A FEASIBILITY EVALUATION OF A TMO-BASED APPROACH TO FINE-GRAINED MULTIMEDIA SYNCHRONIZATION

K. H. (Kane) Kim
University of California
Irvine, CA, USA
http://dream.eng.uci.edu

Doo-Hyun Kim
ETRI
Dae-Jeon, Korea
doohyun@etri.re.kr

Sung-Jin Kim
University of California
Irvine, CA, USA
sungjink@uci.edu

ABSTRACT: We consider it important to provide tools enabling the application system developers to specify and effect desirable temporal relationships among multimedia data elements, including both inter-stream and intra-stream synchronization requirements, relatively easily. The tools enabling easy application of the global time based coordination principle are particularly desirable. In this paper we report the results of an initial feasibility evaluation of such an approach and supporting tool.

KEYWORDS: real time, global time, multimedia, TMO, time triggered, message, object, middleware, distributed, programming, synchronization.

INTRODUCTION

Multimedia applications are steadily growing in both variety and functional complexity. One of the key issues in multimedia processing, especially, in distributed processing of multimedia data streams on a network of computing platforms, is still the accurate maintenance of the temporal relationship among data elements of various media types [Bla96, KSK02]. Moreover, the amount of efforts consumed by the designer and the programmer for specification of such temporal relationship and programming of the processing steps to maintain the relationship among the processed data has become a major issue confronting the multimedia software technology research community.

A multimedia data stream transferred from a source node to one or more sink nodes consists of consecutive logical data units (LDUs). In the case of an audio stream, LDUs are periodically observed individual samples or blocks of such samples transferred together from a source to a sink(s). Similarly, in the case of a video data stream, one LDU may typically correspond to a single video frame. The data elements in a stream must be presented at a sink node in manners exhibiting the same temporal relationship that existed among the data elements when they were captured at the source node. Timing the movement or play of each data element to maintain a desirable temporal relationship among the data elements in a single data stream is called the intra-stream synchronization.

The temporal relationship maintenance must also be applied to related data streams. One of the most commonly encountered situations where such requirement is evident, is where the simultaneous playback of audio and video must be done with the so-called "lip synchronization". If both media are not played in good synchronization, the result will not be accepted to be of high quality. In general, the inter-stream synchronization involves relationships among the data of various kinds of media including pointers, graphics, images, animations, text, audio, and video. In the cases where multiple senders and multiple receivers work simultaneously such as in multi-party non-moderated video-conferencing applications, inter-stream synchronization requirements appear in highly challenging forms.

A fundamental approach that holds great promises in this area but has been minimally exploited by the research community is the global time based coordination of distributed actions [Kop90, Kop97, Aky96]. A part of the reason for slow exploitation of this promising approach is due to the lack of cheap facilities for providing a good-quality global time base. Recent maturing of the GPS receiver technology as well as emergence of hardware and software mechanisms for high-quality synchronization of the real-time clocks in the computing platforms networked to support cooperative distributed computing [Kop97], have eliminated that part of the reason.

Still, it is important to provide tools enabling the application system developers to specify and effect desirable temporal relationships among multimedia data elements, including both inter-stream and intra-stream synchronization requirements, relatively easily. The tools enabling easy application of the global time based coordination principle are particularly desirable. In this paper we report the results of an initial feasibility evaluation of such an approach and supporting tool.

The tool has been devised to support easy practice of object-oriented (OO) real-time (RT) distributed programming. OO RT distributed computing is now a rapidly growing branch of computer science and
engineering [Bol00, IEE00, ISO, OMG00a, OMG00b, WOR]. Its growth is fueled by the strong needs present in industry for the RT programming methods and tools which will bring about orders-of-magnitude improvement over the traditional RT programming practiced with low-level programming languages and styles.

Since early 1990's the first co-author and his collaborators have been establishing a new-generation OO RT programming scheme called the time-triggered message-triggered object (TMO) programming scheme [Kim94, Kim97, Kim99a, Kim00]. The support tools for the TMO scheme can be based on well-established OO programming languages such as C++, C#, and Java and on ubiquitous commercial real-time OS kernels or even on the Microsoft Windows family of OS kernels. Being a high-level programming approach, the TMO scheme offers the following benefits to complex distributed application designers:

(TB1) The TMO model is a natural and syntactically small extension of the conventional object model(s) such that typical OO programmers can adopt it with relatively small efforts.

(TB2) The TMO scheme enables RT programming and distributed programming in a highly abstract and yet high-precision form, relieving the programmer of the burden of dealing with underlying OS services and network protocols and allowing the programmer to be focused on essential design activities.

(TB3) The scheme enables systematic design-time guaranteeing of timely service capabilities of objects.

An integral component of the TMO programming scheme is the use of a global time base. The scheme enables the programmer to design global time based coordination of distributed actions very easily.

As an initial feasibility study, we adopted a simple video-data streaming program written by the research staff at ETRI, Korea as a baseline program. This program is based only on the API associated with the Microsoft Window OS family. It is a real-time distributed computing program which is written in a prevalent low-level concurrent programming style. It captures video scenes by use of a camera attached to the source node and plays them at the screen of the sink node. We produced a TMO program which was much easier to understand, analyze, optimize, and expand. The programming approach is discussed and some measured performance data of the TMO program are also presented. The results of this feasibility study are quite positive.

In the next section, an overview of the TMO programming scheme is provided to make the paper reasonably self-contained. Then in the section that follows, a single-node version of the video-data streaming program written by ETRI staff is first described and this is followed by the discussion on an equivalent TMO program. Thereafter, the two-node version of the video-data streaming program by ETRI staff which uses the RTP protocol [Sch96] and the equivalent TMO program are discussed.

**AN OVERVIEW OF THE TMO SCHEME**

The time-triggered message-triggered object (TMO) scheme was devised to support economical reliable design and implementation of RT systems [Kim94, Kim97, Kim99a, Kim00, Kim02]. The TMO programming scheme is a general-style component programming scheme and supports design of all types of components including distributable hard-RT objects and distributable non-RT objects within one general structure.

TMOs are devised to contain only high-level intuitive and yet precise expressions of timing requirements. No specification of timing requirements in (indirect) terms other than start-windows and completion deadlines for program units (e.g., object methods) and time-windows for output actions is required. For example, priorities are attributes often attached by the OS to low-level program abstractions such as threads and they are not natural expressions of timing requirements.

At the same time the TMO scheme is aimed for enabling a great reduction of the designer's efforts in guaranteeing timely service capabilities of distributed computing application systems. It has been formulated from the beginning with the objective of enabling design-time guaranteeing of timely actions. The TMO incorporates several rules for execution of its components that make the analysis of the worst-case time behavior of TMOs to be systematic and relatively easy while not reducing the programming power in any way.

**TMO structure and design paradigms**

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

- **ODS-sec** = object-data-store section: list of object-data-store segments (ODSS's);
- **EAC-sec** = environment access-capability section: list of gate objects providing efficient call-paths to remote object methods, logical communication channels, and I/O device interfaces;
- **SpM-sec** = spontaneous-method section: list of spontaneous methods;
- **SvM-sec** = service-method section.

Major features are summarized below.

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

(b) Clear separation between two types of methods:

The TMO may contain two types of methods, time-triggered (TT-) methods (also called the spontaneous methods or SpMs), which are clearly separated from the conventional service methods (SvMs). The SpM executions are triggered upon reaching of the real-time clock at specific values determined at the design time whereas the SvM executions are triggered by service request messages from clients. Moreover, actions to be taken at real times which can be determined at the design time can appear only in SpMs.

(c) Basic concurrency constraint (BCC):

This rule prevents potential conflicts between SpMs and SvMs and reduces the designer's efforts in guaranteeing timely service capabilities of TMOs. Basically, activation of an SvM triggered by a message from an external client is allowed only when potentially conflicting SpM executions are not in place. An SvM is allowed to execute only when an execution time-window big enough for the SvM that does not overlap with the execution time-window of any SpM which accesses the same ODSSs to be accessed by the SvM, opens up. However, the BCC does not stand in the way of either concurrent SpM executions or concurrent SvM executions.

(d) Guaranteed completion time and deadline:

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.

Triggering times for SpMs must be fully specified as constants during the design time. Those real-time constants appear in the first clause of an SpM specification called the autonomous activation condition (AAC) section. An example of an AAC is

\[
\text{"for } t = \text{from } 10\text{am to } 10:50\text{am every } 30\text{min start-during } (t, t+5\text{min}) \text{ finish-by } t+10\text{min}"
\]

which has the same effect as

\[
\{\text{"start-during } (10\text{am}, 10:05\text{am}) \\text{ finish-by } 10:10\text{am},
\text{"start-during } (10:30\text{am}, 10:35\text{am}) \\text{ finish-by } 10:40\text{am} \}
\]

An underlying design philosophy of the TMO scheme is that an RT computer system will always take the form of a network of TMOs, which may be produced in a top-down multi-step fashion [Kim97]. The designer of each TMO provides a guarantee of timely service capabilities of the object. The designer does so by indicating the guaranteed execution time-window for every output produced by each SvM as well as by each SpM executed on requests from the SvM and the guaranteed completion time (GCT) for the SvM in the specification of the SvM. Such specification of each SvM is advertised to the designers of potential client objects. Before determining the time-window specification, the server object designer must convince himself/herself that with the object execution engine (a composition of hardware, node OS, and middleware) available, the server object can be implemented to always execute the SvM such that the output action is performed within the time-window. The BCC contributes to major reduction of these burdens imposed on the designer.
Middleware supporting OO RT program

A cost-effective way to support execution of OO RT distributed programs is to realize an execution engine by developing middleware running on well established commercial software / hardware platforms. An efficient middleware architecture, named the TMO Support Middleware (TMOSM), has been developed [Kim99b, Kim02]. TMOSM uses well established services of commercial OSs, e.g., process and thread support services, short-term scheduling services, and low-level communication protocols, in a manner transparent to the application TMO programmer. Prototype implementations based on several most popular OS kernels such as Windows NT/XP, Windows CE, and Linux exist [KHK02, Kim99b, Kim01, Sho98]. Our experiences indicate that even this middleware extension of a general-purpose OS (Windows NT/XP) can accurately enact the time-window for activating a method as small as 10ms and the method completion deadline as short as 3ms unless a device driver not preemptible for an excessively long time gets involved. Prototype implementations that use inter-object communication facilities rather than TCP or UDP, e.g., CORBA ORB and Microsoft COM, exist also [Kim01].

A friendly application programming interface (API) wrapping the services of TMOSM has also been developed and named the TMO Support Library (TMOSL) [Kim99b, 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 RT distributed programs.

EXPERIMENTAL INCORPORATION OF AN H261 MODULE INTO THE TMO PROCESSING FACILITY AND PERFORMANCE STUDIES

As an initial study on the feasibility of the TMO programming of multimedia processing applications, especially the TMO based realization of the global time based synchronization of data streams, we adopted a simple video-data streaming program written by the research staff at ETRI, Korea as a baseline program. This program is based only on the API associated with the Microsoft Window OS family (i.e., Win32 API). It is a real-time distributed computing program which is written in a prevalent low-level concurrent programming style. It captures video scenes by use of a camera attached to the source node and plays them at the screen of the sink node. First, a single-node version of the simple video-data streaming program which is a concurrent program running on one node and picks data from a camera and plays them on the screen, is discussed.

A single-node version of the simple video-data streaming program written in a low-level style

As depicted in Figure 2, the ETRI's program consists of a class named the H.261 Medium class and a simple application program which uses the class. The H.261 Medium class contains member functions which use various low-level functions supplied by Microsoft, peripheral vendors, and protocol stack vendors. The simple application which uses the H.261 Medium class was designed to take a stream of camera inputs, put the video-stream through an encoding and decoding exercise, and playing it on a display device. The structure of this simple application named the H.261_Demo is depicted in Figure 3.

H.261_Demo is depicted in Figure 3.

The H.261_Demo application program and the internal member functions of the instantiated H.261 Medium objects are interrelated as depicted in Figure 4. The application program performs the following three steps:

1. Instantiate the H.261 object "Output-Side" and
invoke "Open()": The "Open()" function creates a thread and binds it to another member function, "WindDrawThread()". This is a tricky style of concurrent programming. The latter function opens a window and instantiates a "WinDraw" object and a "Decoder" object. Thereafter, the thread does not do anything but it hangs on to let the window continue to exit.  

(2) Instantiate the H.261 object "Input-Side" and invoke "Open()": The "Open()" function creates a thread and binds it to another member function, "ReadData()". This is again a tricky style of concurrent programming. The "ReadData()" function opens a window and instantiates a "WinCap" object and an "Encoder" object. It then waits for a play signal which should come from the application program via another member function "Play()". This is yet another instance of tricky concurrent programming.  

(3) Invoke the "Play()" in the Input-Side object: This will result in a play signal being generated and then the thread which has been waiting for that signal from a position inside the "ReadData()" function will enter the loop in which it repeatedly gets a video-frame from the camera, puts it through a service from the "Encoder" object, and invokes the "WriteData()" function of the Output-Side object. The "WriteData()" function in turn invokes a service of the "Decoder" object and then a "draw" service of the "WinDraw" object.

Although the H.261 Demo program functions quite efficiently, it is not so easy to expand this implementation to accommodate complex data processing steps between the encoding step and the transmission (WriteData()) step on the input side and between the decoding step and the display (WinDraw->Draw) step on the output side. It would be particularly painful to incorporate such complex data processing steps that would involve multi-threaded executions.  

**TMO-structured enhancements of the H.261 Demo program: TMO-H.261-1-Node Demo programs**

The first TMO-structured enhanced version of the H.261 Demo program was created essentially in the form depicted in Figure 5. It was named the TMO-H.261-1-TMO Demo program. Each SpM in a TMO is automatically assigned a thread by the TMO execution engine, TMOSM, and thus the TMO-H.261-1-TMO

---

**Figure 4. The H.261 Demo program from ETRI**

**Figure 5. A TMO-structured variation of the H.261 Demo program**
The program possesses a simple, elegant, and easily expandible structure.

The next step was to create a TMO-structured version of the H.261 Demo program that involves two TMOs. The purpose of this step was two-fold. First, this was meant to be a demonstration of the expandability of the TMO scheme. If a TMO becomes large and complicated by incorporating additional application logic, it would be useful to decompose it at some point into multiple TMOs. The two-TMO version, named TMO-H.261-2-TMOs, was also meant to be a demonstration of the ability of the TMO processing facility to communicate a video-data stream from one TMO to another in a manner that would require no great efforts on the part of the programmer while yielding a good performance. Secondly, the TMO is a distributable computation unit. In other words, once a two-TMO version is created, then creation of a distributed computing system in which the two TMOs are located in two different nodes would require trivial efforts. Therefore, this was meant to be an intermediate step toward creation of such a distributed computing version.

In the TMO-H.261-2-TMOs program depicted in Figure 6, a new component was introduced to overcome the message size limitation adopted into the TMO execution engine, TMOSM. TMOSM instantiations running on different sites communicate among themselves messages which are limited to 500 bytes in size. On the other hand, each encoded video-frame ranges in size over 1,200 - 3,500 bytes. Therefore, the packetizing component was produced for use in TMO1 before a video-frame was transmitted to TMO2. In this version, the encoded video-frame was sent by TMO1 via 3 - 7 non-blocking calls for the H261_Decode_SvM in TMO2. This SvM simply deposits the 500-byte packet coming as a parameter from the caller into the ODSS.

**TMO-H.261-2-Nodes Demo program**

After creating the TMO-H.261-2-TMOs program, we proceeded to generate with trivial efforts a version that runs the two TMOs on two different nodes. That version was named TMO-H.261-2-Nodes. One interesting phenomenon which has been observed occasionally is that of frame obsolescence. The buffer used in TMO2 was of the size needed to hold 10 frames, each consisting of up to 10 packets. Now, it was possible for all packets forming the \((N+1)\)-th frame to arrive fully at the buffer of TMO2 before all the packets of the \(N\)-th frame arrived. We suspect two possible reasons. One is that TMOSM and the underlying OS may occasionally deliver the service request messages (i.e., H261_Decode_SvM call messages) from TMO1 to the node hosting TMO2 in different orders. The other is that the service requests are non-blocking calls and thus the next call is often made before the previous calls are fully processed by H261_Decode_SvM. Moreover, an SvM in a TMO can run in a pipeline mode when multiple requests come within a short interval.

TMO2 was designed to discard obsolete video-frames. In other words, H261_Decode_SpM would not pick the \(N\)-th frame after picking up the \((N+1)\)-th frame. In video-handling, this is a natural thing to do in general except in some special situations. It is better to play the \((N+1)\)-th frame on time even if the \(N\)-th frame was missing instead of delaying the play of the \((N+1)\)-th frame until the unfortunate \(N\)-th frame arrives quite late. This is particularly so in a high-quality video-processing system.

Of course, an ideal situation is where such frame obsolescence does not occur while playing all frames on time. We monitored the occurrences of frame obsolescence while running the demo programs. In the case of the two-TMO version running on one node, i.e., TMO-H.261-2-TMOs, no frame obsolescence has been observed. On the other hand, in the case of the two-TMO version running on two different nodes, i.e., TMO-H.261-2-Nodes, two frame obsolescence events were detected while processing 2,400 frames. Further studies are required to identify the causes of frame obsolescence.
Performance measurement of the TMO-H.261-2-TMOs and TMO-H.261-2-Nodes programs

We considered the following latency to be one of the most meaningful performance measures:

\[ \text{Frame travel time} := \text{Latency from the instant at which a frame has been captured to the instant at which the receiving TMO issues a command for display of the frame.} \]

First, we measured the frame travel time while running the TMO-H.261-2-TMOs program which required one node. The invocation cycle was the same 50 milliseconds for both SpMs. The results are shown in Figure 7. The frame travel time ranged over 40 milliseconds - 70 milliseconds.

On the other hand, we realized that we had chosen the start time-windows for both the SpM in TMO1 and the SpM in TMO2 unwisely. When both TMOs run on the same node with a single CPU, this would obviously lead to quite poor performance. We then adjusted the start time-windows such that the SpM in TMO2 would start 25 milliseconds after the SpM in TMO1 would start. The result was a significant improvement in the frame travel time as shown in Figure 8. Now, the frame travel time ranged mostly over 20 milliseconds - 40 milliseconds except for half a dozen occasions out of 2,400 measurements. This kind of performance optimization possibility is a positive consequence of the emphasis in devising the TMO scheme on the analyzability of RT programs.

The next step was to run the TMO-H.261-2-Nodes program and measure the frame travel time. The results are shown in Figure 9. The frame travel time ranged over 30 milliseconds - 60 milliseconds except for a couple of occasions at the beginning. Therefore, the performance is better than the case of Figure 7 while being worse than the case of Figure 8. In this two-node version, it is not worth trying to adjust the start time-windows for SpMs. Such adjustment would be worthwhile only when the network delay jitter is very small.

EXPERIMENTAL INCORPORATION OF AN RTP MODULE AND AN H261 MODULE INTO THE TMO PROCESSING FACILITY

We adopted not only the H.261 Medium Class but also the RTP Medium Class from ETRI along with an application program, which we call here "the simple RTP+H.261 Demo program", as a part of the baseline program. The application program and the internal member functions of the instantiated RTP Medium objects are interrelated as depicted in Figure 10.

We created one TMO-structured enhanced version of the RTP+H.261 Demo program and the version was named the TMO-RTP-H.261 Demo program. The simple and easily extensible structure of that program is...
depicted in Figure 11. A two-node version of this was implemented. Since the RTP module was capable of communicating a message much larger than a video-frame, there was no need to incorporate a packetization module. In the first implementation, executions of the RTP module were delegated to the threads which were called LIITs (Local I/O Interface Threads) and different from the threads executing TMO methods. Delegations are realized via calls for an API function in TMOSL, use_LIIT(). As a result, SpM executions and RTP function executions can proceed concurrently.

Performance measurements were also done and the data showed fine performance of this TMO-structured version, too. The situation is similar to those of earlier discussed variations of the H.261_Demo program. The details of the data are not included here due to the space limit.

CONCLUSION

The TMO programming scheme and supporting tools enable easy realization of inter-stream and intra-stream synchronization of multimedia data based on the global time based coordination principle. TMO programs handling such synchronization tend to have simple and easily expandable structure. In this paper we reported the results of an initial feasibility evaluation of the TMO-based approach and supporting tool. They are clearly positive results.

Exploration of the potentials of the TMO programming scheme in economic and reliable implementation of multimedia processing application programs is at the early stage. Numerous conceivable approaches for using the TMO programming scheme and support facility in handling multimedia data streams need to be evaluated in future research.

Acknowledgements: The research reported here was supported in part by the NSF under Grant Numbers 99-75053 (NGS) and 00-86147 (ITR), and in part by the US DARPA (NEST Program) under Contract F33615-01-C-1902 monitored by AFRL. The work of Doo-Hyun Kim was also supported by ETRI, Korea. No part of this paper represents the views and opinions of any of the sponsors mentioned above.

REFERENCES

[Aky96] Akyildiz, I.F., and Yen, W., "Multimedia


