A TMO-based approach to tolerance of transmission jitters in tele-audio services

Doo-Hyun Kim*, K H (Kane) Kim†, Sheng Liu† and Jin-Hyung Kim‡

*KAIST, Dae-Jeon, Korea. Email: doohyun@etri.re.kr
†University of California Irvine, CA, USA. Email: khkim@uci.edu
‡KAIST, Dae-Jeon, Korea. Email: jkim@kaist.ac.kr

An approach to realizing high-quality audio streaming tele-audio services over networks by applying the global time based coordination principle is presented. The goal is to play the audio stream at a remote site with minimal loss of the temporal relationship among the audio data units in spite of the jitters in the transmission delays over networks. The results of a feasibility evaluation of this global time based approach for audio streaming tele-audio services with tolerance of transmission jitters are presented. The effective programming tool used was the Time-triggered Message-triggered Object (TMO) programming tool, a tool enabling construction of real-time distributed computing programs in the form of networks of high-level easily analyzable real-time objects.

Keywords: Real time, global time, multimedia, tele-audio, stream, jitter, tolerance, TMO, time triggered, message, object, middleware, distributed, programming, synchronization.

1. INTRODUCTION

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 [Aya01, Bla96, Cel00, Her98, Hun01, Ste96, Bla96, KSK02]. Moreover, the amount of effort consumed by the designer and the programmer in the specification of such temporal relationship and programming of the processing steps to maintain the relationship among the processed data elements has become a major issue confronting the community engaged in multimedia software technology research [Aky96, Bat99, Bai98, Gib95, Kas01, KDH02, Lia01, Ste96, Xie99, Zan95].

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 captured individual samples or blocks of samples transferred together from a source to a sink(s). The data elements in a stream must be presented, i.e., replayed, at a sink node in a manner exhibiting the same temporal relationship which 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 [Jef95, Kad96, Kan01, Moo98, Ram94, Sre00].

The temporal relationship maintenance must also be applied to related data streams. One of the most commonly encountered situations where such a requirement is evident, is the simultaneous playback of audio and video with the lip synchronization [Liu96, Owe99, Xie99]. If both media are not played with good synchronization, the result will not be of acceptable high quality. In general the inter-stream syn...
Chronization involves relationships amongst the data of various kinds of media including pointers, graphics, images, animations, text, audio and video [Aya01, Bai98, Bae96, Bla96, Cel00, Hun01].

A fundamental approach which holds great promise in this area, but has been minimally exploited by the research community is the global time based coordination of distributed actions [Kop90, Kop97, Aky96]. One reason for the slow exploitation of this promising approach is has been the lack of cheap facilities for providing a good quality global time base. Recent maturing of the GPS receiver technology [Jef95, Kad96, Kan01, Lia01, Moo98, Ram94, Ste96] that dealt only with QoS-related information such as sequence number, time-stamp, delay, jitter and packet loss. In the last few years, attempts to achieve better end-to-end QoS have been made. These attempts include marking protocols in the play of the audio stream at a remote play site. Such reduction of the impacts of communication delay jitters appear in minimal forms in the play of the audio stream at a remote play site.

Still, it is important to provide tools enabling the application system developers to specify and effect desirable temporal relationships among multimedia data elements relatively easily, including both inter-stream and intra-stream synchronization requirements. The tools enabling easy application of the global time based coordination principle are particularly desirable.

In typical communication network environments, message communication delays show varying degrees of jitters [Jef95, Kad96, Kan01, Lia01, Moo98, Ram94, Ste96]. Therefore, it is necessary to realize audio streaming tele-audio services over a network in such a manner that the impacts of communication delay jitters appear in minimal forms in the play of the audio stream at a remote play site. Such reduction of the impacts of communication delay jitters is called the transmission jitter tolerance. It is a key issue in realizing high quality audio streaming tele-audio services. In this paper we report the results of a feasibility evaluation of a global time-based approach for streaming tele-audio services with effective tolerance of transmission jitters.

In Internet technology research communities, Quality of Service (QoS) has been a major research issue since the possibility of running Voice over IP (VoIP) was first recognized. In order to achieve voice ‘toll’ quality, call-voice datagrams must arrive in a timely manner with only a few datagrams dropped on the path [Jia00, Fin02, Has00, Iid00]. The VoIP started with the use of RTP (Real-time Transport Protocol) [Sch96a] and RTCP (Real-Time Control Protocol) [Sch96b] that dealt only with QoS-related information such as sequence number, time-stamp, delay, jitter and packet loss. In the last few years, attempts to achieve better end-to-end QoS have been made. These attempts include marking protocols (IEEE 802.1Q/p [Lin98], DiffServ [Blk98] and MPLS [Rsn01]), reservation protocols (RSVP) [Bar97], classifying protocols (IntServ) [Ber00] and policy-based systems. The IEEE 802.1Q/p, DiffServ, and MPLS are LAN, MAN and WAN protocols, respectively.

In line with these efforts, ITU-T is defining a generic IP service performance model for use in planning and offering global IP services [ITU02]. This model categorizes most multimedia services into five classes (from class 0 to class 5) and defines different levels of QoS objectives for each class. Among them, class 0 is for real-time, jitter sensitive, high interactive applications such as VoIP and video teleconferencing, and the QoS objectives are delay of 100ms and jitter of 50ms. In such a network environment where the transmission delay and jitter are bounded, the end-systems will be responsible for containing the play jitter within the level of human tolerance, 80ms, which is the empirical suggestion estimated to be 80ms in [Ste96].

The programming tool-kit used during the feasibility study was the Time-triggered Message-triggered Object (TMO) programming tool-kit and it supports easy practice of object-oriented (OO) real-time (RT) distributed programming. Since the early 1990’s the second co-author and his collaborators have been enhancing the TMO programming scheme. The TMO scheme has been devised to enable construction of real-time distributed computing programs in the form of networks of high-level, easily analyzable real-time objects [Kim94, Kim97, Kim99a, Kim00]. OO RT distributed computing is now a rapidly growing branch of computer science and engineering [Bo100, IEE01, 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 major improvements over traditional RT programming practiced with low-level programming languages and styles.

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) so that typical OO programmers can adopt it with relatively little effort.
• (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 focus on essential design activities.
• (TB3) The scheme enables systematic design time guaranteeing 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 easily design global time-based coordination of distributed actions. This paper presents a TMO-based approach to minimizing the impacts of the local jitter, i.e., jitter in the application processing rate, of the end-system and the transmission delay jitter on the quality of tele-audio services.

The global time-based approach to tolerance of the jitter in transmission delays requires high-precision capture time-stamping of audio data units. During this experimental study we learned that with widely used commercial OSs such as Microsoft Windows, such capture time-stamping was not easy. After considerable measurements and observations of the behavior of the OS and the device driver involved, a technique for overcoming the capture time-stamping problem has been devised. The experiment validated the practicability of the TMO based approach for high-quality audio streaming tele-audio services. The programming approach for audio streaming tele-audio services is discussed in this
In the next section, an overview of the TMO programming scheme is provided to make the paper reasonably self-contained. Then in Section 3, the structure of an audio streaming tele-audio service TMO network is discussed. Section 4 presents the measured data which show the difficulty of achieving high-precision capture-time-stamps. A technique, devised to overcome the capture-time-stamping problem, is presented in Section 5 and the measured data showing the high quality audio play at a remote site are also presented. The paper concludes in Section 6.

2. AN OVERVIEW OF THE TMO PROGRAMMING 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 are 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 at 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 start with the objective of enabling design time guaranteeing timely actions. The TMO incorporates several rules for execution in 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.

2.1 TMO structure and design paradigms

TMO is a natural, syntactically minor, and semantically powerful extension of the basic 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); Each ODSS is a group of data members and is a unit that can be locked for exclusive use by one method execution at a time as well as for shared use by multiple concurrent method executions which perform read-only operations on the data members contained.
- 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.

The 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 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 only allowed when potentially conflicting SpM executions are not in place. An SvM is only allowed to execute when an execution time-window opens up that is 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. However, the BCC does not stand in the way of either concurrent SpM executions or concurrent SvM executions.

(d) Guaranteed completion time for method execution and deadline for result return

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 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

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

which has the same effect as

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

A provision is also made for making the AAC section of an SpM only contain candidate triggering times, not actual triggering times, so that a subset of the candidate triggering times indicated in the AAC section may be dynamically chosen for actual triggering. Such a dynamic selection occurs when an SvM within the same TMO object requests future executions of a specific SpM. Each AAC specifying candidate triggering times rather than actual triggering times has a name. An underlying design philosophy of the TMO scheme is that an RT computer system will always take the form of a composition of hardware, node OS and middleware (a middleware) available, the server object can be implemented to be transparent to the application TMO programmer. Prototype implementations based on several of the most popular OS kernels such as Windows NT/XP, Windows CE and Linux exist [KHJK02, 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 pre-emptible for an excessively long time) gets involved. Prototype implementations that use inter-object communication facilities rather than TCP and UDP, e.g., CORBA ORB and Microsoft COM, also exist [Kim01].

A prototype implementation in the form of a CORBA service, TMOSM/AnyORB/NT, that runs on platforms equipped with Windows NT and a basic ORB and supports CORBA-compliant application TMOs has also been obtained [Kim01].

A friendly application programming interface (API) wrapping the services of TMOSM has also been developed and called the TMO Support Library (TMOSL) [Kim99b, Kim00]. It consists of a number of C++ classes. Recently, this API has been further refined to minimize the burdens on the programmer by relying on a versatile macro processor [Kim00d]. TMOSL empowers C++ programmers with powerful and natural mechanisms for the specification of unique and essential features of RT distributed programs.

3. A SIMPLE TMO NETWORK USING LOW-PRECISION CAPTURE-TIME-STAMPING

A simplistic TMO structuring of a LAN based two-PC system performing global time-based synchronous audio streaming tele-audio services is depicted in Figure 2. In this approach, a call-back function to be invoked by the microphone driver whenever the driver captures an audio sample, is designed to perform the time-stamping in addition to its normal function of moving the captured audio data into some data structure in the application workspace. This time-stamping is much less accurate than the time-stamping done inside the microphone interrupt handler, which is a front-end part of the device driver. However, this is the price for relying on COTS device driver rather than building a customised one.

The call-back function deposits the picked audio data into the ODS of Sender TMO by calling a public member function known as Audio_Depositor. The data structure in the

2.2 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 [Kim96a, 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

![Figure 2](image)
ODES which is used to hold the audio data is a circular queue. Then Audio_Acquire_SpM, which runs periodically at the interval of X ms, extracts one audio sample from the circular queue in each run and transmits to Receiver TMO by calling Receiving_SvM of the latter TMO. In our experimental studies X was set to 25, which also meant that the microphone controller was initialized to generate interrupts at every 25 milliseconds and let the interrupt handler copy away the audio sample captured up to that point while continuously sampling the audio signals generated by the microphone.

Note that Audio_Depositor is neither an SpM nor an SvM. It is a public member function of Sender TMO unlike SpMs and SvMs which are private member functions. Because of its public nature, Audio_Depositor can be called directly by the call-back function of the microphone driver which is located outside the TMO. The call-back function is carried out by a non-real-time thread assigned by Windows OS. That same thread also executes Audio_Depositor whereas SpMs and SvMs are executed by real-time threads assigned by TMOSM. Therefore, Audio_Depositor is quite different from SpMs and SvMs. On the other hand, by being a member function of Sender TMO, Audio_Depositor can access the ODS of Sender TMO. Moreover, Audio_Depositor provides a service to an external client, i.e., in this case the call-back function. Therefore, Audio_Depositor can be viewed as a special SvM (SSvM). It is not only special because its a public member but also because it can only be invoked by local clients.

Although Audio_Depositor function and Audio_Acquire_SpM share the circular queue in the ODS, the sharing is safe and does not cause any non-negligible blocking of either party for two major reasons. First of all, CPU time-slices are first allocated by TMOSM to real-time threads running SpMs and SvMs and only those time-slices not assigned by TMOSM to any real-time thread are given by the underlying kernel to non-real-time threads. Secondly, the circular queue is actually implemented in the form of a pair of non-blocking update buffers (PaNBUB) of which the concept was presented in [Kim02]. The PaNBUB scheme is depicted in Figure 3 and it works essentially as described below.

Each of the two communicating parties, the SSvM and the SpM, owns a circular buffer. Each party can read the other party’s buffer but cannot write into it. Each party can write to its own buffer without being blocked by anything and thus is a non-blocking writer of its own buffer. When each party reads the other party’s buffer, it follows the double-checking reader protocol produced in [Kop93]. Basically, after reading complex data in the other party’s buffer through multiple machine cycles and before completing the reading procedure, a double-checking reader checks if the owner (which is the non-blocking writer of the buffer) has come inside the buffer or not. So, the double-checking reader sees the data again and performs the check again and this may be repeated until the reader finds that its reading has been completed without any possible interference by the non-blocking writer.

In the PaNBUB scheme, each party owns an update counter in addition to its own circular buffer. Therefore, when the SSvM (Figure 3) inserts a data item (captured audio sample) into its own circular buffer, it first tags the data item with the current value of its update counter, then inserts the data item and increments the update counter. When the SpM accesses the circular buffer owned by the SSvM, the SpM first reads the update counter of the SSvM. Once the SpM has successfully read the update counter of the SSvM, the SpM compares the newly read value of the update counter against the value of the update counter attached to the data item that it picked (i.e., read) last time. Through this comparison, the SpM knows whether there is any new data item which it did not pick before. If the update counter indicates that there are some new data items in the buffer, the SpM reads those without any interference from the owner, i.e., the SSvM, because of the following: If the SSvM needs to access the buffer, it will be accessing the buffer slots which are different from those being accessed by the SpM due to the aspect of the PaNBUB scheme, which is described below.

After the SpM has successfully read items in the circular buffer owned by the SSvM, it inserts into its own circular buffer the value of the update counter of the SSvM attached to the data item that the SpM read the last. Note that this saved counter value gives a clear hint to what recent data item(s) in the buffer of the SSvM were read by the SpM. The SSvM’s counter value saved in a slot of the buffer of the SpM is actually tagged the value of the SpM’s update counter. Also, the SpM increments its own update counter as the last step in inserting the read value of the SSvM’s update counter into its buffer. Note that if the SpM has no other data to send to the SSvM, which is the case in the arrangement of Figure 2, one can replace the circular buffer of the SpM by a simple integer containing the last value of the SSvM’s update counter which has been seen in the tag field by the SpM.

When the SSvM inserts new data items into its buffer, the steps involved are actually more than those explained above. The SSvM first reads the update counter and the buffer of the SpM. From the data items read from the buffer of the SpM, the SSvM learns which slots in its own buffer were read last time by the SpM. Those items read by the SpM can then be erased from the buffer of the SSvM. After taking these steps, the SSvM takes the previously explained steps, i.e., tags the data item with the current value of its update counter and then inserts the data item and increments the update counter. In a fully symmetric situation where the SpM produces data in addition to counter values in its own buffer for reading by the SSvM, the SSvM will also have to save into its own buffer the values of the SpM’s update counter tagged to the data items that it has read.

If the SSvM finds out that its own circular buffer is full

![Figure 3](Image)

*Figure 3* The pair of non-blocking update buffers used between the SSvM and the SpM
and none of the buffer slots can be erased, then the SSvM will wait a while and then check the update counter and the buffer of the SpM again. Therefore, pick-up by the SpM of the audio samples from the buffer owned by the SSvM will never be interfered with by the SSvM for any significant amount of time. Similarly, a deposit by the SSvM of audio samples into its own circular buffer will never be interfered with for any significant amount of time, except when buffer saturations occur.

On In the receiver site, Receiving_SvM deposits a received audio sample into the circular queue in the ODS. Then, Audio_Play_SpM, which runs periodically at the interval of 25 ms, extracts one audio sample from the circular queue in each run and plays it through the speaker driver. In this case, the circular queue shared by the SvM and the SpM is just the standard type of a circular queue. This is because there cannot be any data access conflicts between SpMs and SvMs under the BCC. Audio_Play_SpM was designed to pick just one audio packet and play it, even if it finds multiple packets in the circular queue. On the other hand, experiments have shown that the SpM quite frequently finds that the circular queue to be empty.

Intervals between successive plays of audio data units were measured. Figure 4 shows some measured data. As shown, the interval fluctuates between 15 milliseconds and 65 milliseconds which means that the jitter in the play interval can be as large as 50 milliseconds.

Subsequent investigations showed that this jitter was not only due to the jitter in network communication delays but more seriously, caused by the irregular delays in invocation of the call-back function triggered by the commercial vendor’s device driver for the microphone. Recall that the call-back function performs audio capture time-stamping.

Figure 5 shows the measured intervals between successive time-stamping operations. The interval ranges over zero to 65 milliseconds. This means that although the microphone generates an audio sample and raises an interrupt every 25 milliseconds, the delay in starting the call-back function invoked by the microphone interrupt handler fluctuates over the range of near 0 to 40 milliseconds. This is mainly because the thread running the call-back function is a non-RT thread executing at a much lower priority level than the level than the threads running TMO methods do. To make the call-back function to be run by an RT thread requires replacing the commercial vendor’s device driver by a custom-designed one.

We also examined the interval between successive iteration start-times of Audio_Acquire_SpM. The interval ranged from 18 to 33 milliseconds. Ideally the interval is supposed to be 25 milliseconds, this represents the jitter of about 15 milliseconds or 8 milliseconds in either direction from the ideal value. This is due to the fact that the TMOSM instantiation used in the experiment used time-slicing and allowed a TMO method to get 1 time-slice every 9 milliseconds. Therefore, if TMOSM failed to find any ready TMO when it was prepared to allocate a time-slice it instead allocated it to another system thread. If shortly afterwards, a SpM iteration becomes ready, then almost 9 milliseconds must pass by before the SpM iteration gets a time-slice. Therefore, the interval ranging from 18 to 33 milliseconds indicates that TMOSM runs quite reliably.

To make the call-back function to be run by an RT thread requires replacing the commercial vendor’s device driver by a custom-designed one. Instead of exploring that path, we formulated a different approach for reducing the jitter in the tempo of playing successive audio samples – which is discussed in the next section.

4. A TMO NETWORK USING IMAGINARY CAPTURE-TIME-STAMPING

If a high-precision capture-time-stamping can be realized conceptually then it is not difficult to achieve nearly jitter-free audio play in a remote site as long as a reasonably tight bound on the network communication delay exists. One can ensure that the buffer in the remote play site is always non-empty and each audio data unit is played at the time which is called the target play time (TPT) and determined by adding a fixed delay to the capture time. The fixed delay added is called the target streaming tele-audio delay (TSTAD).

In such a case, the capture-to-play delay, i.e., the delay from the instant at which an audio data unit is captured to the instant at which the data unit is played in a remote site, will emerge as the next big concern. Therefore, a technique for realizing high-precision capture-time-stamping is the main subject of discussion here.

The first step taken was to remove the call-back function and make Audio_Acquire_SpM directly access the data buffer kept inside the microphone driver. The microphone driver model, defined as a part of the Microsoft Windows specification, supported such direct access to the data buffer.
We hoped to see some improvement in the precision of capture-time-stamping through this change, but no significant improvement was observed (although some small improvement was evident). The measured intervals between successive time-stamping operations ranged from zero to 50 milliseconds. The main reason for insignificant improvement was that the microphone interrupt handler was again delegating (to a non-RT thread) the job of saving the captured audio sample in a buffer slot.

Given the situation where high-precision capture-time-stamping was not possible without replacing commercial microphone drivers with a custom-built ones, the next best cost-effective approach (devised by the authors) was to take advantage of the fact that audio samples were captured at the fairly precise interval of 25 milliseconds. If the microphone starts at 10am, the first audio sample will be produced at 25 milliseconds after 10am. Therefore, it is correct to attach the capture-time-stamp valued at 25 milliseconds after 10am to this audio sample. Similarly, the next audio sample should tag the capture-time-stamp valued at 50 milliseconds after 10am. Since there is no way of creating these capture-time-stamps in 'perfect real time', i.e. generating the capture-time-stamp valued at 75 milliseconds after 10am precisely at 75 milliseconds after 10am, they are called here the Imaginary Capture-Time-stamps (ICTs).

In order to use ICTs, it is necessary to ensure that audio samples are picked up by Audio_Acquire_SpM in the sequence of their birth-times, i.e. times at which they are generated by the microphone controller, without losing any of them. If an audio sample is lost on its way from the microphone controller to the memory used by Audio_Acquire_SpM and it is not detected by the SpM, then all subsequent audio samples will have the wrong ICTs attached. Therefore, tests were conducted with the configuration depicted in Figure 6. The microphone driver was initialized to use three buffer slots. The SpM was designed to check the buffer slots in a circular fashion. During the tests the SpM never found an example of the buffer slots contained audio samples in the wrong order. From this and other factors it was also clear that audio samples were never lost in this node equipped with the microphone.

The remote play site, Audio_Play_SpM, picks only one audio packet from the queue that holds arrived audio packets in each round of its execution. The ICT attached to the audio packet at the front of the queue is used to compute TPT with the following formula:

\[
TPT = ICT + TSDTTAD
\]

Then TPT is compared against the current time.

Case 1: TPT < Current Time
In this case, the network communication delay experienced was greater than TSDTTAD. The audio packet must be played immediately. This case should never or rarely occur.

Case 2: TPT > Current Time
This is a normal case. However, one must consider the possible need for playing this audio packet during the next round of the SpM execution rather than during the current round, see Figure 7. Suppose that round i of the SpM is currently in execution and CT represents the current time. Round i is not supposed to last longer than GCT (guaranteed completion time) which is shorter than the SpM period. The SpM period was chosen to be 25 milliseconds during this experimental study. GCT is normally quite a bit shorter than the SpM period otherwise, different methods in the same TMO and other TMOs resident in the same node cannot have much CPU time to use. In a sense, the SpM has the right to use the GCT amount of CPU time in each round without being interfered with by other methods.

Let Bi denote the time at which the i-th SpM period begins. The case of TPT > CT can now be divided into two sub-cases.

Case 2.1: TPT < Bi + GCT
In this case, the SpM can wait until TPT arrives and then play the audio packet.

Case 2.2: TPT > Bi + GCT
Unless TSDTTAD is chosen to be a dangerously small number, TPT cannot be later than Bi+1. Therefore, TPT in this case falls into the period in which the SpM is not active. This means that the audio packet cannot be played at TPT. A compromise is to play it either immediately before Bi + GCT or immediately after Bi+1. The best compromise is then to play it immediately before Bi + GCT if TPT is closer to Bi + 1 and if not then play it immediately after Bi+1.
This strategy, based on the use of ICTs, was implemented as a part of this experimental study. The intervals between successive plays of audio data units were measured. Two PCs which were running TMOSM on top of Windows XP OS and connected via an Ethernet LAN, were used. TSDTTAD was chosen to be 48 milliseconds. Figure 8 shows some measured data. As shown, the interval fluctuates between 18 milliseconds and 33 milliseconds and this means that the jitter in the play interval is limited within 10 milliseconds in either direction. Therefore, these data show that a major improvement in the temporally stable replay of audio streams in remote sites can be achieved by use of the ICT based streaming tele-audio approach.

The jitter shown in Figure 8 is mainly due to the variable fluctuating delays in starting each round of Audio_Play_SpM. The improvement in the jitter in the play interval could also be felt by human ears.

5. CONCLUSION

A practical approach to implementing the global time-stamp based temporally stable tele-audio-streaming service was presented. The core of this practical implementation scheme is in two parts. One part is the TMO programming scheme including the execution engine centered around middleware TMOSM and the associated API, TMOSL. The TMO scheme enables relatively easy high-level programming and manipulation of global time-stamps and timely processing of audio and other multimedia data.

The other part is the approach to use ICTs which are in a sense retroactive time-stamps. This enabled us to overcome the weaknesses of commercial microphone drivers. At present designing device drivers is not a mature art and timing behaviors of many device drivers in popular OS environments are less than desirable. Therefore, it is expected that the ICT approach will find a few other major applications.

The global time-based audio-streaming tele-audio service implementation scheme presented in this paper can be used in a straightforward manner in LAN environments where tight bounds on network communication delays can be easily determined. However, its use in widely open Internet environments is a different matter and requires much further research [Chr92, Jia00]. A particularly important issue is to handle packet losses which is a serious problem unlike in LAN environments [Per98, Ros00, San00].

REFERENCES


[Jia00] W. Jiang and H. Schulzrinne, “Modeling of Packet Loss and Delay and Their Effect on Real-Time Multimedia Service Qual-


