Implementation and Performance of MidART

Oscar González
Computer Science Department
University of Massachusetts
Amherst, MA 01003

Chia Shen
MERL - A Mitsubishi Electric Research Lab.
201 Broadway
Cambridge, MA 02139

Ichiro Mizunuma
Morikazu Takegaki
Industrial Electronics and Systems Lab.
Mitsubishi Electric Corp.
8-1-1, Tsukaguchi-honmachi, Amagaski, Hyogo, 661
e-mail: mizunuma, takegaki@con.sdl.melco.co.jp

Abstract

In this paper, we describe our experience in the implementation of MidART – Mid*dleware and network* A*rchitecture for distributed* R*eal-*T*ime systems. Our MidART project addresses the problem of middleware design to support high speed network based distributed real-time applications. The uniqueness of MidART lies in the simplicity of services provided and the flexibility of data reflection models, compared with more general purpose but much more complicated middleware such as CORBA implementations. This simplicity leads to ease of understanding and ease of use by application builders, while its flexiblity sufficiently serves the needs of the class of real-time applications MidART is designed for.*

1 Introduction

It is becoming ever more important for both industry and academia to design distributed real-time systems using open, standard, commercially available computers and networks. This is largely due to (1) network and processor technology advances, (2) cost considerations, and (3) the desire for easy system integration and evolution. Currently, there is no network middleware for open standard networks and operating systems for real-time applications. Existing systems are largely proprietary. On the other hand, socket interface is cumbersome and difficult to use for application builders. Moreover, real-time applications need end-to-end quality of service provision. To facilitate the construction of distributed real-time applications on open off-the-shelf systems, we must first provide easy-to-use real-time programming models and services to the real-time application designers.

In this paper, we describe our experience in the implementation of *MidART* – *Mid*dleware and network *A*rchitecture for distributed *R*eal-*T*ime systems [5]. Our MidART project addresses the problem of middleware design to support high speed network based distributed real-time applications. The class of applications we are dealing with are those in which humans need to interact (e.g., control and monitoring) with instruments and devices in a networked environment through computer-based interfaces. Examples of such applications include distributed industrial plant control systems, multi-machine surgical simulation systems, virtual labs, and large telescope control systems. The end-to-end delay requirements for these applications range from one or two milliseconds to hundreds of milliseconds.

Our initial proof of concept prototype was constructed on PC running QNX real-time operating system over ATM networks as reported in [5]. Currently the project has evolved to implement the entire middleware in C++ on Windows NT as well as Unix platforms with Fast Ethernet. In this paper, we report our experience and insights gained, as well as performance findings of this implementation.

1.1 Overview of MidART

The MidART middleware provides a set of real-time application specific but network transparent programming abstractions that support individual application data monitoring and control requirements. The focus of our middleware is to support the end-to-end application real-time data transfer requirements with a set of easy-to-use communication service programming interfaces. The two key services provided by MidART are Real-Time Channel-Based Reflective

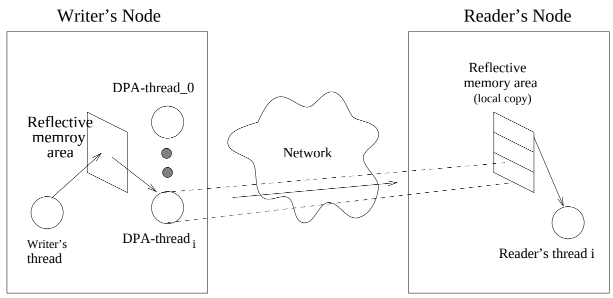


Figure 1: RT-CRM High Level Architecture

Memory (RT-CRM) [7] and Selective Channels [5]. Selective Channels allow applications to dynamically choose the remote node(s) which data is to be viewed from and sent to at run time. This is accomplished via a set of channel start and stop protocols, and channel bandwidth resource overbooking schemes.

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 update transfer. A writer has a memory area where it stores its current data, while a reader establishes a corresponding memory area on its own local node to receive the data reflected from the writer. Data reflection is accomplished by a data push agent thread, a DPA-thread, residing on the writer's node and sharing the writer's memory area. This agent thread represents the reader's QoS and data reflection requirements. A virtual channel is established between the agent thread and the reader's memory area, through which the writer's data is actively transmitted and written into the reader's local memory area. 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 thread. The selection of the particular writer at any time is done via Selective Channel protocols.
- A writer memory area may be connected to many remote reader memory areas simultaneously. There can be many data push agent threads representing many readers associated with the same writer memory area.
- RT-CRM supports both synchronous and asynchronous data reflection models.

For more detail of Selective Channels and RT-CRM, readers are refered to [5] and [7].

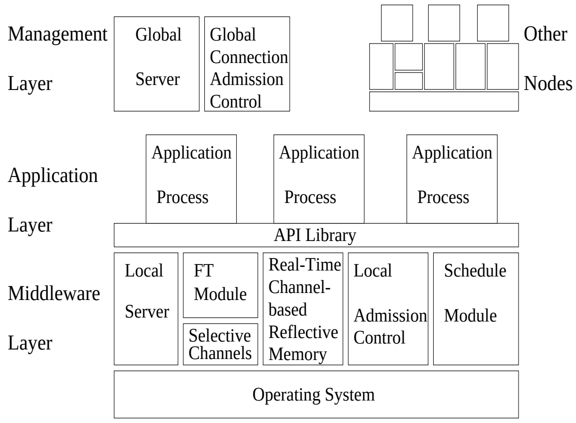


Figure 2: System Architecture

The uniqueness of MidART lies in the simplicity of services provided and the flexibility of data reflection models, compared with more general purpose but much more complicated middleware such as CORBA implementations [3] [6]. This simplicity leads to ease of understanding and ease of use by application builders, while its flexiblity sufficiently serves the needs of the class of real-time applications MidART is designed for.

2 MidART Architecture

MidART in essense is a simple distributed computing environment designed to support the development of real-time applications on open off-the-shelf systems. To be effective and capable of supporting a wide variety of applications with different requirements, MidART is organized in layers and it is highly modularized and object oriented. The layered architecture reduces the complexity of the design and allows a clean separation of the application from the middleware and its services. Moreover, a modular design increases the flexibility of the middleware by allowing the addition, removal and modification of services without changes to other modules. In addition, it facilitates the implementation of different policies and mechanisms for supporting a particular service.

Figure 2 shows the MidART architecture from the perspective of one of the nodes in distributed environment. From this point of view, the modules and services that make up MidART reside in one of following three layers: 1) management, 2) application and 3) middleware. The following subsections describe each layer and its components.

2.1 Management Layer

The *management layer* oversees the non-real-time interactions between application programs executing in different nodes of the distributed environment. The only module residing in this layer is called the *Global*

Manager (GM). This module has two components: the *Global Server* (GS) and the *Global Connection Admission Control* (GCAC). A description of each of the components of the Global Manager is provided below:

- The Global Server provides MidART's applications with a set of administrative services that facilitates the registration, location, creation, destruction and attachment of *Re*flective *M*emory *A*reas (ReMA). Locating these services in a reliable server allows upgrades and modifications to application programs to be done in a plug and play fashion. The Global Server is the process responsible for maintaining the *global definition table*. The table is a network wide database that allows *Local Server* to know the characteristics of all the reflective memory areas currently active [1]. The characteristics of a ReMA include a unique name/identifier, the size of the memory area, its QoS parameters, its writer location, and a list of the reader threads attached to it. This information is needed by the Global Admission Control and the Local Servers in order to set up additional areas, remove existing ones and attach new readers.

  The database also contains all of the registered services. Before an application can create a reflective memory area, the service provided by a reflective memory area must be registered. Upon creation of the memory area, the Global Server will assign a unique identifier to that particular memory area.

- The Global Connection Admission Control is called by the Global Server when a reader application wants to attach to a ReMA. This module works in unison with the *Local Connection Admission Control* (LCAC) modules of both the writer's and reader's nodes to determine whether the system can support the association between the reader and the ReMA. The GCAC is responsible for performing a schedulability analysis of the data reflection QoS requirements on the network before admitting an association. A description of the LCAC module and its functionality can be found in section 2.3.

  Although the Global Manager is conceptually one entity, in actual implementation, we have two choices. One is to construct the GM as a centralized unit residing on one server node at a time, possibly having a replicated GM on a second server node for reliability. A second choice is to implement a distributed GM where each node in a network participate in the GCAC process and keeps a copy of the global definition table. For this second implmentation choice, a distributed ReMA registration, creation, destruction and attachment protocol is necessary. We have implemented the first choice in the current version of MidART and are considering to move to a distributed GM implementation in the next step.

2.2 Application Layer

The only component of MidART located at application layer is the API (Application Programming Interface) library. An application process uses the library calls to invoke MidART's services.

The MidART library is the backbone of our design, since an efficient, flexible, intuitive and straightforward API is capable of supporting different applications. In addition such an interface, not only facilitates the construction of distributed application programs, but in this case, it also relieves application programmers from cumbersome and error-prone details of handling operating system and network interfaces. The list of the MidART API calls can be found in the Appendix. Section 4 contains a simplified example on how to use them to build an industrial application.

2.3 Middleware Layer

The rest of MidART modules are located at this layer. All of these modules resides within the *Local Server Process* (LSP), which together support the Real-Time Channel-based Reflective Memory and the Selective Channels services. A description of each of the modules in the Local Server Process is given below:

- The *L*ocal *S*ever (LS) module is responsible for handling request and communicating with the application programs, with the Global Manager and Local Servers located in other nodes of the network. The Local Server is also responsible for maintaining the *local memory area table*. This table holds the description and access information to the local reflective memory objects.

  An efficient mechanism for accessing the *local memory area table* is needed, since all modules within the Local Server Process, as well as the MidART library calls require some of the information that resides in the table. Section 3.1 describes the structure and implementation details of this module.

[1] Active means that the ReMA 1) has been created and 2) has not been removed. An active ReMA may have no readers/writers at certain times.

- The *R*eal-*T*ime *C*hannel based *R*eflective *M*emory module is responsible for the creation and management of the local *R*eflective *M*emory *A*rea(ReMA) objects. As described previously, a ReMA is an association between a writer's memory and a reader's memory on two different nodes in a network. The data stored in the writer's side is transferred/reflected to the reader's area through a virtual channel. Thus, this module is responsible for the creation of the virtual channel and the pair of objects involved in the transferring of data through the channel.

  The objects responsible for the reflection of data are the *D*ata *P*ush *A*gent (DPA) and Receiver objects. A DPA object resides on the writer's node and shares the writer's memory area. The DPA object represents the reader's QoS and data reflection requirements. Similarly, the Receiver object handles incoming datagrams and it is responsible for updating and maintaining the contents of the ReMA and its buffers at the reader's node.

  The RT-CRM module is capable of supporting several configurations or combination of configurations for the association between a DPA and a Receiver. Figure 3 shows three possible configurations which are described below:

  - **DPA-to-Receiver:** In this configuration, see figure 3 (a), each ReMA at the writer's node has a DPA thread which transmit the data over the virtual channel. Similiary, the corresponding reflective memory area at the reader's node has a Receiver thread working in its behalf.
  - **Single DPA - Multiple Receivers:** Here, a group of ReMA at the writer's node are associated with a single DPA thread. As shown in figure 3 (b), the DPA is responsible for sending data to multiple receivers threads working on behalf of each ReMA at the reader's node.
  - **Multiple DPA - Single Receiver:** Figure 3 (c) illustrates this configuration, where a single receiver at the reader's node is responsible for handling incoming data from different DPA threads.

  Each configuration has its unique implications on (1) the number of context switching and (2) the amount of demultiplexing overhead of messages the middleware will impose. The current implementation only supports the DPA-to-Receiver configuration, but we are currently investigating the benefits of using other configurations.

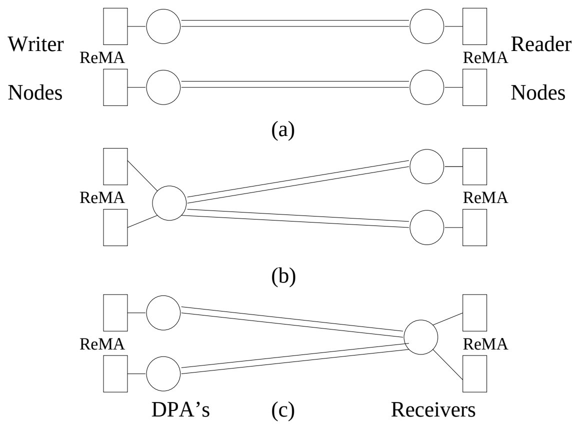


Figure 3: Configurations for an DPA-Receiver Association

- The *S*elective *C*hannels (SC) module sets up the necessary connections and mechanisms needed to support the channel-switching algorithm. In particular, a multicast control channel must exist to run the Selective Channel prototcol as described in [5].

- The **Scheduler** module is responsible for the scheduling of the active DPA and receiver objects. Currently, we are in the process of evaluating the benefits of using rate-based and/or fixed priority rate monotonic algorithms.

- The **Fault Tolerance** module provides two optional services. First, it runs a Heart Beat hand shaking protocol to determine which nodes in the network are active. The other service is a thin layer of messaging protocol/transport protocol on top RT-CRM that improves the reliability of UDP, but is not as expensive as TCP. It maintains sequence numbers in each datagram, buffers messages at the sender, and notifies the application of missing messages.

- The *L*ocal *C*onnection *A*dmission *C*ontrol module performs local schedulability analysis and collaborates with the Global Connection Admission Control module to determine whether to admit an association to a ReMA. The LCAC performs two types of analysis. On a writer's node, the LCAC checks the schedulability of the DPA thread on its CPU based on the QoS of the reader. Similarly, on a reader's node, the LCAC determines whether the data reflection QoS requested by the reader can be scheduled on its CPU. Of

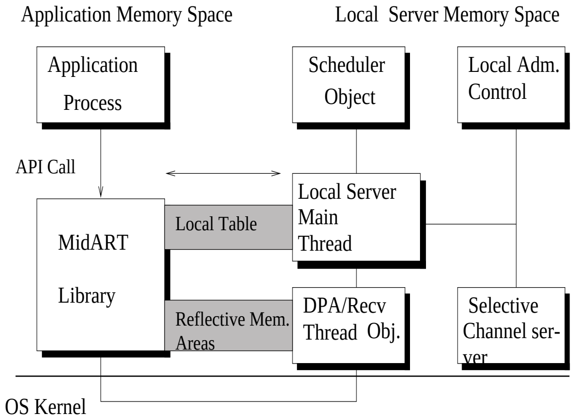


Figure 4: Architecture of Local Server

course, a node can be simultaneously a writer and a reader.

The LCAC module also interacts with the *Scheduler* when an association between a writer's ReMA and a reader's area is added or removed. The type of interaction depends on the policy being used by the Scheduler. For example in a priority scheme, a new association may demand an adjustment on the priorities of the DPA or receiver threads.

3 Implementation Details

Under current implementation of MidART, an application can use all of the API function calls and services provided by the Real-Time Channel based Reflective Memory and Selective Channels abstractions. To make use of these services an application process only needs to compile its program with the MidART library and its corresponding header file. In addition, the Local Server Process needs to be run on each node in the network and the Global Server Process only runs in one node. We are currently working on the implementation and evaluation of the algorithms for admission control, both local and global, and scheduling of threads.

The following subsection describes the implementation details of the Local Server Process, the Global Server, some key objects and the communication mechanism used among the different components of the MidART library.

3.1 Local Server

The Local Server is implemented as a multi-threaded process. Figure 4 illustrates a high level architecture of the Local Server.

The main thread of control creates a shared memory area that serves as a container for the *local memory area table*. The reason for selecting a shared memory area, as opposed to an object, is twofold. First, the container needs to be locked into memory, in order to avoid page faults while the *write* and *read* library calls execute or when DPA/receiver threads access a reflective memory area. The second factor relates to the required visibility of the local memory table area. For an efficient implementation, the information contained in the table needs to be visible to all Local Server's threads and application processes in a node. Since NT provides the simple mechanism for locating, attaching to and locking a shared memory area, we selected it as the container of the table. It is important to note that the implementation provides the necessary mechanisms to guarantee the consistency and integrity of the local memory table via locking protocols.

Other responsibilities of the main thread include the creation of the: 1) Scheduler Object and Local Admission Control Server, 2) thread of control for Selective Channel Server, and 3) threads responsible for handling the communication with the Global Server, the local application programs (via the API calls) and LS located in other nodes. A brief description of each of these threads follows:

The Selective Channel Server is implemented as sequential UDP server. It supports the channel switching protocol describe in [5]. This protocol enables an application to start or stop the DPA thread associated with a particular ReMA.

The thread handling the communication with the Global Server implements a TCP client. Once the connection is established, it remains open as long as the LS is active.

The thread communicating with other Local Servers runs as a sequential server. This server dynamically creates threads that reside within the address space of the local server as applications set up reflective memory areas. These threads are responsible for running the DPA and receiver objects.

3.1.1 Reflective Memory Area

The reflective memory areas are implemented as objects. The buffers that make up the ReMA are created as shared memory areas. This allows a writer's application process direct access to the same area used by the DPA object. Similarly, a reader application shares the buffers with the receiver object.

In order to locate these objects rapidly during a read or a write library call, all application processes in a particular node have access to the *local memory area table*. For each ReMA object, the table contains the necessary information to access the ReMA buffers directly. In addition, all of the buffers belonging to a ReMA are locked in memory.

3.2 Global Server

The Global Server is also a multi-threaded process. The server is implemented as a concurrent TCP server. All of its client's requests require either the creation/update of an entry in the global definition table, information about a particular entry or the destruction of an entry.

3.3 Communication Mechanism

In order to understand the interactions between the key components of MidART, we can trace the invocation of an API call using Figure 4 as an aid. The figure illustrates the communication mechanism used between an application process and the local server process.

When an application makes an API call to create a Reflective Memory Area, the MidART library sends the request to the main thread of the local server. The communication between the library and the local server is achieved by passing a message. Our implementation uses the Windows NT mailslot API to pass messages between the Local Server and the MidART library. Once the local server validates the request and obtains a memory identifier from the global server, a reflective memory area is created with its associated DPA thread. An entry in the *local memory area table*, indexed by the identifier, is created.

The local server returns a message, via a mailslot, containing the identifier and the result of the operation to the calling thread. Since an application process can be multi-threaded, each thread is required to startup the MidART library before using any of the API. This is necessary because the middleware needs to create the return mailbox for each thread.

Not all of the MidART API calls communicate with the Local Server using the mailslot API. For the *write* and *read* API calls the communication between the MidART API and the objects belonging to the Local Server is done using shared memory. For these calls, our implementation allows the library, with a valid identifier, direct access to the shared memory area where the local table is located. The table contains a pointer to the current write or read buffer of the ReMA. Thus, an efficient mechanism for writing and reading is supported.

3.4 Operating system mechanism used

We are currently working on the implementation of the Scheduler module. Our implementation uses the operating system supported thread priority and timer facilities to achieve desired preemptive real-time scheduling.

Our current design uses some abstraction and services particular to NT. However, we have been careful to make sure that compatible abstractions are also available in a POSIX compliant operating system. For example, the mailslot API can be easily implemented using message queues. Another example is the *Event* IPC mechanism, the behavior of a Windows NT event can be implemented using the *pthread_condition_wait* function found in the Pthreads library. By doing so, we have traded some compatibility in favor of better performance.

4 An Example of Using MidART API

In this section, we demonstrate, via a simple example, how to use the MidART API calls to build a flexible and efficient monitoring and control application. A brief description of each of the MidART API calls is given in Appendix A.

The example is intended to replicate the normal operating conditions that occur at the monitoring computer of an operator in an industrial plant. The operator receives data from devices and Programmable Logic Controllers (PLC's) distributed throughout the plant. The operator can send control messages to the various devices and controllers. Also, the operator station is capable of displaying streaming video from cameras.

In order to keep the example brief, we concentrate on how to implement a service that allows an operator to selectively received data from a particular PLC. This example has the following participants:

- *PLC_1* Data: This PLC acquires new sensor data (i.e. temperature readings inside an electrical transformer or chamber) at regular intervals (*plc1_rate*).
- *PLC_2* Data: This PLC also acquires new sensor data from a different equipment and location at regular intervals (*plc2_rate*).
- Operator: The operator first is only interested in monitoring data from PLC_1 at regular periods. After sometime an external event (Z) is detected and from this moment on, the operator stops monitoring the data from PLC_1 and starts monitoring data from PLC_2.

The Pseudo-code for the above application programs using the MidART API is shown in Figure 5.

5 Performance

The objective of our initial performance tests is to characterize and validate our current implementation

```
Process PLC 1 ()
  CRM_RegisterService(svc1)
  CRM_CREATE(svc1)
  Forever
    CRM_WRITE(svc1, DATA1)
    WaitFor(plc1_rate)
END Process PLC 1

Process PLC 2 ()
  CRM_RegisterService(svc2)
  CRM_Create(svc2)
  Forever
    CRM_Write(svc2, DATA2)
    WaitFor(plc2_rate)
END Process PLC 2
```

```
Process OPERATOR ()
  CRM_Attach(svc1)
  CRM_Attach(svc2)
  CRM_Start(svc1)
  Until EVENT(Z)
    CRM_Read(svc1, BUF)
    CRM_Stop(svc1)
    CRM_Start(svc2)
    CRM_Read(svc2, BUF)
END Process OPERATOR
```

Figure 5: Pseudo-code of Monitor Plant Example

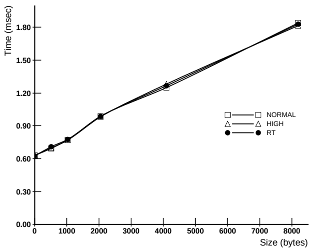


Figure 6: Avg RTT (1 writer).

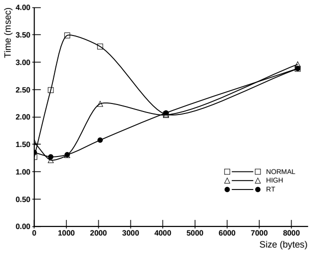


Figure 7: Max RTT (1 writer).

MidART. In particular, we are interested in the following three aspects about our implementation: 1) measure the application-to-application latency introduced between the writer and the reader of a ReMA, 2) determine the overhead that RT-CRM incurs compared with raw UDP/IP, and 3) measure the overhead costs of MidART API calls involved in the registration/deregistration of services and the creation/destroy of reflective memory areas.

The performance experiments discussed below were conducted using two single CPU Pentium Pro 160MHz PCs with 32MB of RAM running Windows NT 4.0. We used an Eagle FastEthernet switch from Microdyne to connect the two machines.

5.1 Latency

To determine the latency introduced by RT-CRM, we measured the application-to-application round-trip time (RTT) of a write event. RTT measurements are needed, since the clocks in the two machines are not synchronized. For all of the experiments described in this section, each data point is the result of 1000 runs on an unloaded system and network.

In our first experiment, we create a writer thread at one node and a reader thread at the other node. The reflective memory area is attached under a synchronous data push operation mode and blocking read mode. The period for the writer is 50 milliseconds. After the reader attaches to the writer's memory area, a time stamp is recorded at the writer's host every time this thread invokes the *write* API call. The moment the data is received by the reader application thread, an acknowledgment is sent back to the writer's host. Upon the arrival of the acknowledgment a second time stamp is recorded. The difference of the two time stamps provides the value of the RTT.

Figures 6 and 7 show the average and maximum RTT respectively for different message sizes and priority classes assigned to threads involved (DPA, receiver, writer and reader). The priority classes are: Real-Time, High and Normal. For each performance

msg size (byte)	UDP(avg)	RT-CRM (avg)
1	0.459240	0.626410
512	0.526845	0.709782
1024	0.582634	0.776303
2048	0.763597	0.988170
4096	0.978962	1.266156
8192	1.460686	1.828897

Table 1: Round Trip Latency. (Time in msec)

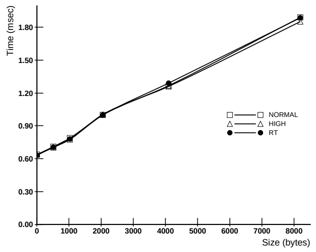
curve, all the threads involved in the measurements are assigned to the same priority class. One important result of this experiment is that under all the class priorities the average RTT is acceptable for a wide range of applications. The worst case (i.e., maximum) only occured very rarely. We are currently investigating the cause of such worst case occurrences. Moreover, that the worst-case RTT using Real-Time priorities remains linearly proportional with the message size, which is not the case for the other two priority classes.

The overhead that RT-CRM incurs compared with raw UDP/IP for this experiment is shown in Table 2. The major source of overhead cost comes from the additional memory-to-memory copies done in RT-CRM.

We have done additional experiments measuring the RTT under a more loaded system. Instead of only one application writer, in this second experiment we used five writers and five readers. The periods of the writers were 1.0, 0.5, 0.2, 0.1 and 0.05 seconds. Figures 8 and 9 show the average and maximum RTT for this setup.

5.2 Overhead costs of API calls

Overhead costs of the API calls are important in order to obtain a characterization of the MidART implementation. At the same time, they provide us with feedback, which can be used to fine tune the implementation or to review some of the design decisions made. The cost of each of the calls evaluated is shown in Table 3. Each data point is the result of 100 runs



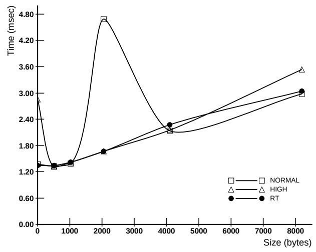


Figure 8: Avg RTT (5 writers). Figure 9: Max RTT (5 writers).

API Call	Average	Maximum
Register	2.695833	9.400913
Unregister	2.960822	3.088380
Create	7.144702	7.854627
Destroy	3.134065	3.27359

Table 2: Overhead cost of API Calls in milliseconds.

in an unloaded system and network.

Register and Unregister calls must contact the global name server to obtain the IP address where the area is going to be located from a name string. A Create call does the following:

1) Contacts global server to obtain id.

2) Creates memory area (allocates buffers and locks them into memory)

3) Creates mutex for read/write access.

4) Creates an event to synchronize with DPA thread.

6 Related Work

Recently [4] has reported the design and implementation of a real-time CORBA event service. Although the high level semantics of their real-time event service is actually very similar to the Real-Time Channel-Based Reflective Memory in MidART, their design has potential bottleneck problems due to the usage of a centralized "Event Channel" where all multiplexing and filtering of communication take place. In addition, all their performance (both for utilization and latency) was done on a single machine, i.e., no network involved at all. The utilization was done on a uni-processor Pentium and the latency was done on a dual-processor shared memory Sun UltraSPARC 2 with two 167 Mhz CPUs. Moreover, their performance was done by putting all consumers, suppliers and the Event Channel in the same process without the real ORB overhead tested at all. Thus it is difficult to see how well their event service can really support distributed real-time applications over the network. In contrast, all of our MidART performance tests have been done in a truely distributed environment with off-the-shelf commercial network components. As it is well known that most of the end-to-end latency and delay in a real networked system comes from network interface card, network interface drivers, memory-to-memory copying and process context switching. If performance tests were only done in a single node and all major tested software were contained in the same process, then one is simply testing the mere speed of the processor and OS system calls.

Other work and technology that are also related to MidART include distributed shared memory (DSM) [1], reflective memory[10], and memory channels [2]. The Real-Time Channel-based Reflective Memory in MidART is much more flexible compared with either the hardware supported reflective memory and memory channels, or the software supported distributed shared memory. Due to space limitations, we will not fully review these technologies and interested readers are refered to [7] for an in-depth comparison.

7 Concluding Remarks

We have described the implementation and preliminary performance results of MidART. As the application-to-application round trip delay tests show, our current implementation can support the class of real-time applications which MidART is designed for. These performance data were obtained without any fine tuning and optimization of our current initial implememtation. One of our next steps is to do a more thorough performance analysis to find out where we can optimize and streamlining our middleware operations.

It is known that although Windows NT provides real-time class priorities to threads and non-degradable priority scheduling, the OS itself does introduce priority inversion problems [8]. Thus another item of our current work is obtaining experimental data on how bad the priority inversion problem is with respect to MidART threads and its application threads. One possibility is that this priority inversion problem can be masked via careful design, e.g., rate control network I/O traffic, and limiting the amount of critical section and GUI activities. We are also studying an alternative of using one of the real-time Windows NT extensions [9].

As mentioned in Section 1.1, the uniqueness of MidART lies in the simplicity of services provided and the flexibility of data reflection models. Our current version has a total code size of only less than 215K bytes where the Global Server is 67.5K bytes, the Local Server 69.6K bytes, MidART library 77.4K bytes. For example, the applications we used in our RTT performance testing include an application writer process and a reader process (with various threads). These programs including the MidART library have sizes of 86.5K bytes for the writer and 73.2K bytes for the reader.

The source code of MidART can be obtained free of charge for research purpose with a source license agreement signed between the interested party and Mitsubishi Electric Information Technology America (ITA). See our web site at www.merl.com for future releases and license agreement.

References

[1] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, pages 18–28, February 1996.

[2] Digital Equiqment Corporation. Memory Channel Overview. *www.unix.digital.com/bin/textit/cluster/memchanl/memchanl.html*, May 29 1996.

[3] Object Management Group. *The Common Object Request Broker: Architecture and Specification, 2.0 ed.* July 1995.

[4] T. Harrison, D. Levine, and D. Schmidt. The Design and Performance of a Real-Time CORBA Event Service. In *OOPSLA '97*, October 1997.

[5] 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.

[6] D. Schmidt, A. Gokhale, T. Harrison, and G. Parulkar. A High-Performance Endsystem Architecture for Real-time CORBA. In *IEEE Communications Magazine*, volume 14, Feb. 1997.

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

[8] M. Timmerman and J-C. Monfret. Windows NT as Real-Time OS?. *Real-Time Magazine*, Q2 1997.

[9] M. Timmerman and J-C. Monfret. Windows NT Real-Time Extensions: an Overview. *Real-Time Magazine*, Q3 1997.

[10] VME Microsystems International Corporation. Reflective Memory Network. *White Paper*, February 1996.

Appendix

```
// Creates an entry in the global definition table.
CRM_RegisterService (void *lpServiceInfo, void *lpHostName)
// Deletes an entry in the global definition table.
CRM_UnRegisterService (void *lpServiceInfo, void *lpHostName)
// Creates a new reflective memory area in
// the global definition table.
CRM_Create (int Size, int Period, int CreationFlags,int nBuffers,
void *lpServiceInfo, void *lpLayoutInfo)
// Removes a reflective memory area
// and terminates all DPA threads and network connections.
CRM_Destroy (int mId)
// Allows local writers to map to an existing memory area.
CRM_Map(int mId, int nBuffers)
// Disassociate the calling thread and a reflective memory area.
CRM_UnMap (int mId)
// Creates a reflective memory area of m_H buffers.
// It attaches a reader's thread to the reflective memory area
// and establish network connections with the writer's ReMA.
CRM_Attach(int Period, int Deadline, int AttachFlags,
int Buffers, void *lpServiceInfo, void *lpElementsName)
// Detaches a reader thread from the reflective memory area
// by removing the associated DPA-thread and its connection.
CRM_Detach (int mId, boolean AllConnections)
// Activates the associated DPA thread on the writer's node.
CRM_Start (int mId)
// Fills the reader's buffers with existing data from
// the writer's buffers.
CRM_StartInitH (int mId, int HowMany);
// Halts the reflection of the memory area by suspending
// the associated DPA thread.
CRM_Stop (int mId)
// Reads a single memory buffer into the area specified by
// the call. The most recent available data is copied.
// This operation may block, depending on the AttachFlags
// used in CRM_Attach.
CRM_Read (int mId, void *lpBuffer, void *lpMemStatus,
void *lpElementsName)
// Reads h buffers counting back from the most recently updated
// into the area specified by the call.
CRM_ReadH (int mId, void *lpBuffer, int Hbuffers)
// Reads all available data in the buffers into lpBuffer.
CRM_ReadAll (int mId, void *lpBuffer, int HowMany)
// Writes data pointed by lpData into the reflective memory area.
CRM_Write (int mId, void *lpData)
// Resets all of the contents of the reflective memory area.
CRM_Reset (int mId)
// Allows an application to start using the services offered
// by the middleware library.
CRM_Startup (int InitFlags)
// Terminates the use of the middleware library.
CRM_Cleanup ();
```