# User Level Scheduling of Communicating Real-Time Tasks

*Chia Shen*

MERL - A Mitsubishi Electric Research Lab.
201 Broadway
Cambridge, MA 02139
shen@merl.com

*Oscar González*[*]
*Krithi Ramamritham*[†]
Computer Science Department
University of Massachusetts
Amherst, MA 01003
[ogonzale,krithi]@cs.umass.edu

*Ichiro Mizunuma*
Industrial Electronics and Systems Lab.,
Mitsubishi Electric Corp
8-1-1, Tsukaguchi-honmachi
Amagasaki, Hyogo, 661, Japan.
mizunuma@con.sdl.melco.co.jp

**Abstract**

*Unique challenges are present when one tries to build distributed real-time applications using standard off-the-shelf systems which are in common use but are not necessarily designed specifically for real-time systems. In particular, to realize end-to-end predictability when, say, a process on one node sends data to a process on another node, several issues must be addressed: (1) mapping application real-time requirements into requirements imposed on the system schedulable entities (tasks), (2) ensuring predictable execution of the tasks in the face of possible priority inversions, limited OS level real-time scheduling support, and limited number of priorities, and (3) integrating real-time and non-real-time tasks in the same platform. In this paper, we propose solutions to these challenges. In particular, we present user-level scheduling schemes for communicating tasks. These solutions are practical and are based on simple primitives that can be found in most of today's commonly used operating systems. To validate our design and to examine the feasibility of user-level scheduling in actual systems, we have implemented our solutions in MidART running on PCs with Windows NT operating system over UDP/IP and Fast Ethernet LANs. This paper contributes to further our understanding of how to build real-time systems using commercially available off-the-shelf components.*

[*]The research has been done during this author's internship at MERL.

[†]Research supported in part by the National Science Foundation Grant CDA-9502639.

## 1 Introduction

It is becoming evident that distributed industrial real-time applications, such as process control, factory automation and plant control systems, will move towards using open, standard, commercially available and more general purpose computers, operating systems and networks. For example, more expensive workstations will be replaced with off-the-shelf PCs, and adoption of Windows NT will allow the use of PCs for real-time control. There is also a lot of momentum toward making control networks IP-based [4]. Moreover, Ethernet offers cost and support advantages over industrial fieldbuses as a control and device-level network [3]. On the other hand, more and more desktop applications are also starting to embody real-time elements, such as the presentation of continuous media and the remote control of instruments and devices in distant learning environments. This potential of using general purpose platforms for a very wide spectrum of real-time applications has prompted many contemporary general purpose operating systems to offer some minimal degree of real-time scheduling and programming support.

However, unique challenges are present when one tries to build distributed real-time applications using standard off-the-shelf systems which are in common use but are not necessarily designed specifically for real-time systems. In particular, to realize end-to-end predictability, say, when a process on one node sends data to a process on another node, we must deal with several issues: (1) mapping application real-time requirements into requirements imposed on the system schedulable entities (tasks), (2) ensuring predictable execution of the tasks in the face of possible priority inversions, limited OS level real-time scheduling support, and limited number of priorities, and (3) integrating real-time and non-real-time tasks in the same platform.

In this paper, we address these challenges. In particular, we present user-level scheduling schemes for communicating tasks that are practical and are based on simple primitives that can be found in most of today's commonly used operating systems. The scheduling schemes consist of a combination of server-based execution, rate control of message communication, and a simplified form of dual priority scheduling [6]. Our schemes are designed to allow both real-time and non-real-time tasks to execute on the same platform using open off-the-shelf machines and networks. To predictably schedule real-time activities, it is well-known that the worst case execution time is required. To obtain these times, we provide a profiler together with our scheduler support to extract the execution time and message delay parameters for the server components.

For concreteness, the above schemes for user-level scheduling are designed and implemented in the context of MidART, a real-time network middleware [16, 7, 13]. This middleware provides a distributed real-time application development software package with an easy-to-use programming interface for real-time data acquisition and communication. We present our design, implementation and experimental results for a distributed real-time system realized using PCs with Windows NT operating system over UDP/IP on Fast Ethernet LANs. Our implementation utilizes the guidelines and recommendations developed in our previous experimental work on using Windows NT for real-time applications [15].

The paper is organized as follows. In the next section, we motivate our work further by contrasting it with previous work on the integrated scheduling of real-time and non-real-time scheduling. In Section 3, we analyze the characteristics of the applications that we intend to support and in Section 4, we list the set of features and problems often found in contemporary operating systems. Section 5 presents an overview of MidART. Section 6 presents our user-level scheduling schemes. The implementation details and experimental results are described in Section 7. We conclude the paper in Section 8.

## 2 Related Work and Scope of Research

Much of the previous research on scheduling support for integrating real-time and non-real-time tasks has been done by either (1) extending existing operating systems [8, 17, 14, 10], or (2) virtualizing the underlying hardware to multiplex a real-time kernel with the original operating system [1, 9]. These approaches have certain obvious drawbacks: Approach (1) needs to modify the source of the existing operating system, and approach (2) is difficult to support real-time and non-real-time activities that are closely related, e.g., in the case where there are real-time and non-real-time threads belonging to the same process and need to share an address space. The extended operating systems based on these approaches are not easily accepted or adapted by many vendors, and thus are not readily available to general users. Moreover, most of these research systems have been done on operating systems (e.g., FreeBSD and Solaris Unix) that are from an earlier generation.

The research reported in this paper differs from previous work in aiming at supporting the integration of real-time and non-real-time tasks by developing schemes only at the user level, utilizing the facilities that are present in most of the contemporary operating systems. These facilities include support for real-time scheduling, such as non-degradable priorities, and real-time class threads – support not present in the older generation of these operating systems. For example, Windows NT 4.0 provides a REALTIME class such that threads in this class have non-degradable priorities and their execution has precedence over timesharing threads [5, 15]. It also supports timers with 1 millisecond granularity and periodic callback routines which can be used for real-time periodic tasks.

Real-time tasks require predictable scheduling, execution and completion with bounded variability. In reality, predictability comes in different granularities:

- At one end of the spectrum, the direct physical control of robots, devices and instruments by local loop controllers, embedded controllers and PLCs may require sub-millisecond to one millisecond delay bound. Unfortunately, the effects of nondeterminism present in today's operating systems [11] make it impractical to design user level scheduling to support real-time applications with sub-millisecond precision. These types of applications are better served by real-time kernels such as those described in [1, 9].
- At the other end of this spectrum, a deadline value of one second is good enough for a video on demand file server [2] or database transactions for financial applications. These applications can be directly built with existing operating system primitives as shown in [2].
- In the middle of this spectrum, there is a large

population of real-time applications that can tolerate end-to-end roundtrip delay/response time in the tens to hundreds of milliseconds range such as those described in [13]. These applications often involve the interaction between human operators/users and remote data/image collection, video monitoring, file accessing, and command issuing in order to view videos, monitor and control devices at a distance. Our goal in this research is to further our understanding of how to build real-time systems using commercially available off-the-shelf systems for such applications.

Our specific focus in this paper is on supporting applications in the middle spectrum, specifically, applications where human users need to interact, control, and monitor instruments, devices and facilities in a networked environment. Applications in this spectrum may be considered by some as "soft" real-time, thus can tolerate longer latency/delay caused by the occasional misbehaving of the operating system and the network.

## 3 Application Characteristics and System Support Needed

To design appropriate scheduling schemes, we must first understand the application domain. Below is a characterization of the types of distributed application tasks that we intend to support:

- *Command and control*: Human operators sporadically issue commands which often need immediate data transmission and delay bounded data reception.
- *Video/audio*: Periodic transmission, requiring low jitter display.
- *Device/instrument monitoring*: Sporadic or periodic data collection and transmission, including alarms, and immediate data display upon reception.
- *Trend graph*: Periodic data collection and transmission (perhaps at a coarser granularity), and periodic data display upon reception.
- *Background*: Non-real-time activities such as logging data to disk, reviewing video segments, and sending email. Best effort or soft real-time data transmission and reception.

As one can see, sending and receiving data in various forms is a key component in most of the above application tasks. In essence, each of these tasks can be viewed as consisting of four subtasks:

- A writer, such as a sensor, that generates the data to be sent.
- A sender that sends the data to the destination.
- A receiver that receives the data from the network.
- A reader that reads and uses the data received.

For some application tasks, such as the issuing of a command by an operator, the data needs to be sent to the receiver immediately after it is generated and the data needs to be read immediately after it is received. For other application tasks, such as displaying the video image of a monitored industrial plant, the writer (i.e., the camera) and the reader (i.e., the display function at the operator station) can execute at its own pace based on the requirements of the application. Based on these observations, we have derived the following set of operations that the underlying system and the scheduler must support [16]:
Data Sending Operations:

- *Synchronous Data Send*: Data send operation is triggered by application writes.
- *Asynchronous Data Send*: Data is sent periodically, and the period is independent of the writer's period. Also, the last data item written could be sent or the data could be sent in the order written.

Data Reception Operations:

- *Blocking Read*: Application reads block while awaiting the arrival of a data update message from the writer's node. When the message is received, the reader application is signaled.
- *Non-Blocking Read*: Application reads return the current contents of the communicated data. That is, the reader's application will not be immediately notified upon the arrival of data update messages.

In the rest of the paper, we will focus on solutions to the scheduling of the four common subtask types resulting from the above observations.

## 4 General Characteristics of General Purpose Operating Systems

The features and problems often found in contemporary operating systems, such as Windows NT, IRIX and Solaris, include:
*Features:*

- Preemptive priority-based scheduling and/or round-robin scheduling.
- Non-degradable priorities.
- Mechanisms for priority adjustment (e.g., set priority of a process or thread to a different level).
- Periodic timers to trigger periodic events and release of thread execution.

*Problems:*

- No priority inheritance among processes/threads, and no priority tracking from user threads to system/network protocol stack threads.

  This makes real-time end-to-end scheduling with network communication very difficult. For example, order of execution of socket level calls do not necessarily respect the priorities assigned to the threads which make the socket calls.
- Limited number of priorities.

  This implies that we cannot use a unique priority for all the real-time tasks.
- No specific support for specifying and guaranteeing timing constraints for tasks besides basic priority-based scheduling.

  This implies that we must assure (beforehand) that the priority assignment policies are guaranteed to meet the timing constraints.

Given these features and problems, the research question we focus on is "How can we alleviate the problems using the features?". Note that some operating systems may only have a subset of the problems listed above. Our goal is to come up with solutions that are general enough to be employed on any off-the-shelf systems.

## 5 Data Transfer Support Provided by MidART

MidART supports a real-time application's end-to-end data transfer requirements with a set of easy-to-use communication service programming interfaces. A key service provided by MidART is the Real-Time Channel-Based Reflective Memory (RT-CRM) [16].

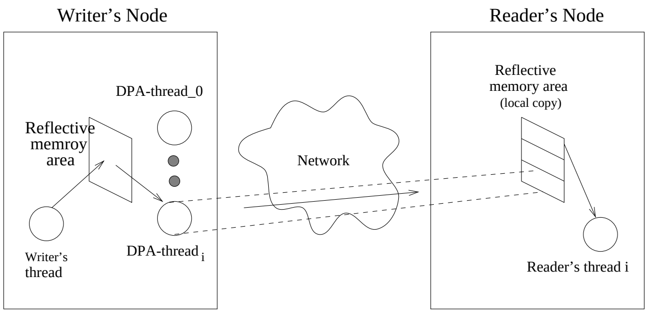


**Figure 1. RT-CRM High Level Architecture**

Figure 1 depicts the high level architecture of RT-CRM. RT-CRM is an association between a writer's memory and a reader's memory on two different nodes in a network with a set of protocols for memory channel establishment and data transfer. A writer has a memory area where it stores its current data (e.g., a PLC stores all the sensor data), while a reader establishes a corresponding memory area on its own local node to receive the data reflected from the writer (e.g., an operator station receives and displays monitoring data). Data reflection is accomplished by a Data Push Agent (DPA) residing on the writer's node and sharing the writer's memory area. This agent represents the reader's QoS and data reflection requirements. A virtual channel is established between the agent and the reader's memory area, through which the writer's data is actively transmitted and written into the reader's local memory area by a Data Receive Agent (DRA). In this architecture, we support the following features:

- A reader memory area may be connected to multiple remote writer memory areas simultaneously. However, at any moment only one writer is permitted to write into the reader's memory area via the associated agent. The selection of the particular writer at any time is done via Selective Channel protocols [13] [1].
- A writer memory area may be connected to many remote reader memory areas simultaneously. Data is pushed to each reader according to their individual requirements.

In particular, given a reflective memory area $ReMA$, since a DPA is a separate thread of control from the writer's application thread, RT-CRM can support (a) *Synchronous* as well as *Asynchronous* data push operations and (b) *Blocking* as well as *Non-Blocking Read* operations.

For more details of RT-CRM, readers are refered to [16]. The uniqueness of MidART lies in the simplicity of services provided and the flexibility of data reflection models. This simplicity leads to ease of understanding and ease of use by application builders, while its flexibility sufficiently serves the needs of the class of real-time applications MidART is designed for. Specifically, with the set of data push and reception operation modes provided by MidART, we can support the application requirements described in Section 3 with many combinations of operation modes. Table 1 lists two possible such combinations – SB (Synchronous Blocking) and AN (Asynchronous Non-blocking).

## 6 Server-based User Level Scheduling

In this section, we will present solutions for the challenges listed in the Introduction, i.e., mapping of application requirements into schedulable entities,

[1] Selective Channels allow applications to dynamically choose the remote node(s) to which data is to be sent or from where data is to be viewed. This is accomplished via a set of channel start and stop protocols, and channel bandwidth resource overbooking schemes.

| Modes | Data Type | Deadline | Application |
|---|---|---|---|
| SB | Sporadic | G | Command |
| AN | Periodic | G | Trend graph |
| SB | Sporadic | G | Plant data |
| AN | Periodic | G | Video/Audio |
| SB | Sporadic | NG | Background |

**Table 1.** S = Synch., A = Async., B = Blocking, N = Non-blocking, G = Deadline Guaranteed, NG = No Guarantee.

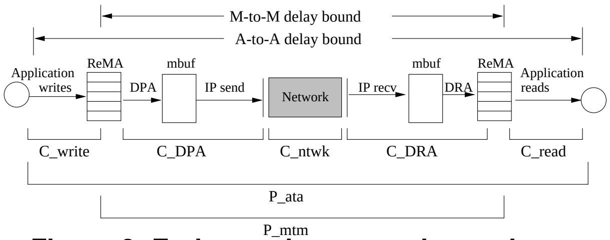


**Figure 2. End-to-end computation and communication.**

end-to-end scheduling with limited number of priorities and accommodating both real-time and non-real-time application tasks in the same system, on commercial off-the-shelf platforms. We first describe how application real-time requirements are mapped into system schedulable entities (tasks). Our mapping scheme accommodates the precedence constraints imposed by the communicating application tasks. Then we proceed to present our user level scheduling method which is a combination of three schemes — server-based execution for task communication, rate controlled uniform size messages and a simplified form of dual priority scheduling [6].

### 6.1 Model Precedence Constrained Tasks with Release Jitter

In general, when applications need to communicate over IP across a LAN, the end-to-end computation and communication entities include the application threads that generate/consume the messages, socket level send and receive, network interface and network transmission. In particular, Figure 2 depicts the tasks involved in an end-to-end scenario[2]. The Application writes and reads are application processes/threads that include MidART library calls M_Read and M_Write [7]. ReMA is the Reflective Memory Area, set up as shared memory between the applications and MidART. DPA and DRA use sockets for communication over UDP/IP. *mbuf* is the memory used by the sockets. Since we have no control over the network interface and network transmission, these two have been merged into the black box called Network. C_x is the worst case computation/communication time of the respective task entity x. So, in our nomenclature, application read/write threads, DPAs, and DRAs are all *tasks* that must be explicitly scheduled.

End-to-end can be classified into application-to-application (A-to-A) , and memory-to-memory (M-to-M) [16]. Some of the application activities, such as command-and-control, require A-to-A delay bound (i.e., deadline) guarantees, whereas others, such as trend graphs, only require M-to-M guarantees. To support A-to-A, we must schedule all the tasks from application writes to application reads as a precedence constrained task graph. On the other hand, to support M-to-M, we can schedule application writes and reads independently, while treating all the rest of the tasks as one precedence constrained task set. Therefore, in Figure 2 P_ata and P_mtm are the periods of an A-to-A task set and a M-to-M task set respectively. Note that by definition, the synchronous data push as well as the blocking read modes impose precedence constraints between the tasks involved, while the asynchronous data push and the non-blocking read modes support independent task representation naturally. In particular, a precedence constraint exists between an application write and DPA for synchronous mode, and between DRA and an application read for Blocking mode.

To enable the end-to-end scheduling (for both A-to-A and M-to-M) on any platform with only priority-based scheduling support, we model precedence constrained tasks as independent periodic tasks with release jitter as follows.

- Let $J_y$ be the jitter of task entity y.
- In the case of A-to-A:

  $J_{write} = 0,\ J_{DPA} = C_{write},$
  $J_{DRA} = J_{DPA} + C_{DPA} + C_{ntwk},$
  $J_{read} = J_{DRA} + C_{DRA}.$

- In the case of M-to-M:

  $J_{write} = 0,\ J_{read} = 0,$
  $J_{DPA} = 0,\ J_{DRA} = C_{DPA} + C_{ntwk}.$

To map application timing requirements into system schedulable components, we take the following approach:

- Application writers and readers specify their periods/delay bounds, synchronous or asynchronous data reception, as well as blocking or non-blocking data retrieval semantics.
- With reference to Table 1, timing requirements are mapped as follows:

[2] This is a simple pictorial representation. The computation time and memory size are not drawn to scale.

(1) SB application: The writer's period is used for all the tasks end-to-end, and the release jitters of tasks are calculated according to the A-to-A case as above. Only if the required deadline is larger than the response time is the application admitted.

(2) AN application: Reader and Writer will have their independent periods and delay bounds. Computation and communication tasks $C_{DPA}$ and $C_{DRA}$ will use reader's period, while their delay bound/deadline $D_x$ is calculated as:

- $D_{DRA} = D_{read} - C_{read}$
- $D_{DPA} = D_{read} - C_{read} - C_{DRA} - C_{ntwk}$

Their respective release jitter will be calculated according to the M-to-M case as described above.

- All the C_x values are derived via a MidART profiler which is described in Section 7.2.

### 6.2 Server-based Execution of Task Communication

Below we show how a "communication server" can be used to handle several problems at once: message sending and receiving at the priority of the sending/receiving process priority, minimizing priority inversion, and dealing with limited operating system priority levels. Our main focus will be the scheduling of the application readers, writers and the DPAs. DRAs are currently executed using a simple event driven model, that is, we do not explicitly account for the priority of the received messages. Our intent is to study whether the receiver will obtain the specified timing requirements, such as periodic data reception, through our rate control and explicit scheduling schemes of message handling on the sender side only. As corroborated by the experimental results in Section 7.3, the solutions described here in the context of DPAs implicitly address the corresponding scheduling problems for message reception by DRAs.

In order to support applications with end-to-end timing constraints, we need to ensure that high priority messages will not be blocked – for an unpredictable amount of time – by a low priority message transmission. MidART decouples applications and their communication needs by using DPAs to carry out the data reflection and DRAs for message reception on behalf of the application threads. The application threads may be assigned priorities that are entirely different from the priorities of their data messages for better system performance. In this case, priority tracking is needed between the DPA and the messages. As we discussed in Section 4, general purpose operating systems do not support such *priority tracking*. To alleviate this problem, we have developed a server-based approach for communication over sockets.

The server is a message transmission manager — all network communication is handled by the server. The server knows the priorities of the applications that are requesting message transmission and schedules the messages (more specifically, the DPAs that push the messages) such that priority tracking is achieved. It is worth noting that whereas many of today's real-time schedulers only schedule with respect to the abstract notion of time, thereby simply acting as "capacity" managers, our server not only schedules the message transfers but also actually carries out the communications on behalf of the applications. This allows us to alleviate the priority inversion problem for message handling.

Specifically, we provide application interfaces to specify the timing and message size requirements of an application. Then the server handles the socket establishment and schedules the message transmission according to the specification. Allowing users to specify the requirements of application components in terms of timing, data and relationships among tasks, such as periods/rates, data size, delay/response bounds, and precedence relations is a much more intuitive and effective proposition than requiring application designers to assign relative priorities to their threads/processes. Then the user specified real-time requirements are translated into unique priority levels to be used only within the server. This approach also alleviates the problem of limited priority levels inherent in a typical operating system. Thus, we assign and manage tasks' priorities explicitly inside our server, while providing the users with the ability to specify task timing, message size and precedence/synchronization constraints.

In summary, the above server based approach is able to minimize the effect of priority inversion, handle priority tracking and manage with limited operating system priority levels, and permit user specified timing requirements.

### 6.3 Integrating Real-Time and Non-Real-Time Tasks

A simplified form of the dual priority scheduling algorithm [6] is used to accommodate real-time and non-real-time tasks in our server. In Section 3, we described five types of application activities. They represent applications with real-time constraints, as well as applications without real-time requirements. Real-time applications demand synchronous(S) or asynchronous(A) data transmission and blocking(B) or nonblocking(N) data reception. In general, the synchronous ones constitute the more urgent time critical tasks, while the asynchronous ones are periodic in nature. The non-real-time activities can be serviced with best effort. Based on these observations, we have developed the following scheme for mapping applications into a set of unique priorities, one for each application type, namely, SB real-time, AN real-time, and non-real-time.

- All the priority assignments fall into four priority bands – $P^{highest} > P^{high} > P^{mid} > P^{low}$. (The specific operating system priority levels these bands occupy depends on the number of priority levels available. In the next section, we discuss how these bands are realized with NT.)
- All tasks of an SB application are assigned to the $P^{highest}$ band and will have priorities according to the deadline monotonic algorithm, That is, within the $P^{highest}$ band, SB tasks have unique priorities, decided based on their deadlines [12]. In many applications, e.g. plant monitoring, there are only one or two such entities in a system.
- All tasks of an AN application with timing constraints will be assigned a priority in the $P^{high}$ band according to the deadline monotonic algorithm.
- All the non-real-time activities are assigned a priority in the $P^{mid}$ band. These priorities can be either assigned according to their requested execution rate, or randomly if they do not specify any rate.

Before we provide details concerning our server's operation, it is necessary to give an overview of the dual priority approach. Basically, it executes real-time tasks such that they will not miss their timing constraints while at the same time improving the responsiveness of the system to non real-time tasks (compared to always giving real-time tasks the higher priority). To this end, each real-time task normally executes with a priority lower than that assigned to non real-time tasks. When the time comes for a real-time task to execute in preference to a non real-time task so that the real-time task will not miss its deadline, its priority is increased to be above that of the non real-time task. This time, called the priority promotion time, $PT_i$ for each invocation of a task $T_i$ with deadline $D_i$ is calculated according to the iterative formula (5) in [6] as follows:

$$\begin{aligned} PT_i &= D_i - R_i && (1)\\ R_i &= w_i + J_i && (2)\\ w_i^{m+1} &= C_i + S^o + \lceil \sum_{j \in hp(i)} \lceil \frac{w_i^m + J_j}{T_j} \rceil C_j && (3) \end{aligned}$$

where iteration starts with $w_i^0 = 0$, and ends when $w_i^{m+1} = w_i^m$, and $R_i$ is the worst case response time. Thus, promotion times take into consideration the release jitter $J_i$ described in Section 6.1 and the one-quantum blocking time $S^o$ due to the transmission of a message from a lower priority task (described in the next section).

In other words, the server executes at the lowest priority $P^{low}$ servicing real-time tasks until one of two events happens – the arrival of a non-real-time task, or the occurrence of the promotion time of a real-time task.

- Upon the arrival of a non-real-time task requesting service, the server's priority will be raised to $P^{mid}$, and will stay at $P^{mid}$ as long as there are still non-real-time tasks to service, or until the promotion time of one or more real-time tasks occurs.
- When the promotion time of one or more real-time tasks occurs, the server's priority is raised to $P^{high}$ or $P^{highest}$ depending on whether the task is an SB task or an AN task. In some sense, $P^{highest}$ can be viewed as lying at the high-end of the $P^{high}$ band. This is the view taken by the implementation described in the next section.

In summary, our server services all the communication tasks according to their respective periodicity, data size and priority. It implements the dual priority algorithm such that each DPA is explicitly scheduled within the server. This is done using four unique operating system supported priorities $P^{highest}$, $P^{high}$, $P^{mid}$, and $P^{low}$.

As we shall see in Section 7, our current implementation uses a simplified form of the above dual priority scheduling approach. Basically, we do not execute a real-time task until its promotion time. An implementation that is faithful to the above description of dual priority scheduling is part of our future work.

### 6.4 Rate Controlled Uniform Size Message

Using the server described above, there is still a possibility of a high priority message being blocked by a lower priority message for the duration of the time to transmit one message. In the worst case, this blocking time can be quite long if message sizes are not bounded. To limit this blocking time to a small quantum, we adopt a uniform message size to allow message transmission with fixed preemption points. This way, a high priority message will be blocked for at most one quantum.

Through a MidART profiler, we identify the optimal message size $S^o$ for any particular network setup. The message size serves as the non-preemptive unit of computation time and transmission time (Henceforth, we use $S^o$ as both the unit message size and the time needed to send and receive a message of this size.). All data to be pushed is divided into messages of this unit size.

# 7 Implementation and Experimental Results

To evaluate our approaches for satisfying end-to-end constraints, we have implemented the server-based dual priority scheduling in MidART on Windows NT 4.0. In this section, we first describe our implementation, and then present our experimental results from this implementation.

## 7.1 Asynchronous DPA Server (ADS)

The ADS depicted in Figure 3 is part of MidART's local server [7]. The ADS is made up of two active threads of control and various objects that facilitate the communication between the two threads and support the user-level scheduling schemes described in Section 6. The two active threads are called *Dispatcher* and *Pusher*. The relevant objects include the *Promoted Push Queue*, the *Push Queue* and the *DPA Server Object*.

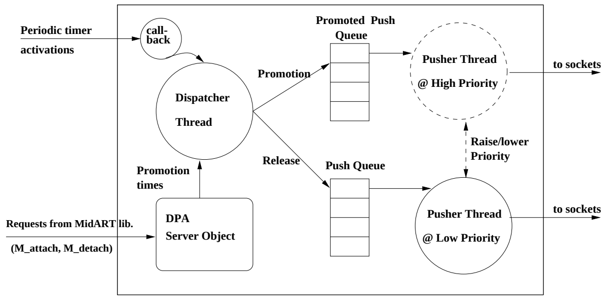


**Figure 3. Dual Priority DPA Server.**

Each agent that pushes the data stored in a ReMA resides either in the *Promoted Push Queue* or in the *Push Queue*. DPAs associated with ReMA entries that have been released (i.e., their current periods have started), but whose promotion time has not arrived yet, reside in the *Push Queue*.

The *Pusher* thread, as the name suggests, executes on behalf of a DPA. It pushes the data associated with a ReMA into the network. It (is scheduled by the system such that it) transmits the data associated with these entries only when there are no soft tasks executing in the middle priority band.

The *Dispatcher* acts as the rate controller and is responsible for moving a DPA into the *Promoted Push Queue* when its promotion time arrives (this involves a simple pointer manipulation). In our implementation, the above operation is equivalent to raising the priority of a DPA thread from $P^{low}$ to $P^{high}$ in Dual Priority scheduling although our approach avoids the actual cost of priority setting in many cases when the server is already in $P^{high}$ and one or more promotion times occur.

This thread uses the Multimedia Timers in periodic mode to promote a DPA in a timely manner. Specifically, the *Dispatcher* runs at the TIME_CRITICAL priority level supported by Windows NT. It blocks waiting for the callback function associated with the timer to indicate the occurrence of the promotion time for a particular DPA. The communication between the callback function and the *Dispatcher* thread uses the message queues provided by the Windows NT operating system.

The *Pusher* works at priorities HIGHEST, NORMAL, and BELOW_NORMAL in the REALTIME priority class [15], servicing non-real-time tasks at NORMAL priority and the other two corresponding to $P^{high}$ and $P^{low}$ respectively. This thread selects the promoted DPA from the *Promoted Push Queue* and transmits the associated messages divided into blocks of uniform size $S^o$ . The entries in the *Promoted Push Queue* and the *Push Queue* are sorted based on their respective priorities, which allows us to deal with the problem of limited priority levels within the operating system. The separation of functionality between the *Dispatcher* and the *Pusher* prevents or minimizes priority inversion at the socket level by ensuring that high priority data push requests suffer blocking time due to priority inversion for only the duration of pushing one data block of size $S^o$.

## 7.2 Profiler

To service real-time tasks, we need to know the computation time, communication overhead and propagation delay for the scheduler to be effective. Our solution is to use profiling at system configuration time on the specific platform that our scheduler will be used. Our profiler toolkit consists of two application programs, which make use of MidART's services to set up a memory-to-memory data transfer between two different nodes in a networked environment. These programs are run with a minimally instrumented version of MidART's code that takes time stamps at different points on the data transfer path between the two nodes.

One of the programs of the toolkit profiles the end-to-end timing parameters of different entities involved in the networked application. The entities that can be profiled in addition to the application programs are (1) the M_Write and M_Read MidART library calls,

(2) the DPA and DRAs, and (3) the Server. Data collected by this program provides the following information:

- Application-to-application Round Trip Time.
- Memory-to-memory Round Trip Time.
- Computation time and overhead of ADS's scheduler.
- Jitter introduced at each stage of the data path.
- Time to complete library calls M_Write and M_Read.
- Time to push the data associated with a ReMA.
- Optimal block size for pushing messages.

The other program included in the toolkit calculates the receiver throughput at the reader node. This is used to calculation buffer allocation. The timing and buffer information collected by the toolkit can be utilized effectively by the run-time scheduler and admission control algorithms.

## 7.3 Experimental Results

We have done extensive experimentation in studying the feasibility of our user level scheduling schemes. This section presents some of these results. All the experiments were carried out on a Pentium II PCs with Windows NT 4.0 operating system over 100BaseT Fast Ethernet. We have isolated the network segment from the rest of the LAN for running all the experiments. The two PCs we used in the experiments have processor speeds of 333MHz and 266MHz respectively. All the experiments reported in this section have been run on both machines to verify that the observations are not machine dependant.

### 7.3.1 Alleviating Priority Inversion

Figures 4 to 7, we show how our server can alleviate the possibility of priority inversion at the socket level. T1 is a high priority task sending messages of size 1K bytes with a period of 20ms, while T2 is a low priority task sending messages of size 64K with a period of 10ms. These two tasks model a scenario where a command-and-control task with high priority and small message size coexists with another periodic task with large data transfer. Figures 4, 5, 6, and 7 show the results for each of the tasks when they were both executing in the system. The figure presents the time between data pushes for 1000 executions. When they execute on Windows NT without our user level server's scheduling support, Figures 4 and 6 show how the data sending periods deviate widely from the expected periods. At many points, we can observe that the high priority task's data sending periodicy requirement is violated. Once we place the two tasks under the control of our server's scheduler, they show a remarkable difference in their execution pattern, i.e., in Figures 5 and 7, one can see how their rates are stabilized. Note that the 1ms deviation from the 10ms or 20ms periods in these figures are due to the 1ms timer accuracy provided by Windows NT.

### 7.3.2 Message Block Size $S^o$

As described in Section 6.4, if we do not bound the message sizes, a high priority task can potentially be blocked for a very long time. Thus we have carried out experiments to determine the value for $S^o$. $S^o$ should not be too large in order to minimize the blocking time, and it need not be smaller than 1ms since that is the accuracy of the timer provided by Windows NT.

Figure 8 shows the time taken to send messages as a function of the message sizes from 1K to 60K. In this experiment, we have recorded the overhead of (1) sending each message in its entirety, (2) dividing each message into 1K size data blocks to send, and (3) dividing each message into 8K size data blocks to send (if the message is greater than 8K). Sending a message in its entirety is basically the overhead cost of the socket calls plus UDP/IP packetization, while dividing up a message will incur extra MidART overhead. As one can see, a message of size 60K can take up to 6ms to send in its entirety. This will be too long a blocking time for many real-time tasks. The figure also shows that dividing messages up into 1K impose too much overhead for large messages. Uniform message size of 8K incurs only very small amount of overhead compared with not dividing the message up at all. We have also done experiment on the receiver side to verify that the 8K is a good choice for the value of $S^o$. The receiver throughput is shown in Figure 9, where it can be seen that 8K data size basically performs as well as the messages in their entirety, while 1K data block sizes reduces the throughput to only one third of what 8K size can achieve.

### 7.3.3 User Level Scheduling Overhead

Figure 10 shows the major overhead of our scheduler as a function of the number of timers being used. Since we support periodic tasks using the dual priority scheduling algorithm, periodic timers are used in our implementation for the promotion time notification. As shown in the figure, 16 timers impose an overhead of 0.27ms. The maximum standard deviation was 0.061 for all the timer overhead data points. From these tests, it appears that our user level scheduling scheme can easily be used for tasks with a few milliseconds timing requirements.

### 7.3.4 Receiver Periodicity

Since the DRAs are event driven, we need to verify that our user level scheduling of DPA and message rate control can actually help the DRAs to maintain the application requested data reception periodicity.

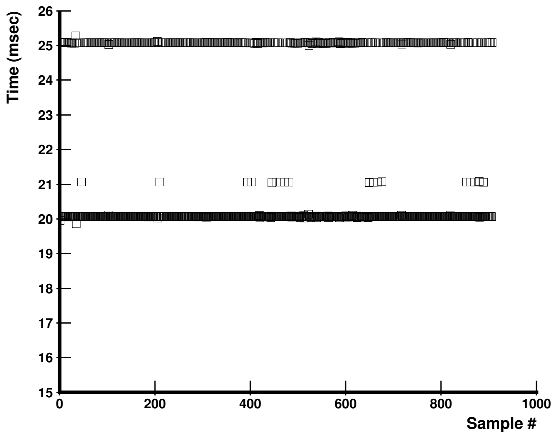


**Figure 4. T1 without server scheduling.**

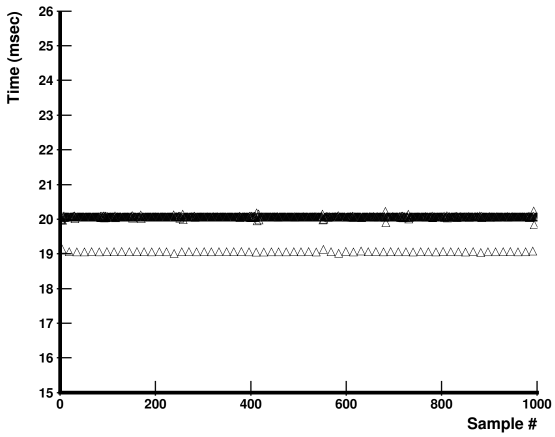


**Figure 5. T1 with server scheduling .**

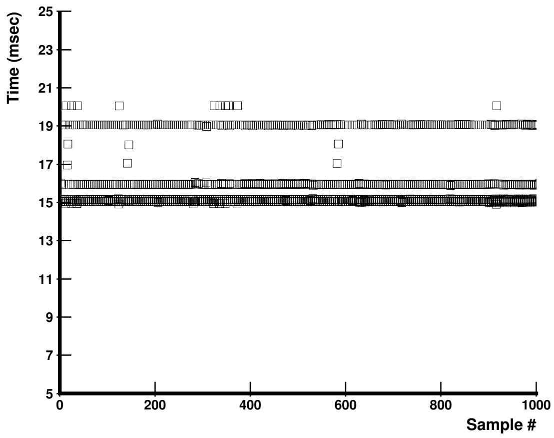


**Figure 6. T2 without server scheduling.**

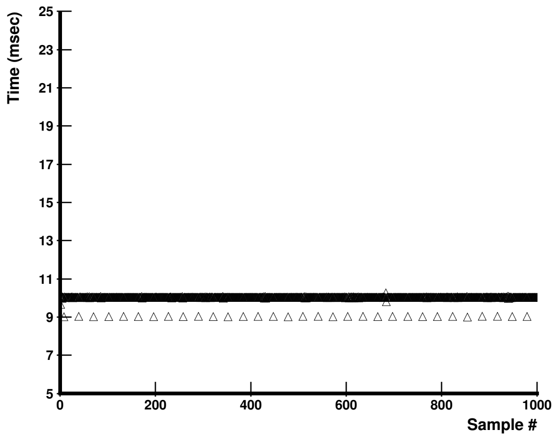


**Figure 7. T2 with server scheduling.**

In this experiment, 14 end-to-end real-time tasks execute in AB mode (Asynchronous data push and Blocking read). The 14 writers simultaneously communicate with 14 real-time readers on a remote machine for 2.5 minutes. Each writer sends a message of 8KB periodically. The period for the end-to-end task is calculated according to the following formula where $BaseRate = 30ms$ and $Step = 20ms$:

- $T_i = BaseRate + Step * i$

Figure 11 plots the average periodicity for the two highest rate readers, i.e., $T_1$ and $T_2$. The standard deviation for $T_1$ is 1.7 and for $T_2$ is 1.5. The periodicities for all the other readers are similar or better than these two tasks since their periods are less demanding on the system. We can see that the readers are able to obtain their respective required periodicity very well.

#### 7.3.5 Non-Real-Time Task Performance

Our last experiment integrates real-time and non-real-task execution. We need to ensure that non-real-time tasks do receive good service using our user level dual priority server, while real-time tasks maintain their required timing requirements. To evaluate this, we have added a non-real-time application, which transmits a fixed amount of data from one node to another using MidART's services, to the set of 12 real-time writers and readers as specified in the last section.

The non-real time application sends a total of 60MB of data from one node to the other in blocks of 8KB every 10ms. Figure 12 shows the time that elapses at the receiver from the arrival of the first 8KB block to the last one as the number of reader applications increases from 1 to 12. If there are no real-time tasks at all, the response time for the non-real-time is 75 seconds. Figure 12 shows that when there is one real-

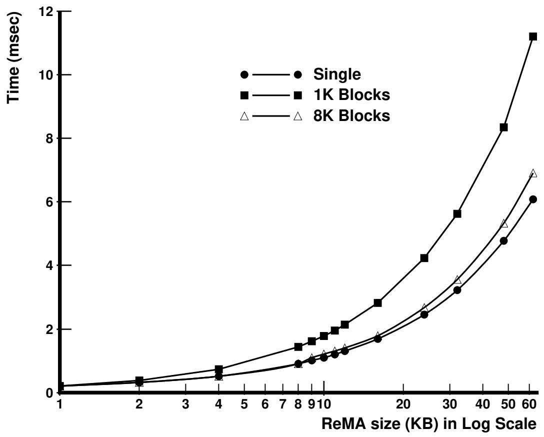


**Figure 8. Server Overhead for Sending.**

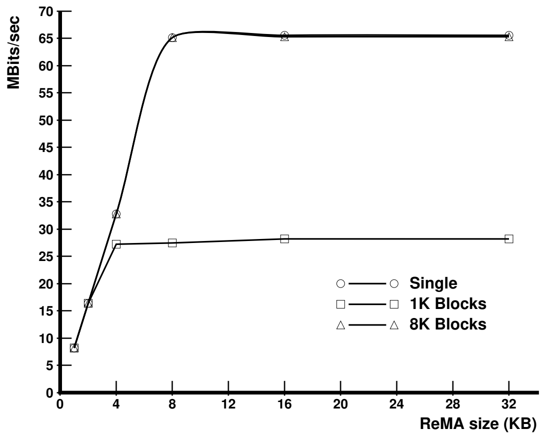


**Figure 9. Receiver Throughput.**

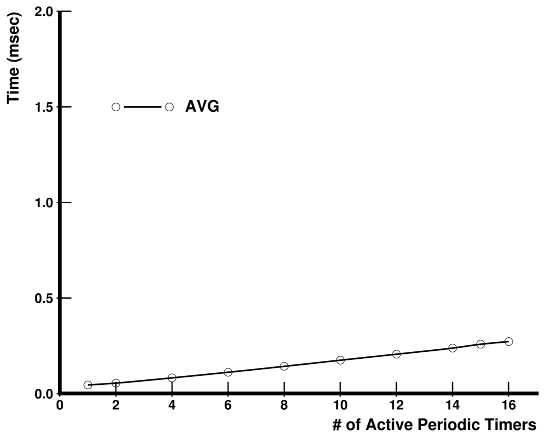


**Figure 10. User Level Server Timer Overhead.**

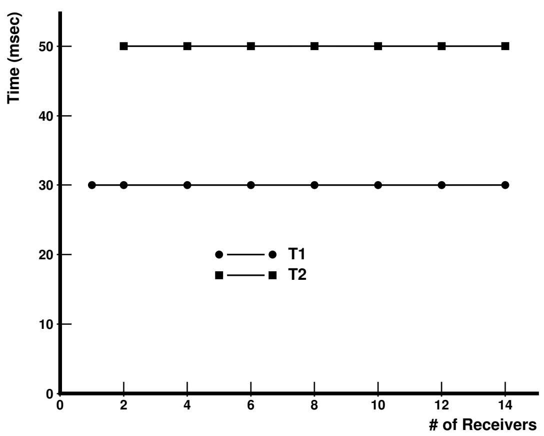


**Figure 11. Receiver Periodicity.**

time reader in the system writing 8KB of data every 50ms, the response time for non-real-time application is 75.25 seconds. As the number of real-time readers increases, the response time for the non-real-time application does increase, as expected, from 75.25 to 75.86 seconds. A further delay of the response time is observed when the base period of the real-time tasks decreases from 50 to 30 ms. In this same experiment, we observed that all the real-time tasks maintained their periodicity very well.

## 8 Conclusion and Future Work

The work described in this paper is motivated by the emerging need to ensure end-to-end predictability of real-time communicating tasks in COTS-based distributed systems. The solutions developed to overcome the problems faced in utilizing COTS platforms have been implemented and evaluated using a PC-based Windows NT system with Ethernet as the LAN. Our implementation has been carried out in the context of MidART, a real-time network middleware. Experimental results show how our server-based solution alleviates the problems such as those that occur due to limited operating system priority levels, priority inversion, and the presence of non-real-time applications.

Since regular Ethernet and UDP/IP are used underneath the MidART implementation reported in this paper, exact determinism in end-to-end task communication is not possible. Under normal operating conditions where Ethernet traffic load is less than 60%, we can expect the system to be fairly well behaved. To achieve determinism end-to-end, we need to incorporate some of the more recent real-time solution for the network layer (e.g., [18, 19]) and the protocol stack. One of our future goals is porting our user level scheduler onto Linux where we can achieve kernel level

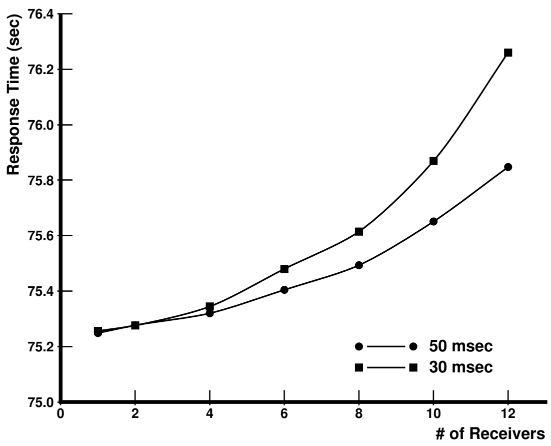


**Figure 12. Response Time of Non-Real-Time Application.**

scheduling.

Currently we are also working on schedulability analysis techniques to develop the necessary admission control policies for readers as well as writers to work with the scheduling techniques reported here. We are also developing an implementation that fully realizes the potential for improved resource utilization afforded by the dual priority approach.

## References


[1] M. Barabanov and Victor Yodaiken. Real-Time Linux. In *Linux Journal*, March 1996.

[2] W. Bolosky, R. P. Fitzgerald, and J. Douceur. Distributed Schedule Management in the Tiger Video Fileserver. In *the 16th ACM Symposium on Operating Systems Principles*, October 1997.

[3] Dick Caro and Rich Mullen. Ethernet as a Control Network. *CONTROL Magazine, Putman Publishing Co*, February 1998.

[4] Deborah Claymon. Control Freaks: Control networks will regulate every factory, house, and office . *The Red Herring*, March 1998.

[5] H. Custer. *Inside Windows NT*. Microsoft Press, 1993.

[6] R. Davis and A. Wellings. Dual Priority Scheduling. In *IEEE Real-Time Systems Symposium*, December 1995.

[7] O. Gonzalez, C. Shen, I. Mizunuma, and M. Takegaki. Implementation and Performance of MidART. In *IEEE Workshop on Middleware for Distributed Real-Time Systems and Services*, December 1997.

[8] P. Goyal, X. Guo, and H. M. Vin. A Hierarchical CPU Scheduler for Multimedia Operating Systems. In *USENIX Symposium on Operating Systems Design and Implementation*, October 1996.

[9] VenturCom Inc. *Real-Time Extension 4.1 for Windows NT, http://www.venturcom.com.* 1997.

[10] K. Jeffay, F. D. Smith, A. Moorthy, and J. Anderson. Proportional Share Scheduling of Operating System Services for Real-Time Applications. In *19th IEEE Real-Time Systems Symposium*, December 1998.

[11] M. Jones and J. Regehr. Results from a Latency Study of Windows NT Or ... The Problems You're Having May Not Be the Problems You Think You're Having. In *Work in Progress, 19th IEEE Real-Time Systems Symposium*, Madrid, Spain, Decmber 2-4 1998.

[12] J. P. Lehoczky. Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines. In *11th IEEE Real-Time Systems Symposium*, December 1990.

[13] I. Mizunuma, C. Shen, and M. Takegaki. Middleware for Distributed Industrial Real-Time Systems on ATM Networks. In *17th IEEE Real-Time Systems Symposium*, December 1996.

[14] J. Nieh and M. Lam. The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications. In *16th ACM Syposium on Operating Systems Principles*, October 1997.

[15] K. Ramamritham, C. Shen, O. Gonzalez, S. Sen, and S.B. Shirgurkar. Using Windows NT for Real-Time Applications: Experimental Observations and Recommendations. In *IEEE Real-Time Technology and Applications Symposium*, June 1998.

[16] C. Shen and I. Mizunuma. RT-CRM: Real-Time Channel-Based Reflective Memory. In *IEEE Real-Time Technology and Applications Symposium*, June 1997.

[17] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. Haruah, J. Gehrke, and C. Plaxton. A Proportional Share Resource Allocation Algorithm for Real-Time, Time-Shared Systems. In *17th IEEE Real-Time Systems Symposium*, December 1996.

[18] S. Varadarajan and T. C. Chiueh. EtheReal: A Host-Transparent Real-Time Fast Ethernet Switch. In *International Conference on Network Protocols (ICNP)*, October 1998.

[19] C. Venkatramani and T. C. Chiueh. Design, Implementation, and Evaluatino of a Software-based Real-TimeEthernet Protocol. In *ACM SIGCOMM*, 1995.