A CORBA BASED INTEGRATED QOS SUPPORT FOR DISTRIBUTED MULTIMEDIA APPLICATIONS

Hung Keng Pung, Wynne Hsu, W C Wong
National University of Singapore
Email: {punghk, whsu}@comp.nus.edu.sg; ccewwl@nus.edu.sg

ABSTRACT

Advance object oriented computing platform such as the Common Object Request Broker Architecture (CORBA) provides a conducive and standardized framework for the development of distributed applications. Most of the off-the-shelf CORBA are implemented over legacy network transports and distributed processing platforms such as TCP/IP and RPC. They are not suitable for real-time applications due to their high processing overheads, and lack of features and mechanisms in supporting quality of service both at the network level and at the end-host level. To overcome this limitation we have designed and implemented a CORBA-based Real Time Stream Service (RTSS) that allows real-time streams to be managed through the 'CORBA channel' but by-passing the heavy CORBA protocol stacks. RTSS aims to achieve an integrated QOS framework that incorporates both host scheduling and end-to-end network-level QOS to better support the processing of distributed multimedia applications over ATM networks. For host scheduling, a novel scheme of frequency-based scheduling mechanism has been proposed to cope with dynamic CPU load condition. The scheme has been implemented for a stand-alone host and will be extended to the networked environment. At the network-level QOS, RTSS provides object-oriented application programming interfaces (APIs) which guarantee end-to-end QOS when operating directly over ATM adaptation layers. The benefits of RTSS for the development of real-time multimedia distributed applications are demonstrated through a number of experiments.

Keywords: quality of service (QOS), CORBA, real-time scheduling

1. INTRODUCTION

Quality of Service (QOS) is an important issue in the development of multimedia applications. Both the developers and users of applications need to have some form of guarantee of an acceptable quality of service provided by the multimedia applications. The issue is made complicated by two facts: (1) most existing multimedia applications run on generic operating systems that have no concepts of real-time requirements. This makes the guarantee of QOS hard to achieve in practice [1]; (2) popular distributed object computing (DOC) technologies such as the Common Object Request Broker Architecture (CORBA) [2] are ideal for the development of portable and interoperable distributed servers/clients applications. However the current implementations of commercial DOC are less suitable for real-time multimedia applications [3]. Motivated by these observations, we have developed a Real-Time Stream Service (RTSS) to better support the development of interactive multimedia applications in conventional UNIX environment. The main goal of RTSS is to provide a stream service for continuous media applications with integrated QOS supports while retaining the merits of CORBA.

RTSS allows real-time streams to be managed through the 'CORBA channel' but by-passing the heavy CORBA protocol stack for transportation of the real-time data streams. Our approach coincides with that of the OMG's 'Control and management of A/V streams" released recently [4][24]. The preliminary results of our work was first reported in ICMCS97 [5]. It demonstrated the feasibility and benefits of separating the data path of real-time stream with its CORBA based control and signaling path. This paper presents the detailed design, implementation and experimental evaluation of RTSS. We hope our experience is particularly useful to people who are extending their off-the-shelf CORBA packages to support the new OMG's A/V stream scheme. The remaining paper is organized as follows: Section 2 outlines the related work; the design approach and functional overview of RTSS are presented in Section 3; Section 4 presents the details of design and implementation of RTSS. We present results of performance measurements in Section 5 and conclude the paper in Section 6.

2. RELATED WORK

Several advances have been made in improving existing distributed object computing environment. They include: the C++ wrappers by [6][7] which does not provide QOS guarantees for real-time streams; the Realvideo [8] and Vxtreme [9] which are not CORBA based and rely on proprietary stream establishment and control mechanisms to access multimedia context; the multimedia information stream services (MISS) [10] which is not CORBA based and lacks useful features such as stream control, end-to-end QOS guarantees and admission control; as well as the proposed extension of TINA-ODL [11] for the specification of the stream interface that may include mechanisms for applications in negotiating network QOS. However, there has been little attempts to integrate host scheduling with the network QOS to ensure an integrated QOS support for multimedia applications within the context of CORBA. One well-known exception is TAO [12] which provides real-time support as part of their CORBA stacks through the implementation of a gigabit real-time I/O subsystem at the low level, a real-time object adapter above their ORB and some means for QOS specification by the applications. The TAO approach is challenging from the research point of view, but it is clearly not in line with that of OMG who prefers solutions requiring minimum changes to existing CORBA implementation with the use of 'out-of CORBA's band' A/V data path (IONA Inc. is taking this approach in developing their Orbix MX [24], which is an implementation of the CORBA A/V stream specification.) Also, despite the system complexity, the practical benefits of real-time CORBA for A/V stream applications is doubtful. This is confirmed in a recent report on an extension of TAO for supporting OMG A/V stream service [25]. The TAO group found through experiments that the throughput of running A/V streams entirely over TAO ORB was inferior to that running over an 'out-of TAO band' TCP pipe.

At the host OS scheduling level, research has been focused primarily on fully preemptive systems [13][14]. Two popular scheduling algorithms are the earliest deadline (ED) and rate-monotonic (RM) algorithms. Both algorithms are optimal as long as the CPU is not overloaded. In the case of ED, the CPU utilization must be less than 1; while in the case of RM, the utilization cannot exceed $n(2^{(1/n)} -1)$. [15][16] have improved the utilization bounds for RM (CPU Utilization < 0.88) by studying the average case and the harmonic characteristics of tasks. However, these algorithms do not allow for the possibility of variable execution and are unable to cope with overload conditions effectively. As a result, many best-effort scheduling heuristics [17][18] have been proposed in an attempt to cope with overload conditions. A related school of thought is the study of imprecise computation technique [19][20]. A task is divided into two parts, a mandatory sub-task which must be executed and an optional sub-task which serves to refine and improve the result. In overload conditions, the optional sub-task can be skipped to allow all mandatory sub-tasks to be completed. This allows for more graceful degradation of performance under overload conditions. In this paper, we have incorporated the idea of mandatory and optional sub-tasks to the host OS level. Note that multimedia scheduling has traditionally been done by replying on a real-time microkernel such as the RT-Mach [*1]. Here, our focus is to perform multimedia scheduling on general OS platform such as UNIX system. This is because while systems such as RT-Mach [*1], Chorus [*2], RBE[*3] can give strict guarantees on the amount of allocated CPU cycles, they do not adapt well to the dynamic characteristics of multimedia applications and provide limited control to the programmer to manage the scheduling, especially under overloaded conditions. Another platform which combines architectural support with OS scheduling mechanisms is the multimedia server on the Spring real-time system[*4]. The Spring system incorporates a planning-based scheduler for distributed computing. Under overloaded conditions, Spring uses an adjustment algorithm that iteratively degrades the QoS of the multimedia server to accommodate hard real-time tasks, although there is no mention of how each multimedia stream under the server will react to the change. Using our proposed frequency-based scheduling approach, the programmer is in full control of how individual stream will react under overloaded conditions.

*1 C. Lee, R. Rajkumar, and C. Mercer. Experiences with processor reservation and dynamic QoS in real-time mach. In the proceedings of Multimedia Japan 96, April 1996.

*2 Geoff Goulson, Andrew Campbell, David Hutchison, and Francisco Garcia. BT URI project; WP2/D2: sessopm acceptance and QoS management in end-systems. Technical Report MPG-95-21, Department of Computing, Lancaster University, 1995.

*3 K. Jeffay and D. Bennett. A rate-based execution abstraction for multimedia computing. Lecture notes in Computer Science, 1018:64-78, 1995.

*4 H. Kaneko and J. Stankovic. A Multimedia server on the spring real-time system. Technical Report UMass CS 96-11, Department of Computer Science, University of Massachusetts, Amherst, 1996.

3. DESIGN OBJECTIVE, APPROACH AND FUNCTIONAL OVERVIEW

A stream is a time-ordered flow of information from a source to a target. Streams imply incremental processing of data which are generally large in volume. Continuous real-time data is a typical example of such streams.

In the context of RTSS, a stream refers to a network communication facility through which two distributed applications are connected via their end point interfaces known as RTSS ports. A stream is modeled as an unidirectional virtual pipe where media data (a continuous flow of particular media such as audio, video or text) are delivered in the same order and rate as generated by the source. This is in-line with most multimedia applications in which information flows are usually asymmetrical.

RTSS provides stream services with different service qualities through a collection of stream interfaces. These interfaces support the creation of RTSS ports, specification of QOS of a port, attachment of handlers and binding of these ports to form a RTSS stream between two collaborating applications. They also provide stream controls such as pausing and resuming of continuous media data. The key design objectives of RTSS are outlined as follows:

- To provide a set of object-oriented interfaces (APIs) as user libraries for programming multimedia applications in CORBA environment.
- To provide a software abstraction over multiple transport protocols for meeting different transmission requirements of media data.
- To allow applications to specify the desired network and host QOS during connections setup and honor them when the connections are successful.
- To achieve a high-level of performance.

Error! Reference source not found. shows interactions of RTSS with other Object components defined in the OMG's Object Management Architecture Reference Module (OMA/RM) [2]. The application objects can converse with RTSS through the IDL interfaces provided. In this case, the request and the result of the request are exchanged via ORB. This is the only way that objects in a different address space (in the same or a remote host) can communicate with RTSS. Application objects can also talk to RTSS directly using the RTSS APIs. These interfaces are only available from the same address space and provide application objects with facilities to attach their handlers (Section 4.1) to an RTSS port. These handlers are "up-called" later when RTSS needs to deliver or receive data to and from these objects. We explain this technique in detail in Section **Error! Reference source not found.**.

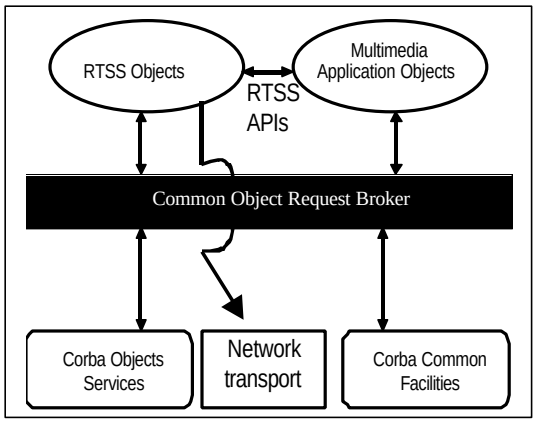
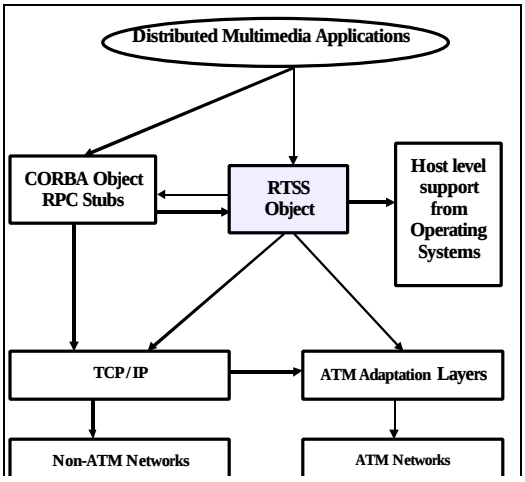


Figure 1 Interaction of RTSS with other object components

Figure 2 depicts the interaction of RTSS with other networking components. The arrow shows the direction of a request; the results of the requests flowing in the opposite direction are not shown. The design approach of RTSS is basically to separate the control and signaling from the



actual data transfer. To this end, we reserve CORBA mainly for non-time-critical signaling and connection management such as registration, locating and binding of services, connection establishment and management and control of real-time data flows.

However, the communication supports for real-time streams are provided by RTSS which has direct access to the transport stacks. This will by pass the processing overheads of the off-the-shelf CORBA (Orbix in our case) and at the same time exploit the network QOS supports (if available). It should be pointed out that applications that are not time-critical can continue to be implemented and operated entirely in the CORBA way within the RTSS environment.

Figure 3 shows the major functional components of RTSS. The 'RTSS Kernel' is the core in our design which consists of a scheduler, an admission control and connection manager and a RTSS stream control manager. A brief description of each component is given below.

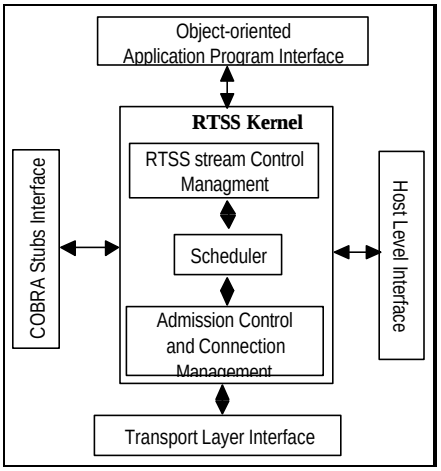


Figure 3: Major Functional Modules in RTSS

- *Object-Oriented Application Program Interface:* Application programs request RTSS services through this interface. The interface may up-call applications' real-time data handlers (see Section 4.1) to generate or to receive real-time data.
- *COBRA Stubs Interface:* RTSS receives a request of service from multimedia application objects as an up-call through this interface. RTSS may also request CORBA for a service provided by other objects through this interface as and when a need arises.
- *Transport Layer Interface:* RTSS provides an abstraction over popular transport protocols, namely ATM's AAL-3/4 and AAL-5, and TCP/IP and UDP/IP. It invokes this interface to establish a connection and to receive and send real-time data to and from RTSS ports. To send control information, however, it exploits the facilities provided by CORBA.
- *Host Level Interface:* This is the module which communicates with the host operating system by means of system calls or library subroutines provided. It is also responsible for creating, scheduling, and managing real-time threads which handle continuous data streams.
- *Scheduler:* The scheduler is responsible for monitoring, controlling and coordinating the activities within RTSS. It invokes the admission control and connection manager to allocate the necessary host and network level resources. It is also responsible for creating real-time threads and assigning and adjusting priorities among these threads. To control a stream (such as to pause or to resume), the scheduler calls the RTSS stream control manager. Details of the scheduling algorithm and its implementation will be described later.
- *Admission Control & QOS Manager:* The responsibility of this module is to interact with the host operating system and to allocate host level resources. These resources include memory buffer and CPU resources. When the resources are successfully allocated, it translates part of the user level QOS parameters into network level QOS parameters and forward them as a request of network resources to the underlying transport protocol. Upon a successful allocation of these resources, it establishes the actual connection. If the requested QOS cannot be satisfied, it refuses the connection and raises the corresponding exception.
- *RTSS Stream Control Manager:* This module provides some convenient ways of control-ling the flow of a real-time stream over a connection. It implements basic stream control services such as start, stop, pause and resume.

4. SYSTEM DESIGN AND IMPLE-MENTATION

4.1 Application Programming interface

RTSS provides services for multimedia application developers through a set of object oriented application programming interfaces. These services include the facilities for creating and deleting RTSS ports (create-port(), delete-port()), attaching handlers to these ports (attach-handler()), connecting ports (connect()), specifying quality of service for a stream, controlling the flow of real-time data streams (start(), stop(), pause(), resume()), query of port's information and a few other miscellaneous interfaces.

It is useful to explain here the up-calls mechanism for supporting the interaction between the RTSS object and the application objects. The related RTSS Port and QOS parameters are also highlighted.

- Up-call and handlers

  The mechanism for supporting interaction between RTSS objects and application objects is through the use of up-calls and handlers. Up-call [21] offers three major benefits: (I) applications do not need to explicitly create and manage threads and buffers, (ii) RTSS decides the best time to up-call (within the QOS constraints) based on local conditions and activities, and (iii) real time programming is considerably simplified as the applications just react to events while communication events are handled by RTSS.

  In an up-call, an application object first attaches its data and QOS handlers to an RTSS port. A connection with a given QOS is then established. The required buffer size is passed by the application to RTSS which allocates the buffer during the connection. RTSS also creates and assigns a thread on which these handlers will be executed. According to the user specified QOS parameters during connection time and the readiness of data, the RTSS decides when to up-call the application to obtain or deliver data. The application writes data to or reads data from the buffer whose address is passed by RTSS during the up-call.

  A *RTSS_data_handler* has two arguments - the size and address of the RTSS internal buffer. In the source side, the application copies the data it has generated into the buffer and sets buffer_size as the length of the effective data. RTSS pushes the data to its stream connection. In the sink side, RTSS pumps the data from the connection, copies them into the buffer, sets buffer_size and up-calls the application. The sink application finally manipulates the data.

  A *RTSS_qos_handler* is similar to the RTSS_data_handler in the way that it is defined and attached to RTSS ports. The use of RTSS_qos_handler is not compulsory and is not up-called periodically but asynchronously to notify the corresponding application object that the QOS specified during connection setup has been violated. If a QOS handler has not been attached, the application will not be notified of such violation. It is also possible for application objects to insert codes in the handler to take an appropriate action for such violation. In this way applications are not only notified of a QOS degradation or violation but are also given an opportunity to make a decision whether to terminate the connection or to take other actions.

  RTSS_qos_handler has only RTSS_qos as an argument, which stores QOS parameters presently being honored. If necessary, the application object can examine these parameters to make a decision on what action to take in the event of a violation.

- RTSS Port

  RTSS ports are communication end-points for the real-time data traffic. A RTSS port has a globally unique name string (RTSS_name) and can be bound from anywhere in the CORBA distributed systems using this name. Each RTSS port has an associated quality of service including internal buffer and scheduling characteristics; and addresses of the application object's data, and optionally the QOS handlers.

  A RTSS port can be one of the two predefined types -- sending or receiving. A sending port gets real-time data from the source multimedia application object such as microphone and pushes the data to the network. The receiving port, on the other end, reads the data from the network and delivers them to the destination application object such as speaker.

- QOS Parameters

Each RTSS port has an associated QOS and is specified in the following structure:

```
typed struct {
RTSS_proto protocol;      "the actual transport protocol to use - TCP, UDP, AAL-3/4, AAL-5"
short server_sap;         "the port number that is needed to established connection between
                           the recei-ving and sending RTSS ports"
short buffer_size;        "the size of internal buffer for a RTSS port"
short buffer_rate;        "source data generation rate in units of buffer per second. The
                           same rate is expected at the receiver""
short actpp*;             "approximate computation time per period in ms estimated and
                           supplied by an application to RTSS"
boolean reliable;         "no data, including those that are late or incomplete, shall be
                           discarded if it is true"
boolean isochorono;       "up call application periodically at the rate as specified by the buffer
                           rate, else up-call as soon as possible".
Short jitters;            "the permissible variation (early or late) in buffer delivery time from
                           the ideal periodic delivery time as specified by the buffer rate"
short latency;            "the maximum tolerable end-to-end delay.  Set as 1/buffer rate"
short error_rate;         "the max. no. of allowable buffer losses per 10,000 buffer"
 } RTSS_qos
```

For details of the design of the programming interfaces and their operations, please refer to Appendix A. Figure 6 presents a simple example to illustrate how the APIs described above can be used to effectively write a simple client program, which connects a microphone object to a speaker object.

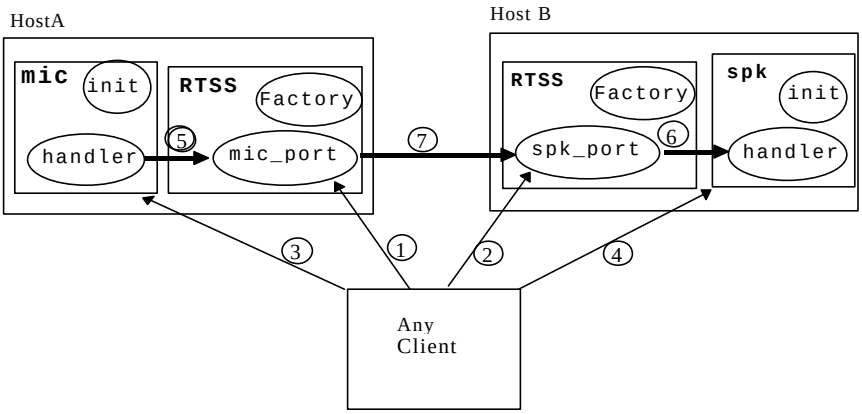


Figure 6: An Example of Microphone-Speaker Connection via RTSS

The client program first (1) creates a microphone RTSS_port (mic_port) in the host where audio is to

```
main() {
   1. mic_port = RTSS::create_port (<mic-name>, RT_WRITE,
      audioqos);
   2. spk_port = RTSS::create_port (<spk-name>, RT_READ,
      audioqos);
   3. mic = microphone::_ bind();
   4. spk = speaker::_bind();
   5. mic->attach_rtss(<spk-name>);
   6. spk->attach_rtss(<mic-name>);
   7. RTSS::connect(mic_port, spk_port);
   8. mic_port->start();
   9. spk_port->start();
   10. .....
   11. mic_port->pause();
   12. mic_port->resume();
   13. ......
   14. mic_port->stop();
   15. spk_port->stop();
```

be captured, and a speaker RTSS_port (spk_port) (2) where the captured audio is to be played. It also binds the actual speaker object (spk)(3), and microphone object (mic) (4) in the respective hosts. It then attaches microphone data handler and speaker data handler to the mic_port (5) and spk_port (6) respectively, and finally connects mic_port to spk_port (7). The mic_port and spk_port can now be started with start(). The ports can then be paused, resumed, and stopped by applying the corresponding operations. At the end of the session, the ports are deleted with delete(). A sequence and relationship of these calls have been shown in the pseudo code below.

4.2 RTSS Scheduler

Host scheduling is one of the most important tasks in RTSS as it handles multiple stream connections with different timing requirements. It is thus important for RTSS to keep track of the timing requirements and schedule the streams in such a way that all of them can meet their deadlines. This is implemented as follows: each RTSS stream has two end points - source and sink. These end points are also known as RTSS ports. Each RTSS port has associated QOS parameters and based on these parameters, a real-time thread based on the rate monotonic fixed-priority scheduling policy is created and attached to this port. This built-in scheduler periodic thread, created once during connection time, and endures until the stream is deleted. The thread runs the data handler attached to fetch and deliver data. It also runs the appropriate protocol processing code associated with the RTSS port to transfer the real-time data. This thread also monitors itself to examine whether it has been scheduled to meet its real-time requirements. If the thread has missed several deadlines in the immediate past it invokes application's quality of service handler to notify the application that QOS specified has been consistently violated. If no quality of service handler has been attached to the port, it cannot notify the application of such violations and the application cannot take any action against such violations at all. In this situation, RTSS adjusts the period (increments the period by 10 ms, in our implementation, and thereby making it less frequent) hoping no other deadlines will be missed in the future.

Our real-time thread is adaptive for it learns from its previous experiences. It fine tunes the computation time as the connection runs; it adjusts the period of the application if it is too frequent to meet the deadline specified because of overload conditions. The detailed implementation of the RTSS real-time thread in Solaris environment is described in [22].

Though the rate monotonic (RM) algorithm (and earliest deadline (ED) algorithm) is simpler to implement and is optimal under normal load condition, its performance is known to degrade under overload condition. To overcome this limitation, we propose a two-level scheduling scheme to meet the dynamic needs of multimedia applications under different CPU load conditions. At the top level, a time slice is given to each independent stream connection. The duration of the time slice is given based on the requested QOS and the current CPU load condition. Within each stream, the scheduling policy is more complex. Each stream is further divided into a sequence of operations. Each operation has a frequency parameter associated with it. This frequency parameter specifies the number of times it must be executed within a pre-defined interval of time. This concept is useful for multimedia applications. For example, in a video-conferencing application, while video frames can be discarded during overload conditions, dropping too many consecutive video frames is undesirable and totally defeats the purpose of setting up a video-conferencing session in the first place. Instead, we would prefer to keep, say 1 frame out of every 4 frames to maintain some kind of continuous motion. To achieve this, we associate a frequency parameter, $f \in [0,1]$ with each operation. Operation with frequency $f = 1$ are mandatory operation and must be executed at every period. Operations with $f = 0.5$ have to be executed every other period and so on. Before an operation is presented to the scheduler for execution, it must first go through a filtering agent. The filtering agent decides, based on the current CPU load condition, whether an operation should be executed or discarded. Those operations that pass the filtering agent are then scheduled for execution according to the rate monotonic (RM) algorithm. In this way, a limit is placed on the demand of CPU by preventing unimportant operations from flooding the execution heap thus allowing the RM algorithm to work at its optimal level. Currently, the scheme has been implemented on a stand-alone host and experiments have been carried out to show that this scheme allows the user to fine-tune his/her QOS requirements dynamically as demonstrated in Section 5.2. Further work is required to integrate this scheme into the distributed environment of RTSS.

4.3 Admission Control

The scheduler works closely with the admission control algorithm to test the availability of CPU, memory and network resources during connection time and refuses a connection if the resources available are not sufficient to meet the application's require-ments. Upon the successful connection establishment, these resources are reserved on behalf of the stream and are used throughout the connection time. Resources are finally released when the connection is terminated. The following subsections describe our QOS translation and admission control techniques.

4.3.1 The CPU resource

For the CPU admission control purposes, this is done at two-level: admission control at the stream level (inter-stream) and admission control at the operation level (intra-stream). Intra-stream admission control is handled by the scheduler as discussed in previous section. In this section, we describe the inter-stream admission control. First, we need to derive the period (Figure 7) data and the computation time for each stream based on the user specified RTSS_qos.

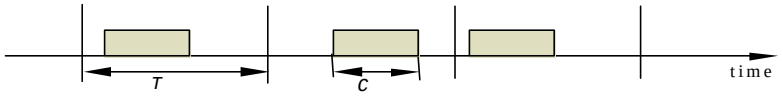


Figure 7. Processing of a stream as a periodical task

The period is simply the reciprocal of buffer_rate specified in RTSS_qos structure and the initial computation time is the actpp (approximate computation time per period) in the same structure, i.e.,

$T_i = \frac{1000}{buffer\_rate}$ (milliseconds), and $C_i = actpp_i$ (milli-seconds).

Note that actpp in practice is the sum of the approximate execution time of the data handler and the transport protocol code associated with that port. During the connection setup, a user just gives a rough number and this value is updated (through actual built-in measurement) as the connection runs.

To keep track the current CPU reservations within a host for each stream, we have a linked-list which links all RTSS stream connections within the system. The head of the list is globally unique. Traversing through this list, the admission control algorithm calculates the sum of the previous CPU reservations. If the sum of the previous CPU reservation and CPU requirement of the new connection is less than some threshold, say 0.88 (assuming RM algorithm is used), we say that the system has enough CPU resource to support the new connection.

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 0.88$$

In this case, the connection will be accepted.

4.3.2 The ATM Network Resources

The ATM network resource (bandwidth) is reserved for each connection. The peak bandwidth is the maximum (burst) rate at which the source produces data, measured in kilobits per second. The mean bandwidth is the average bandwidth expected over the lifetime of the connection, also measured in kilobits per second. The mean burst length is the average size in kilo-bits of packets sent at the peak bandwidth.

The network bandwidth requirement is computed from buffer_rate and buffer_size of QOS parameters. Network bandwidth for the I^{th} thread, in Kilobit per second (Kbps), can be expressed as:

$B_I = (8 \times buffer\_size_I \times buffer\_rate_I)/1024$

This is the maximum bandwidth that a thread can achieve. For constant bit rate applications this is also the average and peak bandwidth requirement. For variable bit rate applications, the average bandwidth depends on the rate of change of scenery on the capture window. The mean burst length for the I^{th} thread (mbl_I) - the average size in kilobits of packets sent at peak bandwidth - can be derived as:

$mbl_I = (8 \times buffer\_size_I)/1024 + 1$

To test the availability of the network resources we supply these parameters, during connection establishment, to the ATM network through ATM Adaptation Layer Interfaces. The ATM network allows the application to specify required network resources for a particular connection. These resources

are reserved on behalf of the application dedicated to that flow and are used throughout the connection time. The resource is released when the connection is terminated. If the network resources available at the time when it receives the request are not sufficient to meet the requirement, the network refuses the connection. At present, the ATM module (based on Fore Systems ATM switch) only allows the application to specify peak bandwidth, mean bandwidth and mean burst requirements.

Since TCP/IP protocol stack and the underneath conventional router-based net-works do not allow users to allocate resources to a particular connection at present time, we cannot reserve the network resource for a particular stream connection if the application chooses to run over these transport protocols. If transport protocol is specified as RTSS_tcp or RTSS_udp, we skip the network resource requirement translation and reservation processes described above and adopt best-effort strategy instead. Nevertheless, router venders are working hard to provide resource reservation mechanism in router-based network.

4.3.3 The Memory Resources

The memory requirement for the a RTSS connection at a host is the buffer_size provided by the user. This is the buffer that is associated with the RTSS_port. At the connection time RTSS allocates the required buffer dedicated for the stream. If it fails to allocate the memory because of lack of system memory, it refuses the connection.

5. PERFORMANCE EVALUATION

We have performed a few experiments to evaluate the feasibility and efficiency of RTSS design and implementation. First, a simple video capture-playback application is created to test the performance of the two-level scheduling scheme in a standalone computer. Then, we carried out a test to measure the maximum end-to-end throughput between two RTSS applications over an ATM network with the integrated RM based host scheduler. The results obtained are then compare with the throughput achieved entirely using Orbix, in exactly the same environment. Third, we develop a video conferencing software as an RTSS application and observe its performance in term of buffer rate in the presence of a CPU-intensive conventional applications. Finally, we test the effectiveness of our admission control algorithm with multiple connections of a stored video stream application. The objective of the experiments is to demonstrate the worthiness of augmenting Orbix with RTSS to support multimedia applications.

5.1 Experiment Environment

It consists of: one standalone Silicon Graphics Indy workstation, two SPARCstation-20s connected to the FORE ASX-200 BX ATM switch through FORE's sba-200e0 ATM adapter card (ForeThought 4.0.0 ATM driver) which supports 155 Mbits/sec (Mbps), and the physical media used to connect the ATM Adapter and ATM switch at OC3 rate. The maximum transmission unit (MTU) for ATM adapter is 9184 bytes. One of the SPARCstation-20s has a CPU running at a clock speed of 60 MHz and the other has a CPU with clock speed of 50 MHz. Each of them has 32 Mbytes of RAM and is running SunOS 5.4 (Solaris 2.4). Both workstations run Orbix 1.3.4 MT.

5.2 Host Scheduling Performance

A simple video capture-playback application, written on a stand-alone Indy workstation, is created to test the performance of the two-level host scheduling scheme under different CPU load conditions. The application captures a 240x180 video image at 10 frames per second and produces three display windows. Two windows display the captured image (one is called the primary window and the other called the secondary window) while the third displays the image after it has been texture-mapped onto a rotating b-spline. All the experiments begin with the application running under normal system load. At time = 3, we overload the system such that some form of degradation from the application is necessary. At time = 24, we restore the system load to normal. This variation of system load is achieved by changing the size of the window containing texture-mapped image. *Our experiment objective is to maintain the primary window to be updated at 10 fps (equal to the rate of capture) while that of the secondary window and the texture-mapped window are to be updated whenever possible.* Figure 8 shows the frame rate of the tasks plotted with respect to time under the normal Unix OS scheduling.

Capture represents the video capture task. Disp1 represents the primary display, Disp2 represents the secondary display, and DispT represents the textured display. When the application is overloaded, the frame rate of all tasks drop and linger at around 5-6 fps. This is not satisfactory since the user already specifies that Disp1 is to be updated at as near 10 fps as possible whereas Disp2 and DispT are secondary considerations. In other words, an acceptable graceful degradation of QOS is to keep Disp1 at as high an update rate as possible at the expense of Disp2 and DispT. This is realized through our scheduler by assigning different frequencies to the tasks. The primary window display had f = 1.0, the secondary window display had f = 0.7 while the textured window display had f = 0.5.

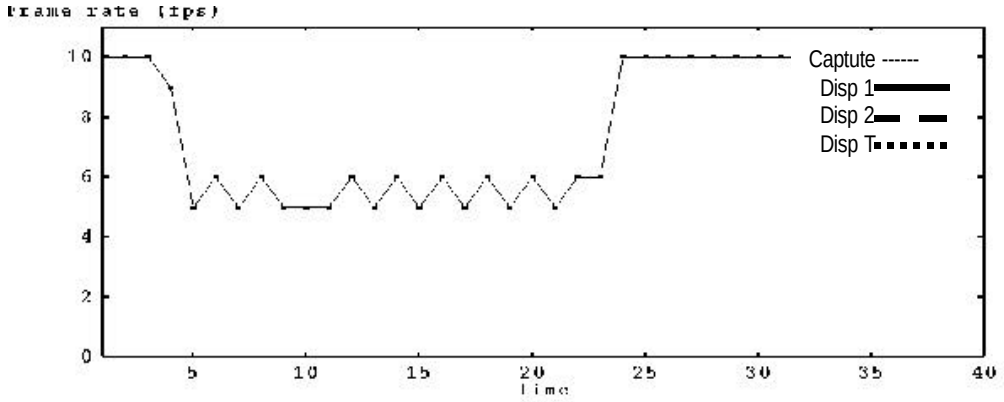


Figure 8. Frame Rate Performance without Frequency-based Support (all curves coincided)

Figure 9 shows that all the tasks begin execution at their desired frame rates. When the CPU is overloaded, the frame rates of all tasks drop rapidly until they eventually stabilizes at some sub-optimal rate. Even at this sub-optimal level, the frame rate of Disp1 is kept as near to 10fps as possible. Once stable, the scheduling mechanism then tries to get the tasks back to their original desired frequencies by raising the frame rate slowly. We see that at time = 15, the application has regained the desired executions for both the primary and secondary window display. However, as it tries to increase the textured window display at time = 17, this once again overloads the system and measures must be taken to restore stability. Once the overload condition is removed (at time=24), the textured window display slowly climbs back to its desired frequency again.

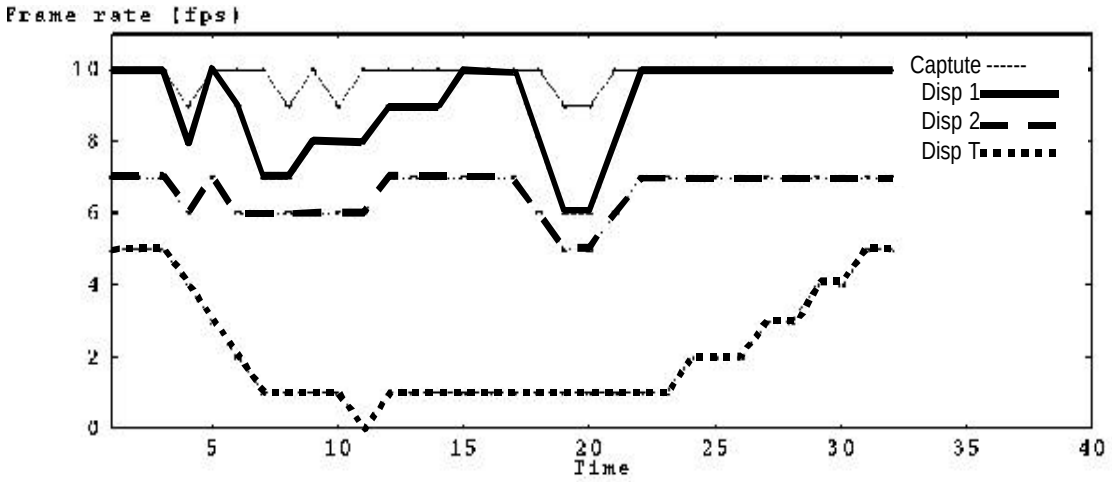


Figure 9. Frame Rate Performance with Frequency-based Support

From this experiment, we see that under overload conditions, the host computer is able to adjust dynamically and has greater control over the timing requirement of individual tasks. For more critical task (Disp1), the performance degradation will be less at the expense of the non-critical tasks (Disp2, DispT). In so doing, we give greater control to the user to fine-tune his/her QOS specification dynamically.

5.3 Throughput Measurement

In this experiment, we generate a stream of 8-bit data (octet) in one system and send it to the other system by means of different techniques supported by RTSS and Orbix. The host scheduling algorithm used in this distributed environment is the normal RM algorithm.

To measure the Orbix throughput, we implemented a pair of simple CORBA server and client. The server receives a sequence of octets as its input from the client and keeps the record of the sum of the octets received during different invocations. The client initializes the necessary environment, sets the timer and then repeatedly invokes the method implemented at server with the fixed size user data. The method has been defined as "oneway". The client sends the data continuously for 10 minutes and inquires the total number of octets received by the server in this period of time.

Let x be the sum of octets received by the server in 10 minutes. The throughput *(T)* can be determined as:

$$T = (x \times 8) / (600 \times 2^{20}) \quad \text{(Mbps)}$$

To measure the RTSS throughput for different transport protocols, we implemented a RTSS application program which sends and receives data. The sender's buffer size is controlled from the buffer_size parameter and the sending and receiving ports were created as non-isochronous to maximize the throughput.

We performed different tests for sender's data buffer size varies from 125 bytes to 128 Kbytes. The sender and receiver socket queue sizes, for all these tests, were set to 64 Kbytes. This size is also the maximum possible socket queue size in SunOS 5.4 and yields the best throughput result. While measuring ATM throughput, we took two more readings -- one at MTU and one at 12000 bytes -- to show the effect of MTU over the performance. The peak and mean bandwidths for AAL-3/4 and AAL-5 were set to the maximum.

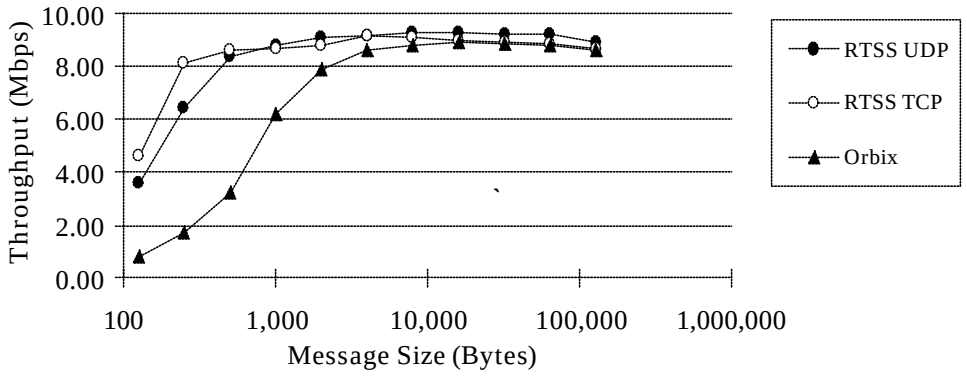


Figure 10: Ethernet Results

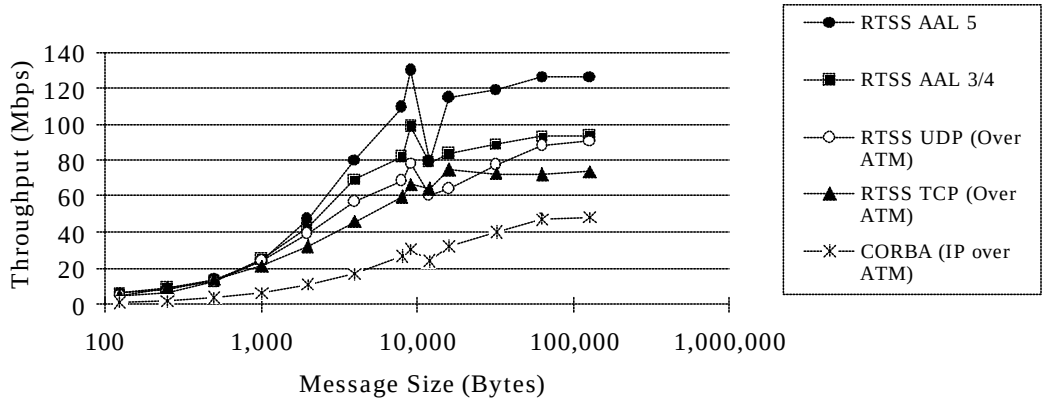


Figure 11: ATM Results

Results obtained while running these tests over an Ethernet are shown in Figure 10; Figure 11 shows the result of similar experiments for the local ATM network. The following observations can be made:

- The Ethernet throughput is stable and has small variations compared with ATM throughputs. Ethernet UDP, as one might expect, performs slightly better than TCP. In our experiment, data loss was not significant.
- The maximum throughput with AAL-3/4 and AAL-5 can be achieved when message size is equal to MTU (Figure 11). In our experiment, at the optimal situation, we could achieve as high as 130 Mbps and 98 Mbps while using AAL5 and AAL-3/4 respectively. The maximum theoretical upper bounds of the user data throughput for AAL-5 and AAL-3/4 with physical line speed 155 Mbps (OC3) are 135.16 Mbps and 124.16 Mbps respectively [23].
- When message size is slightly greater than MTU, the throughput drops considerably as the message needs to be fragmented at the sender and reassembled at the receiver (Figure 11). Depending on the actual implementation, the second packet is usually small and thereby causes further lowering of the throughput rate.
- In ATM local network, Orbix throughput was consistently lower than the rest. For the large message, the best Orbix throughput was roughly 38 percent of the AAL-5, 51 percent of AAL-3/4 and 65 percent of TCP. These observations indicate the worthiness of augmenting Orbix with RTSS.

5.4 Video Conference Using RTSS

We developed a simple video conferencing software as an RTSS application and run the application in the environment as described.

Table 1 shows the results obtained when running the video conferencing software in the real-time mode and ordinary mode. In both modes, we launch, prior to the running of video conferencing a CPU intensive background job which would consume most all of the CPU time if there were no other active jobs in the systems. The requested video and audio buffer rates were 30 and 25 buffers per second respectively and CellB technique was used to compress the video frame. The audio was recorded in μ-law format (8-bit precision) at a sampling rate of 16 kHz. The required audio throughput, therefore, was 128 Kbps. As for the Video, CellB is a variable bit rate compression technique; the size of compressed frame generated at the source (i.e., the throughput) depends on the movement of the image. In both cases we tried to maintain similar movements of the image so as to keep the environment as consistent as possible. The conference session for each test was run for over an hour.

Table 1 shows the buffer rates and throughputs achieved for the different streams in both the modes.

	Non Real-Time Mode				Real-Time Mode			
	Sending		Receiving		Sending		Receiving	
	Throughput (Kbps)	Buffer Rate(fps)	Throughput (Kbps)	Buffer Rate (fps)	Throughput (Kbps)	Buffer Rate (fps)	Throughput (Kbps)	Buffer Rate (fps)
Video1	713.45	19.80	698.71	19.42	1,181.89	29.83	1,179.62	29.77
Video2	967.10	21.59	919.00	20.50	1,404.35	29.86	1,399.61	29.75
Audio1	120.59	23.90	108.41	21.80	127.73	24.99	127.63	24.97
Audio2	123.68	22.47	117.79	21.30	127.60	25.00	127.39	24.96

Table 1: Real-Time mode Vs. Non Real-Time mode of RTSS

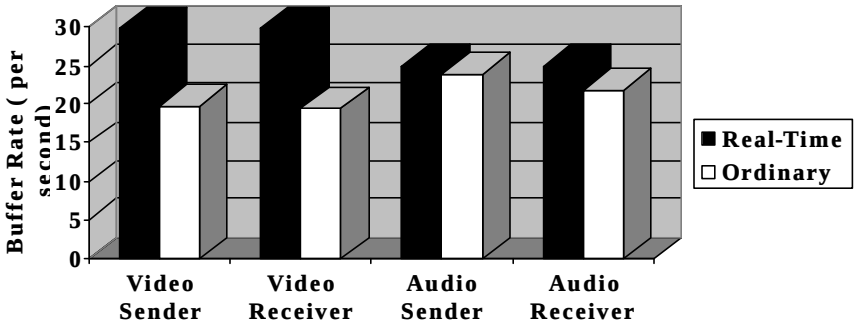


Figure 12: Buffer Rate achieved in Real-Time mode Vs. Non Real-Time mode of RTSS

Figure 12 depicts the buffer rate successfully received by different streams in one of the two hosts involving the video conference in both real-time and ordinary modes.

In the real-time mode, all the streams run in real-time class with different priorities assigned by RTSS. As we can see in the graph, all four streams successfully met the requested QOS, they were scheduled in such a way that they did not miss their deadlines. In non real-time mode, on the other hand, as shown in the graph, they could not satisfy the requested QOS.

5.5 Admission Control Experiment

To test the effect of the admission control algorithm, we successively increase the number of video stream connections between two end-hosts - the source and the sink - and measure the aggregate play-out buffer rate as recorded at the sink. A file of CellB compressed video image (captured at 30 frames per second of live video) was stored in the source's hard disk as CIS (Compressed Image Sequence) format. The CIS format video frame is decoded in software at the sink which requires an average of 9 ms to process a video frame. This same file was used to perform all the tests. With the admission control algorithm in effect, the system admitted limited number of stream con-nections. We repeated the experiments for the case without the admission control. The results we observed are shown in Figure 13.

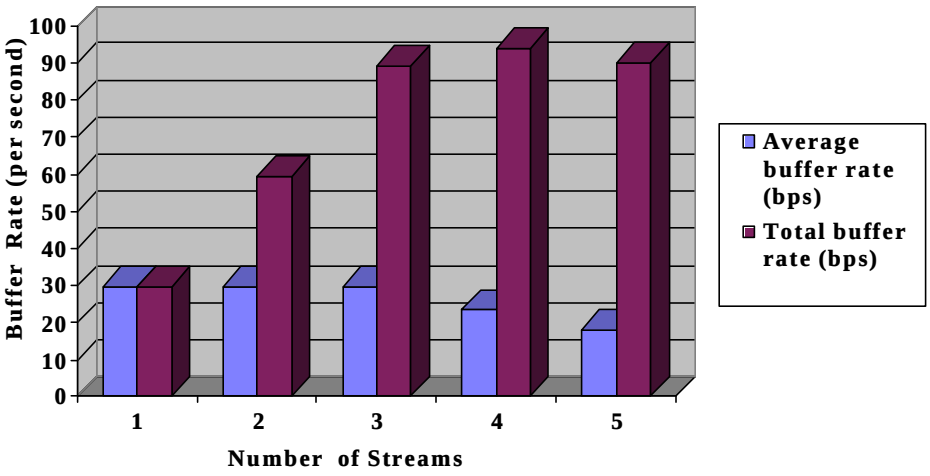


Figure 13: Admission control test

The experiment shows that all the first three streams successfully met the specified requirements - the average buffer rate was maintained at 30 buffers / second. The system did not allow the admission of the 4th stream as it would exceed the 88% of the CPU resource limit (each stream required 27% of the CPU resources as observed). However, without admission control, the addition of the fourth stream has lowered the average buffer rate. Further increase of the number of streams aggravate the situation (Figure 13).

6. CONCLUSIONS

We have proposed, designed and implemented a Real-time Stream Service in Orbix-MT and Solaris environment. The scheme is intended to provide an integrated QOS support for distributed multimedia applications over ATM networks, as well as over TCP/IP networks without network QOS support. It uses the "CORBA channel" for control, signaling and management of real-time stream while by-passing the heavy CORBA stack via a separate but direct data transport channel for the actual transfer of data. The extra complexity of network programming and QOS support are shielded from the application programmers through the use of RTSS APIs.

Results of our performance measurements show that RTSS does not compromise the system performance and is able to meet our design objectives. RTSS maximum throughput rate for memory-memory transfer is superior than when operated entirely over the full Orbix stacks. Its admission control and real-time host scheduling in ensuring quality of service are also demonstrated through experiments. It is worth nothing that these end-to-end QOS enforcement mechanisms have not been addressed in the OMG A/V streams specification.

The feasibility of RTSS for the development of multimedia applications involving continuous real-time data stream was demonstrated through a videoconference application and a file transfer application that ran over a local ATM network and inter-connected Ethernets. In addition, we have shown that the proposed frequency-based host scheduling algorithm is able to give user greater control in fine-tuning his/her QOS specification dynamically under different CPU load conditions. Our next task is to integrate this proposed host scheduling algorithm in the distributed environment and to experiment with fine-tuning the QOS specifications not just under different CPU load conditions, but also under various network load conditions. We are also interested to extend our work on multipoint streams over a multipoint-to-multipoint ATM connection service.

REFERENCES

1. H. Tokuda, "Operating System Support for Continuous Media Application," Chapter 8, book titled, "Multimedia Systems", Edited by John F. Koegel Buford, Addison Wesley Publishing House, 1994.

2. The Object Management Group, The Common Object Request Broker: Architecture and specification, Revision 2.0, 1996.
3. D. C. Schmidt, T. Harrison and E. Al-Shaer, “Object-Oirented Compo-nents for High-Speed Network Programming,” In Proceedings of the 1st Conference on Object-Oriented Technologies, Monterey, CA, June 1995. USENIX.
4. Object Management Group, Control and management of A/V streams specification, OMG Document telecom/97-05-07 ed., Oct 1997
5. Sapkota, BS, H K Pung, LH Ngoh & WC Wong, "A CORBA-based real-time stream service for ATM networks"; International Conference On Multimedia Computing and Systems, ICMCS97, June 3-6, 1997, Chateau Laurier, Ottawa, Ontario, Canada, pg 648-650.
6. D. C. Schmidt, "The Reactor: An Object-Oriented interface for Event-Driven UNIX I/O Multiplexing (Part 1 of 2)," C+ Report, Vol. 5, February 1993.
7. D. C. Schmidt, "The Object-Oriented Design and Implementation of the Reactor: A C++ Wrapper for UNIX I/O Multiplexing (Part 2 of 2)," C++ Report, Vol. 5, September 1993.
8. RealNetworks, "Realvideo player" www.real.com, 1998
9. Vxtreme, "Vxtreme player", www.microsoft.com/netshow/vxtreme/, 1998
10. L. H. Ngoh, "A Real-time Stream Service with End-to-End QOS Guarantees," An Internal Report, Institute of System Science, 1993
11. A. V. Halteren, P. Leydekkers and H. Korte, "Specification and Realisation of Stream interface for the TINA-DPE," PTT Research, 9700 CD Groningen, The Netherlands.
12. D C Schmidt, D L Levine, Sumedth Mungee, The design of Tao real-time object broker, Computer Communications, vol 21, pp 294-324, Apr, 1998.
13. C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-time Environment," Journal of the Association for Computing Machinery, pp 46-61, February 1973.
14. A. K. Mok and M. L. Dertouzos, “Multiprocessor Scheduling in a Hard Real-time Environment,” In Proceeding of the 7th IEEE Texas Conf. Comput. Symp., pp 5-1:5-12, Nov 1978.
15. T. W. Kuo and A. K. Mok, “Load Adjustment in Adaptive Real-time Systems,” In Proceedings of the IEEE Real-Time Systems Sympo-sium, pp 160-170, Dec 1991.
16. J. P Lehoczky, L. Sha and Y. Ding, "The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior,” In Proceedings of the IEEE Real-Time Systems Symposium, pp 166-171, December 1989.
17. J. R. Haritsa, M. Livny, and M. J. Carey, “Earliest Deadline Schedu-ling for Real-time Database Systems,” In Proceedings of the IEEE Real-Time Systems Sym-posium, pp 232-243, Dec 1991.
18. H. Kaneko and J. Stankovic, “A Multimedia Server on the Spring Real-time System,” Technical Report Umass CS 96-11, Department of Computer Science, University of Massachusetts, Amherst, 1996.
19. W. K. Shih and J. W. S. Liu, “On-line Scheduling of Imprecise Computations to Minimize Error,” In Proceedings of the IEEE Real-Time Systems Symposium, pp 280-289, Dec 1992.
20. W. K. Shih, J. W. S. Liu, and J. Y. Chung, “Fast Algorithms for Scheduling Imprecise Computat-ions,” In Proceedings of the IEEE Real-Time Systems Symposium, pp12-21, Dec 1989.
21. Coulson, G., Campbell, A., Robin P., Blair, G., Papathomas, M. and Hutchison, D., "The Designed of a QOS Controlled ATM Based Communications Systems in Chorus", An Internal Report No. MPG-94-05, Department of Computing, Lancaster University, UK, 1994.
22. BHAWANI S. SAPKOTA, A CORBA-based real-time stream service (RTSS) for atm networks, M.Sc thesis, National University of Singapore, 1997.
23. T. Luckenbach, R. Ruppelet and F. Schulz, "Performance Experiments within Local ATM Networks," GMD-FOKUS (Berlin, D)
24. IONA, "IONA Orbix MX", www.iona.com, 1998
25. Sumedh Mungee, Nagarajan Surendran, D C Schmidt, The design and performance of a CORBA Audio/Video streaming service, Washington University technical report #WUCS-98-1

Appendix A

A.1 Creating and deleting an RTSS port

An application creates an RTSS port using the following interface: *RTSS_port RTSS::create_port(RTSS_name, RTSS_type, RTSS_qos).*

This interface first binds the RTSS factory in the node whose address given in RTSS_name parameter. It then makes a remote invocation on the RTSS factory to create a new RTSS port of type RTSS_type (either read or write). While creating the new RTSS port, it performs the admission control to check whether the system has enough processing and memory resources to provide the QOS as specified in the RTSS_qos parameters. It then translates the user specified QOS parameters to the low level network parameters. The actual allocation of the network resources will be done later during the connection time. The call fails if the requested RTSS_qos can not be met.

A RTSS object can be deleted through the use of *char RTSS::delete_port(RTSS_port)* that also releases all the allocated resources.

A.2 Attaching handlers

The following two interfaces are povided for attchment of data and QOS handlers to a given RTSS_port:

boolean RTSS::attach_handler(RTSS_port, RTSS_data_handler);
boolean RTSS::attach_qos_handler(RTSS_port, RTSS_qos_handler);

The given handler will be attached to the specified RTSS_port when the interface calls are successfully invoked.

As the address of the application data handler is only meaningful within a given address space, a RTSS port and the associated handlers (data or QOS) are therefore sharing the same address space. Hence, only objects residing in the same CORBA server as RTSS objects can invoke the above two interfaces.

A.3 Actual Connection

When a read (receive) port and a write (send) port are created and the necessary handlers are attached, they are ready for a connection. The *char RTSS::connect(RTSS_port, RTSS_port)* interface is called to make the actual connection.

Connect( ) has two RTSS ports - one for RTSS_read and the other for RTSS_write - as parameters. If successfully invoked, it returns a value greater than zero, otherwise it raises an exceptional flag and returns a zero.

The two ports must be created with the same RTSS_proto type (protocol type), otherwise connect() would raise a connection refused exception. In general, these two ports should also maintain the same buffer_size, buffer_rate and RTSS_qos. The actual protocol used to make the connection depends on the parameters specified in RTSS_qos structure.

The connection is an unidirectional stream, where data flow occurs only from source port to the sink port. As an illustration in Figure 4, each time a microphone data handler is up-called, it copies the data from audio device to the mic_port's internal buffer. The mic_port then outputs the data to the real-time data stream. On the other end, spk_port obtains the data from the stream and stores them into an internal buffer. The spk_port then up-calls the speaker data handler, which ultimately transfers the media data into the hardware speaker device.

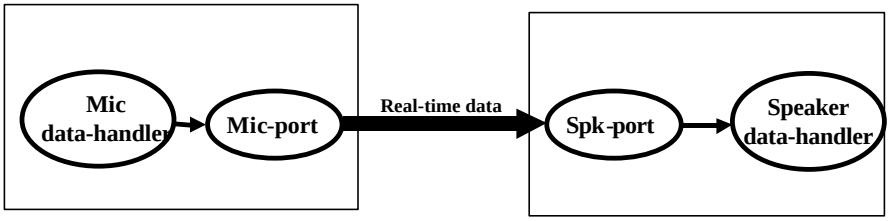


Figure 4: Real-Time data Flow from source port to sink port

A.4 Stream Control in RTSS

RTSS also provides interfaces to control the flow of real time data in a RTSS ports by changing its status. At any given point in time, a RTSS port can have one of the following status:

```
typedef enum {
    RTSS_ready, RTSS_attached, RTSS_connected,
    RTSS_live, RTSS_paused
} RTSS_proto;
```

And, the following interfaces can be applied to a RTSS_port to change its status.

```
RTSS_status RTSS_port->start();
RTSS_status RTSS_port->stop();
RTSS_status RTSS_port->pause();
RTSS_status RTSS_port->resume();
```

We omitted the rewind operation as it is more complex and does not contribute significantly to the value of this work.

Figure 5 illustrates the relationship of the operations and the states of RTSS.

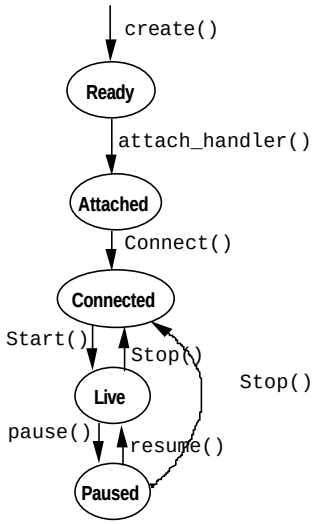


Figure 5: State transition diagram

The operation start() can only be applied to the port which is currently in a connected state. Starting a port basically involves creating a real-time thread on which RTSS runs the handlers and the protocol processing code to receive and send data. Stopping an RTSS port exits the real-time thread and brings the port back to the connected mode. The stop() operation can be applied either to a live port or to a paused port. A port in a live-state can be switched to a pause-state by applying the pause() operation. The operation resume(), on the other hand, performs the opposite action. The operation delete(), as mentioned earlier, can be applied to the port in any of these states.

A.5 Query of RTSS Port's information

There are a number of interfaces that can be used to query information/status of a part:

```
RTSS_position RTSS_port ->get_position();
  "return the no of bytes sent or received so far"
RTSS_status RTSS_port->get_status();
  "return current status of the port"
RTSS_type RTSS_port->get_type();
  "return RTSS_write or RTSS_read"
RTSS_qos RTSS_port->get_QOS();
  "return RTSS_qos structure honored to the port"
string RTSS_port->get_error();
  "return the last error message"
```

A.6 Miscellaneous interface

There are three more interfaces that we have not been covered so far:

```
RTSS_port RTSS::get_port(RTSS_name);
void RTSS::export();
void RTSS::release();
```

The interface get_port() provides a convenient way of binding an existing RTSS port in the distributed system. The input parameter is a string format of RTSS_name. It's functions is similar to but not the same as CORBA's _bind operation (see [2] for detail on _bind()).

Finally, export() constructs and exports an instance of a RTSS factory object while release() deletes an instance of the RTSS object. In a given node, export() should be called prior attempting to connect to the factory while release() is normally called just before the server exits.