSL empowers C++ programmers with powerful and natural mechanisms for specification of unique and essential features of TMO programs. Both TMOSM and TMOSL have been used in an undergraduate course on RT DC programming at UCI (http://dream.eng.uci.edu/eecs123/serious. htm) and in a number of research organizations.

3. Real-Time Multicast & Memory-Replication Channel (RMMC)

In the TMO programming model, the RMMC scheme is an alternative to the remote method invocation for facilitating interactions among TMOs. Use of RMMCs tends to lead to better efficiency than the use of traditional remote method invocations does in many applications, especially in the area of distributed multimedia applications which involve frequent delivery of the same data to more than two participants distributed among multiple nodes.

In this section, the discussion of RMMCs focuses on two areas: the APIs and the execution engine support for RMMCs.

3.1 Application programming interfaces (APIs)

In order for methods in a TMO to send and receive messages over RMMCs, the TMO must contain access gates for the RMMCs in its ODS, i.e., as its data members. Figure 3 illustrates a TMO network consisting of three TMOs which interact with each other via two RMMCs, i.e., TMO1 and TMO2 subscribe to RMMC2, and TMO1, TMO2, and TMO3 subscribe to RMMC1. TMO2 contains two *RMMC access gates*, RMMC1_g and RMMC2_g, which are used for accessing RMMC1 and RMMC2, respectively.

The RMMC access gate, or RMMC-gate for short, can be easily defined by inheriting a C++ class, the RMMCgateBaseClass, incorporated in the *TMO Support Library* (TMOSL) which is a programmer-friendly API set for the C++ TMO programming. The RMMCgateBaseClass provides a set of service functions which represent the API set for using RMMCs.

The following considerations have been given in designing the APIs for the RMMC scheme.

(1) Message type:

The RMMC scheme supports both *event messages* and *state messages* [Kim00]. An event message is inserted into a queue in the execution engine which is sorted on the basis of the *official release time* (ORT) [Kim99b, Kim00]. Physically the execution engine consists of all the computing nodes running application TMOs, including the OS kernel and middleware instantiations on these nodes. Therefore, for each RMMC an event message queue is maintained in every computing node running at least “one subscriber”.

The basic unit serving as a subscriber for an RMMC is actually the RMMC-gate. Multiple access gates for the same RMMC may be created in the ODS of a TMO.

A TMO method that accesses the RMMC-gate always reads the first event message from the queue. Each RMMC handles one type of event messages.

On the other hand, a state message is stored in a variable replicated in all the computing nodes running subscriber TMOs, to be exact, TMOs containing relevant RMMC-gates. The state message variable is located by a unique state message ID in every subscriber node (i.e., node running at least one subscriber TMO). In principle, a state message which has arrived and been released can overwrite the old one with the same ID. A TMO method always reads the most recently “released” state message through an RMMC-gate.

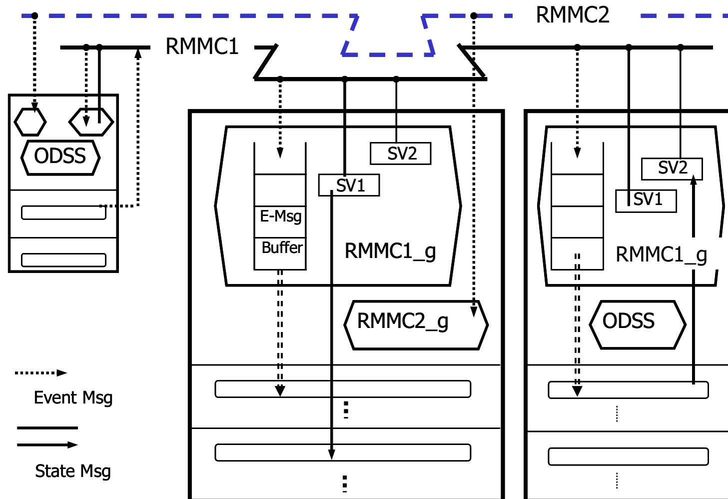

Figure 3. A TMO network connected via two RMMCs

(2) Functionality:

The RMMC scheme should support functionalities such as :

(a) Creation of RMMC-gates;

(b) Activation and deactivation of RMMC-gates; and

(c) Announcing and receiving of event messages over an RMMC and globally updating and reading state messages over an RMMC.

The API set offered by the RMMCgateBaseClass, depicted in Figure 4, includes these functionalities and it will be discussed further below.

(3) Calling convention:

In principle, both blocking and non-blocking calls for acquiring event messages

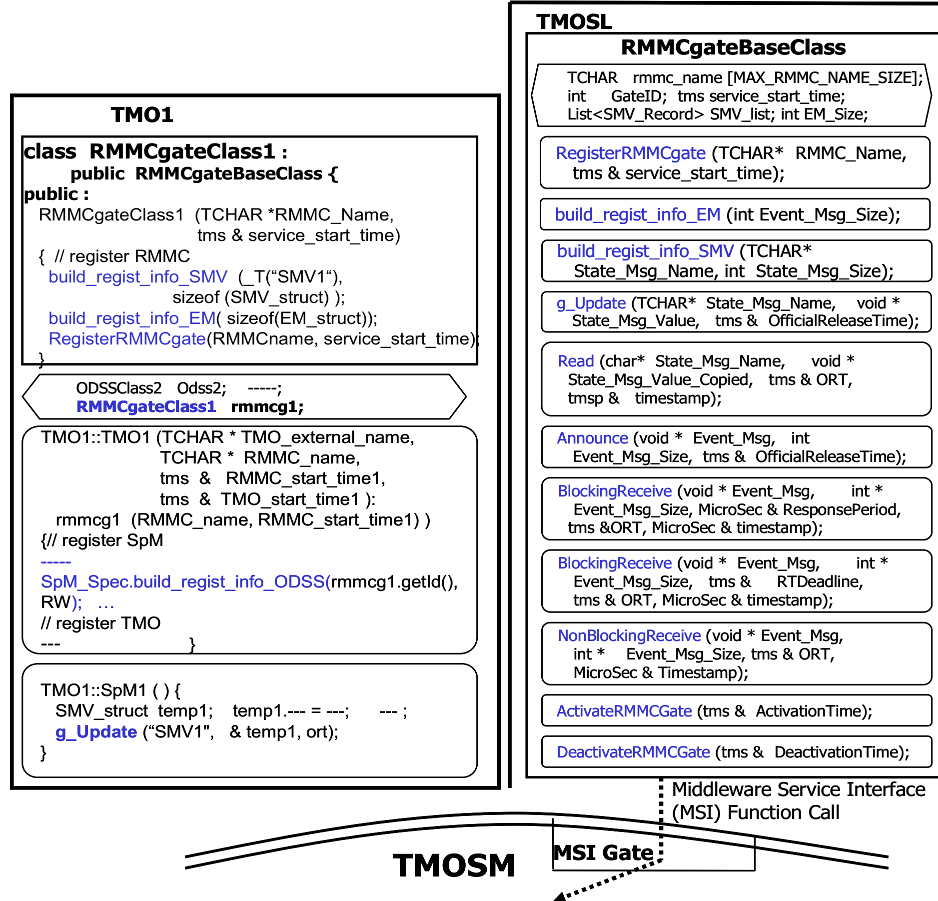

Figure 4. The application programming interface of RMMCgateBaseClass and an example program using RMMC

should be supported in the RMMC scheme. A blocking call to receive an event message puts the calling thread (in other words, a thread assigned to the TMO method which issues such a call) into a waiting queue in the execution engine when no event message is ready and then yields the processor to other ready threads. However, often it is not cost-effective to support blocking calls for sending actions (e.g., announcing an event message). Consider the case where the sender TMO multicasts an event message to five receiver TMOs via an RMMC using a blocking announcement API. Any fault occurring in any of the five subscriber TMOs could potentially put the sender TMO into a stale status.

(4) Timing parameters:

Every function in the defined APIs use some timing specification parameters. As a part of the input parameters, those timing parameters are passed into the execution engine when the function is invoked, and the execution engine takes proper actions to honor those timing specifications. For example, an official release time can be attached to the announcement of an event message, which can be used to ensure that the newly announced information becomes accessible to all receiver TMOs simultaneously. Also, a deadline can be specified as an input parameter in a blocking call for receiving an event message so that the caller can safely return to life when the expected event message gets lost due to a fault in the communication network or the sender TMO.

Each application-specific RMMC may contain a unique set of state message variables. Such decision is reflected in defining an application-specific RMMCgate_class by inheriting the RMMCgateBaseClass in the TMOSL. In other words, the definition of an RMMC-gate shows what types of event messages and state messages pass through the gate.

For one RMMC channel, the size of its event message is fixed. The size information is indicated in a data packet sent to the execution engine, called the registration packet, during the construction of an associated RMMC-gate. The size information can be conveniently entered into the registration packet by use of the API build_regist_info_EM (int Event_Msg_Size). The structure of an event message is known only at the application level, not inside the execution engine.

Each RMMC may handle several types of state messages. Instances of state message variables must be created in each RMMC-gate. The symbol string identifying a state message variable and the size of the state message type are indicated in the registration packet sent to the execution engine during the construction of the associated RMMC-gate. The symbol string and the size information can be conveniently entered into the registration packet by use of build_regist_info_SMV (TCHAR* State_Msg_Name, int State_Msg_Size). A pair of methods, g_Update() and Read(), are used to update and read state message variables.

Using the official release time (ORT) as an explicit parameter is an option for application. If ORT is not given by the sender subscriber, the execution engine will deliver messages to subscriber RMMC-gates as soon as the message is available.

When g_Update() or Read() is called to send a message through an RMMC-gate, the execution engine records a timestamp. The "send-timestamp" can be

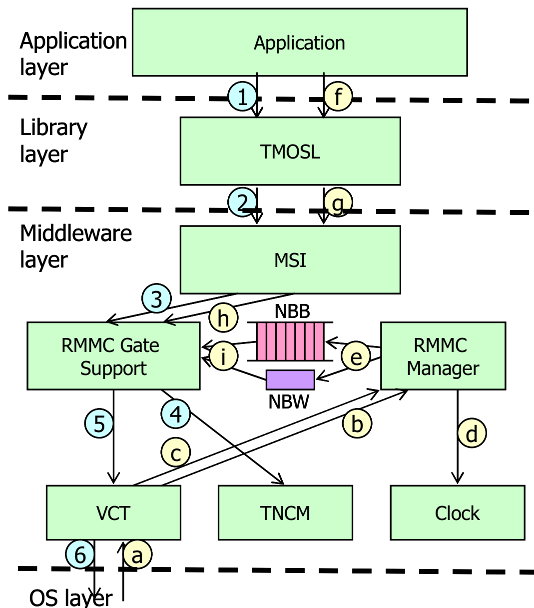

Figure 5. The interactions between the RMMC manager and other middleware components

examined by a receiver of an RMMC message for various application purposes.

The RMMC scheme supports dynamic activation and deactivation of RMMC-gates. DeactivateRMMCgate () can be used to disconnect the associated RMMC-gate from the associated RMMC by providing deactivation time as an input parameter. When DeactivateRMMCGate () is invoked and the global time reaches the specified deactivation time, the execution engine deactivates the associated RMMC-gate and then TMO method executions can still access the RMMC-gate but the gate does not maintain the connection to the RMMC and thus can neither send nor receive event and/or state messages through the RMMC. All incoming messages with ORTs later than the deactivation time cannot be read via TMOSL APIs (e.g., BlockingReceive()). Outgoing messages can still be delivered if the corresponding TMOSL APIs (e.g., Announce()) are invoked before the deactivation time. RMMC gates that were deactivated earlier can be re-activated by calling ActivateRMMCgate (). These deactivation and activation mechanisms can be exploited in application design for conservative use of execution resources.

Therefore, the APIs discussed in this section were devised to make the programming with RMMCs simple and easy while providing all the essential capabilities for high-precision real-time processing and decision making.

3.2 Execution engine support for RMMCs

In designing the middleware support for RMMCs, the performance was the driving factor.

3.2.1 RMMC manager and RMMC gate support

Two middleware components shown in Figure 5, called the RMMC manager and RMMC gate support, contain all the data structures and methods related to RMMC management.

The RMMC gate support component is directly related to the handling of user requests made through TMOSL APIs. The RMMC managers residing at different participating sites maintain the list of subscribers for each RMMC among others.

The message exchanged among distributed cohorts of the RMMC manager can be categorized into five different types.

a) RMMC_CREAT_NEW_GATE_MSG: An RMMC manager which has received a new_gate_creation request from a local TMO broadcasts this message (to all DC nodes).

b) RMMC_ACTIVATE_GATE_MSG: An RMMC manager which has received a gate_activation request from a local TMO multicasts this message (to all RMMC managers).

c) RMMC_DISACTIVATE_GATE_MSG: An RMMC manager which has received a gate_deactivation request from a local TMO multicasts this message (to all RMMC managers).

d) RMMC_EVENT_MSG: An RMMC gate support component which has received an event message announcement request from a local TMO multicasts this message (to all RMMC managers).

e) RMMC_STATE_MSG: An RMMC gate support component which has received a request for globally updating a state message variable from a local TMO multicasts this message (to all RMMC managers).

3.2.2 Relationship between the RMMC components and other middleware components

In order to perform a specific RMMC support task (e.g., announce an event message), the RMMC-related components in a TMOSM instantiation rely on remote cohorts as well as other local middleware components. Figure 5 shows the interactions between the RMMC-related components and other middleware components in the same TMOSM instantiation. The sequence of numerically labeled steps in the figure describes a typical procedure for sending a message via an RMMC, and the sequence of alphabetically labeled steps describes the receiving procedure.

In Step 1, a sending API in RMMCgateBaseClass is invoked, e.g., Announce(). The related procedure defined in MSI (middleware service interface) is invoked in Step 2. MSI is a mediator component between TMOSL and TMOSM. That is, all requests to the middleware services from the library layer must go through MSI before they are forwarded to the corresponding middleware components. In the case of the RMMC scheme, each method (e.g., g_Update, Announce, etc) supported in the RMMCgateBaseClass in TMOSL, must invoke a corresponding method in MSI

in order to access the RMMC support components in the middleware.

Step 3 in Figure 5 is to invoke the corresponding method defined in the RMMC gate support component. This component consults with TNCM (TMO Network Configuration Manager) regarding the destination IP address in Step 4.

Typically, the RMMC gate support component inserts the message into RMMCOutgoingMsgNBB. A *Non-Blocking Buffer* (NBB) is a special data type which allows a single writer and a single reader to access the buffer in a non-blocking manner [Kim03, Kim05]. It is a major extension of the *Non-Blocking Writer* (NBW), a significant innovation by Kopetz which is applicable for communicating state messages only [Kop97]. In Step 6, the message in the NBB is sent out by VCT, the virtual machine for communication threads reviewed in Section 2.2, to other computing nodes.

The RMMC manager registers a receiving handler into VCT so that when the relevant messages (five types of RMMC messages) arrive, the receiving handler can be invoked by VCT. Steps a and b in Figure 5 describe these procedures. When VCT receives an event message or a state message, it invokes the message handler registered by RMMC manager to save this message into an unreleased event/state message list.

The RMMC components also rely on VCT to release the received event/state messages. In Step c, VCT checks whether a corresponding message saved in the unreleased event/state message lists can be released by comparing their ORTs against the current global time. The Clock component is accessed by the RMMC manager to retrieve the current global time in Step d.

In Step e, a message is released if its ORT has passed. If the released message is an event message, it is moved into EventMsgNBBs. If a state message is released, it will be copied into the StateMsgNBW (Non-Blocking Writer) buffer of this state variable.

When the application attempts to receive/read an event/state messages, it needs to invoke RMMC related APIs defined in RMMCgateBaseClass in Step f. The next step is to invoke the corresponding procedure defined in MSI, which in turn invokes the corresponding procedures in the RMMC gate support component in Step h. In the last Step i, the RMMC gate support component gets an event or state message from EventMsgNBBs or StateMsgNBW and returns it back to the application.

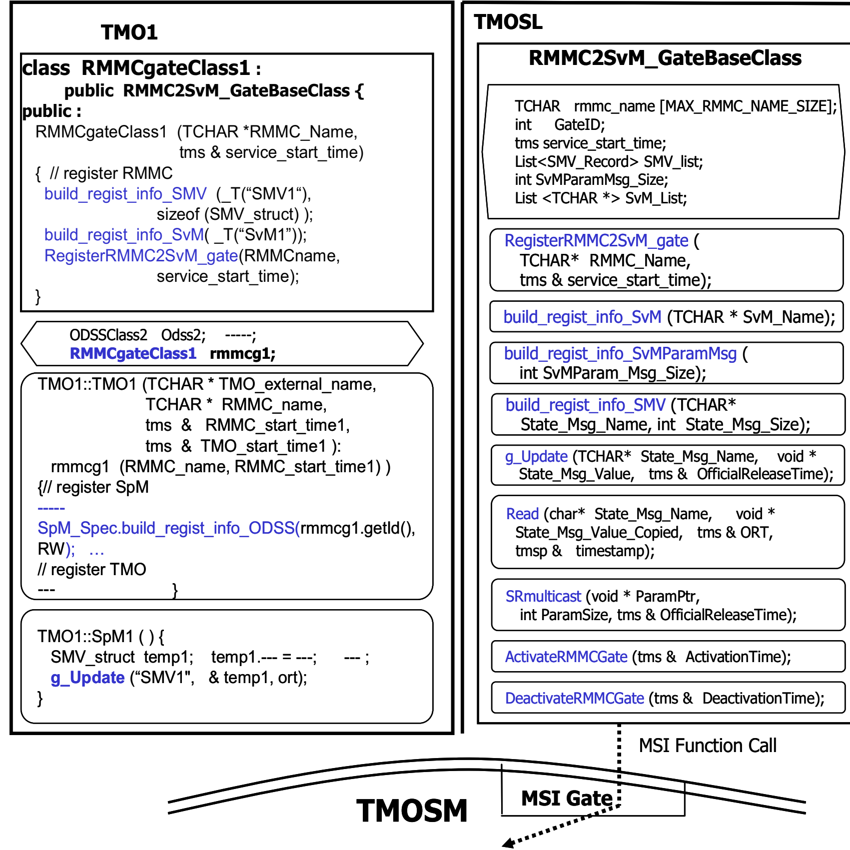

Figure 6. The API set provided by RMMC2SvM_GateBaseClass and an example program using RMMC2SvM

4. RMMC2SvM

RMMC2SvM is another multicast scheme facilitating interactions among multiple TMOs, which enables the invocation of service methods via a multicast channel.

4.1 Application programming interfaces (APIs)

A C++ class, RMMC2SvM_GateBaseClass, has been incorporated into TMOSL to enable the triggering of the executions of SvMs via RMMC2SvM channels. The API set provided by this class is depicted in Figure 6.

Two distinctive APIs supported by the above class are build_register_info_SvM () and SRmulticast (). By calling build_regiser_ info_SvM(), an SvM can be bound to an RMMC2SvM and will be triggered by *SvM parameter messages* afterwards. SRmulticast () can be used by the announcing subscriber. The SvM parameter message sent over the RMMC2SvM creates the effect of calling an SvM which has been bound to each subscriber RMMC2SvM-gate including the announcing subscriber. Only one-way call for such an SvM is allowed.

The support for state messages in RMMC2SvM is identical to the original RMMC's.

4.2 Execution engine support for RMMC2SvM

In order to support RMMC2SvM, a new type of RMMC messages are exchanged among RMMC-related middleware components in different computing nodes.

a) RMMC_SVM_PARAM_MSG: An RMMC gate support component which receives a request from a local TMO for the announcement of an SvM parameter message multicasts this message (to all RMMC managers). This message is interpreted by the execution engine as an one-way call for a certain SvM in every subscriber

An RMMC_SVM_PARAM_MSG message is released when the associated ORT is reached. The message is converted into a remote invocation request to trigger all registered SvMs. Such SvM executions are done in manners not violating the BCC. When an RMMC connection between a remote client and an SvM is desired, the use of RMMC2SvM can lead to an improved response time at the application level over the case where the standard RMMC is used.

5. Multi-Media TMO

In order to evaluate the performance of RMMC based multimedia applications, an audio-streaming experiment involving two different communication mechanisms was conducted. The application program was structured as a TMO network [Kim02, KDH02] and the standard RMMC mechanism was used for transmission of audio-packets from the sender node to the receiver node.

5.1 Application scenario and configuration

Two computing nodes were used in this experiment. The first node is the sender, which is equipped with a microphone. Its job is to obtain audio-packets from the microphone and send them to the receiver node. The second node works as the receiver, which receives audio-packets through LAN and plays them out through speakers.

The sender node is equipped with a P4 2.4GHz processor and 1GB RAM, while the receiver node has a P4 2.6GHz processor and 1GB RAM. They are connected via an 100MB/s LAN.

5.2 Application design and implementation

In this experiment, the sender TMO and the receiver TMO share an RMMC channel. Each TMO has an SpM. In both SpMs, the iteration period is 27ms.

In the sender node, the microphone captures an audio packet every 27ms. In the initialization phase of audio capturing, three application-workspace-buffers

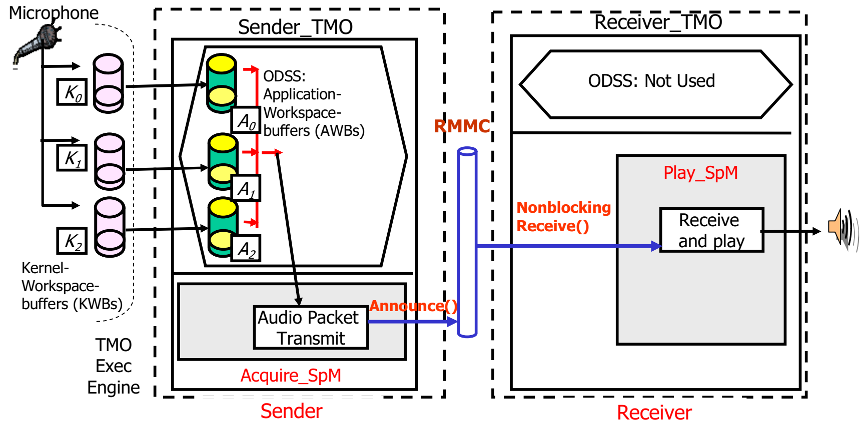

Figure 7. Audio-streaming through the standard RMMC

(AWBs) are registered into the OS kernel. For each AWB, a corresponding kernel-workspace-buffer (KWB) with the same size is allocated in the kernel. When the capturing starts, the audio device driver puts audio packets into KWBs sequentially as indicated in Figure 7, i.e., 0, 1, 2, 0, 1, 2...

To obtain an audio packet from a KWB, a Win32 API for reading a packet must include as a parameter an AWB. For example, if the audio packet in the KWB 0 is to be obtained, the corresponding AWB 0 should be used as a parameter. If the KWB is full, OS kernel copies the audio packet into the AWB. If the audio packet is not fully assembled in the KWB yet, an exception is returned.

Acquire_SpM, which runs periodically at the interval of 27 ms, obtains an audio packet from KWB in every run. In this case, to obtain the audio packet in a correct sequence, recording the buffer ID that was used successfully to get an audio packet in the last run of Acquire_SpM is important. For example, if AWB 2 was used successfully in the last run, then the next AWB that should be used is AWB 0. The reason is that, if the KWB 2 is the latest filled buffer in the kernel, then the kernel will put next audio packets starting from KWB 0. Therefore, to get the audio packet in KWB 0, AWB 0 should be provided in this run.

As described in Figure 7, the Acquire_SpM of the sender captures an audio-packet every 27ms. Then, it calls RMMC::Announce() to push the frame into the RMMC channel. At the receiving side, the Play_SpM checks the RMMC channel by calling RMMC::NonblockingReceive() every 27ms to obtain an audio-packet. Then Play_SpM plays it through the speakers.

5.3 Measurement

The following quantities were measured.

- *Latency*

The time right after one audio packet is acquired from the microphone is recorded as *timestamp1* while the time right after the audio packet is played is recorded

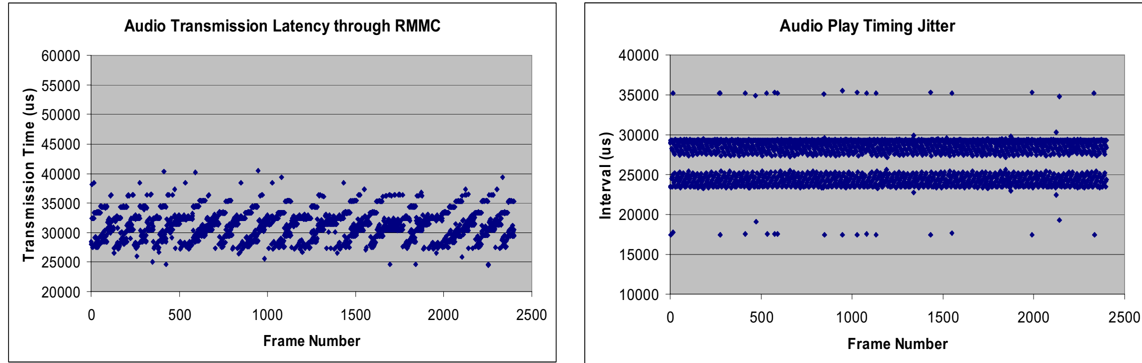

Figure 8. Audio transmission latency and play timing jitter

as *timestamp2*. The duration (*timestamp2 – timestamp1*) is called a latency.

- *Play interval*

The time right after the N-th audio packet is played is recorded as *play_timestampN*. The time right after the next audio packet is played is recorded as *play_timestampN+1*. The duration (*play_timestampN+1 – play_timestampN*) is called a play interval.

- *Jitter* = Maximum play interval – Minimum play interval

As indicated in Figure 8, the latency ranges from 24ms to 41ms and the jitter is about 17ms. These performance data are somewhat better than those observed with another variation of this application where SvM calls instead of an RMMC were used to transmit audio packets across the network.

6. Conclusions

This paper presented one highly promising concrete formulation of a multicast mechanism and an associated programming model, the *Real-time Multicast and Memory-replication Channel* (RMMC) scheme. The RMMC scheme facilitates RT publish-subscribe channel in one of the most powerful forms. It has been incorporated into a powerful high-level real-time distributed computing (DC) object programming scheme, the TMO scheme. The approach adopted TMOSM for supporting RMMCs has been found to be effective.

Our experiences indicate that TMO programmer can get easily addicted to the use of RMMCs in multimedia applications. The first co-author has recently asked research assistants to produce some TMO demonstration programs. It turns out that in most cases, the assistants did not use SvM calls and used RMMCs only to realize real-time distributed computing. From this, one can conjecture that perhaps the one-way service request mechanism is more essential than the two-way service request mechanism since the RMMC event message can be viewed as a generalized case of the one-way service request mechanism. However, much further research is needed in the area of optimal implementation of the middleware support for RMMCs and optimal uses of RMMCs in multimedia applications and others.

Acknowledgment: The research reported here is supported in part by the NSF under Grant Numbers 02-04050 (NGS) and 03-26606 (ITR), and in part by the NSF under Cooperative Agreement ANI-0225642 to UCSD for "The OptIPuter". No part of this paper represents the views and opinions of any of the sponsors mentioned above.

References

[Bla96] Blakowski, G., and Steinmetz, R., "A media synchronization survey: reference model, specification, and case studies", *IEEE Journal of selected areas in communications*, Vol. 14, No. 1, Jan. 1996, pp. 5–35.

[Bol00] Bollella, G., and Gosling, J., "The Real-Time Specification for Java", IEEE *Computer*, June, 2000, pp. 47-54.

[Cel00] Celentano, A., and Gaggi, O., "A synchronization model for hypermedia documents navigation," *Proc. 2000 ACM Symp. on Applied Computing*, Como, Italy, 2000.

[Gim01] Gimenez, G., and Kim, K.H., "A Windows CE Implementation of a Middleware Architecture Supporting Time-Triggered Message Triggered Objects", *Proc. Compsac '01 (IEEE CS Computer Software & Applications Conf.)*, Chicago, IL, Oct. 2001.

[Gor04] Pradeep Gore, Pyarali, I., Gill, C.D., and Schmidt, D.C., "The design and performance of a real-time notification service", *Proc. RTAS '04 (IEEE 10th CS Int'l Symp. on Real-Time and Embedded Technology and Applications)*, May 2004, pp:112-120.

[Kai99] Kaiser, J. and Mock, M., "Implementing the real-time publisher/subscriber model on the controller area network (CAN) ", *Proc. ISORC '99 (IEEE 2th CS Int'l Symp. on Object-Oriented Real-time Distributed Computing)*, May 1999, pp:172-181.

[KDH02] Kim, Doo-Hyun, Kim, K.H., Liu, Sheng, and Kim, J.H., "A TMO-based Approach to Tolerance of Transmission Jitters in Tele-Audio Service", *Int'l*

Journal of Computer System Science & Engineering, 2002, Vol. 17, No. 6, pp. 325-333.

[Kim95] 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. ISADS '95 (IEEE CS 2nd Int'l Symp. on Autonomous Decentralized Systems)*, Phoenix, AZ, April. 1995, pp.305-312.

[Kim97] Kim, K.H., "Object Structures for Real-Time Systems and Simulators", *IEEE Computer*, Vol. 30, No.8, August 1997, pp. 62-70.

[Kim99a] Kim, K.H. Ishida, Masaki, and Liu, Juqiang, "An Efficient Middleware Architecture Supporting Time-Triggered Message-Triggered Objects and an NT-based Implementation", *Proc. 2nd IEEE CS Int'l Symp. on Object-Oriented Real-time Distributed Computing (ISORC '99)*, St. Malo, France, May, 1999, pp.54-63.

[Kim99b] Kim, K.H., "Group Communication in Real-Time Computing Systems: Issues and Directions", *Proc. FTDCS '99 (7th IEEE Workshop on Future Trends of Distributed Computing Systems)*, Cape Town, South Africa, Dec. 20 -22, 1999, pp.252-258.

[Kim00] Kim, K.H., "APIs for Real-Time Distributed Object Programming", *IEEE Computer*, June 2000, pp.72-80.

[Kim01] Kim, K.H., Liu, J.Q., Miyazaki, H., and Shokri, E.H., "TMOES: A CORBA Service Middleware Enabling High-Level Real-Time Object Programming", *Proc. ISADS 2001 (IEEE CS 5th Int'l Symp. on Autonomous Decentralized Systems)*, Dallas, Mar. 2001, pp. 327-335.

[Kim02] Kim, K.H., "Commanding and Reactive Control of Peripherals in the TMO Programming Scheme", *Proc. ISORC '02* (*5th IEEE CS Int'l Symp. on Object-Oriented Real-time Distributed Computing*), Crystal City, VA, April 2002, pp.448-456.

[Kim03] Kim, K.H., "Basic Program Structures for Avoiding Priority Inversions", *Proc. ISORC 2003 (IEEE CS 6th Int'l Symp. on Object-oriented Real-time distributed Computing)*, Hakodate, Japan, May 2003, pp. 26-34.

[Kim05] Kim, K.H., "A Non-Blocking Buffer Mechanism for Real-Time Event Message Communication", to appear in *Real-Time Systems - The International Journal of Time-Critical Computing Systems*.

[KHJ02] Kim, H.J, et al., "TMO-Linux: A linux-based Real-time Operating Systems Supporting Execution of TMOs", *Proc. ISORC '02 (IEEE 5th CS Int'l Symp. on Object-Oriented Real-time Distributed Computing)*, Washington D.C., April 2002, pp. 288-294.

[Kop97] Hermann Kopetz, '*Real-Time Systems: Design Principles for Distributed Embedded Applications*', Kluwer Academic Publishers, 1997.

[OMG02] Object Management Group, "Chapter 24. Real-time CORBA", in *CORBA Specification, Version 2.6.1*, http://www.omg.org/cgi-bin/doc?formal/01-12-35, May, 2002.

[OMG04] Object Management Group, "Chapter 23. Fault-Tolerant CORBA", in *CORBA Specification, Version 3.0.3*, http://www.omg.org/cgi-bin/doc?formal/04-03-21, March, 2004.

[Raj96] Rajkumar, R. and Gagliardi, M., "High availability in the real-time publisher/subscriber inter-process communication model", *IEEE 17th Symp. On Real-Time Systems*, Dec.1996, pp:136-141.

[Sel04] Selic, Bran, "Specifying and Enforcing Software Architectures", Keynote at the Workshop on Software Architecture Description & UML, held in conjunction with the 7th Int'l Conf. on UML Modeling Languages and Applications, Oct. 11-15, 2004, Lisbon, Portugal (available from http://se2c.uni.lu/tiki/tiki-index.php?page=UML2004Program).