DEVELOPMENT OF MIDDLEWARE FOR SOFTWARE OF UNIX BASED CONTROL SYSTEM

Hwawon Hwang, Yongsoo Kim

Instrumentation Research Group, Technical Research Lab,
POSCO, Pohang, SOUTH KOREA

Abstract: Standardization for development of open, economical and efficient control system has been progressed by international and domestic technical committee and numerous vendors around the world. As a major example, we can pick up OPC standard and pc-based control system. Following the advent of pc-based control system, there are a lot of movement of expanding linux based technology into automation field such as development of OPC for linux and linux based HMI. As the first step of coping with global movement of standardization and openness, middleware for management of system resources of MS Windows based control system was developed in POSCO. As the next step for building up standard framework for platform and vendor independent development of software , middleware for linux and UNIX based control system has been developed. We can get the easiness of management and development of software, built-up of control system with high level of extensibility and self-engineering power by development of the middleware. In this paper, development of UNIX based middleware will be introduced in detail. In the last part of this paper, evaluation of performance of middleware will be presented. *Copyright © 2005 IFAC*

Keywords: Computer Software, Process Control, Utility, Linux, Middleware, OPC, Task Management

1. INTRODUCTION

Standardization for development of open, economical and efficient control system has been progressed by international and domestic technical committee and numerous vendors around the world. As a major example, we can pick up OPC standard and pc-based control system. It is necessary to develop open control system using linux based technology as well as windows based technology for real open control systems which are platform and OS independent.

In the case of POSCO, several linux based system are used as process control station or process monitoring system. One example is the linux based HMI station in KwangYang No.2 Hot Rolling Mill. The other is linux based control system in Pohang No.1 Cold Rolling Mill.

When Windows or linux based systems are used as open control systems, there is a few major drawback in the view point of application developer such as different system calls according to operating systems, complicated procedures for use of system resources (memory, file, IPC, remote communication) and difficulty of monitoring and updating of resources which are used for application processes. To solve problems mentioned above, middleware for windows based control system was developed and standardized, which manages system resources and provides application developer with easiness of development of application using system resources. As the next step towards standardization of platform independent middleware, linux based middleware was also developed, being based on the previous windows based middleware system. And then UNIX based middleware system was redesigned and implemented on SUN Solaris and IBM AIX operating systems.

We can get the easiness of management and development of software, built-up of control system with high level of extensibility and self-engineering

power by development of the middleware.
In the next section , the detailed functions of middleware system will be explained and then results of performance test of middleware will be analyzed in section 3, Lastly, future development plan will be mentioned.

2. DEVELOPMENT OF MIDDLEWARE FOR INTEGRATION AND DEVELOPMENT OF APPLICATION SYSTEMS (MIDAS)

MIDAS can provide users with convenience of management and development of software by implementing management functions itemized as follows.

- Management of message exchanges among tasks
- Management of periodic and non-periodic timers
- Management of access to three kinds of files which are named as record file, cyclic file and index file.
- Management of creation and access of shared memory, implementing easy inter-process communication among tasks.
- Adds-on utility graphically showing usage status of system resources and user API(Application Program Interface) helping users to easily access to system resources.

On middleware server system, MIDAS kernel is running with implementing management functions. And user application programs can be programmed using user APIs on client system. Middleware API provided by MIDAS are as follows.

- API for management of tasks
  - Task Registration, Task Deletion, Task suspension,
  - Task Resumption, Task Locking and Unlocking
- API related with message exchanges
  - Asynchronous Message Sending/Receiving,
  - Synchronous Message Sending/Receiving
- API for Timer Management
  - Periodic Timer
  - Non-Periodic Timer
- API for access to files
  - File Open, File Close
  - File Lock/Unlock, Record Lock/Unlock
  - Record Reading/Writing (Three kinds of access methods exist: using record number, read/write pointers or key value)
- API for Access to Memory
  - Shared Memory Reading/Writing

2.1 Task Management

Control flow of task management functions of Middleware is shown in FIG1. If MIDAS kernel receives a request of task registration from a user

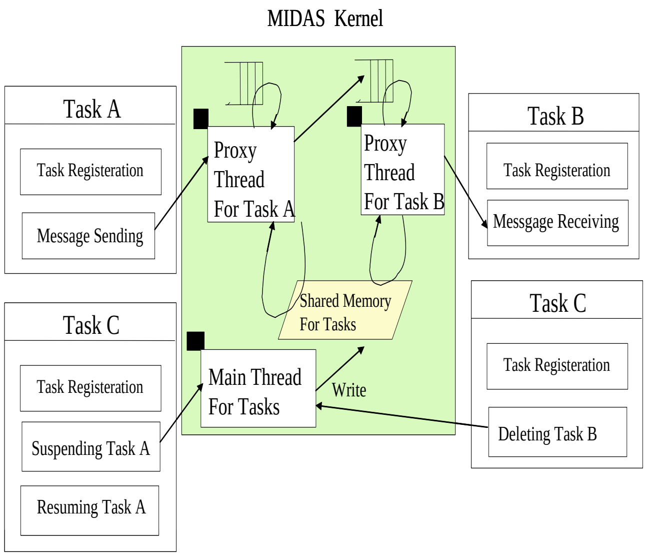


Fig. 1. Control flow of task management

application program, kernel creates proxy thread which handles the user’s requests. The requests related with task operations such as suspension, resumption, deletion and etc are delivered to the corresponding proxy thread through main thread. And then, the proxy thread executes functions handling the requests.

Middleware kernel uses communication method which is supported inside kernel for event delivery and data exchanges among tasks. The following is the operational procedure occurred inside kernel for a message exchange between two user application tasks.

If middleware kernel receives requests of task registration from user application programs called A and B which exchange data , kernel creates two proxy threads which will handle user ‘s requests from task A and B. And then message exchanges are executed by the following procedures. Types of message exchanges are classified as synchronous exchanges and asynchronous exchanges.

- Asynchronous message exchanges :
  Sender task A stores a message in the message queue of proxy thread B by proxy thread A and returns without waiting for acknowledgement message from task B.
  Receiver task B processes the message from task A in its own proxy’s message queue if the message from task A is the only message existing in its queue. But if there are several messages to be processed in the message queue, messages will be processed sequentially in FIFO(First in First Out) method.
- Synchronous message exchanges :
  Synchronous message exchanges are a little different from asynchronous message exchanges in that sender task A is waiting for a response from task B. If there is no response from receiver task, sender task enters into waiting mode until receiving any response.

User application tasks may or may not reside in the same system where middleware exists. Therefore, middleware kernel internally uses two kinds of methods for communication between kernel and client tasks. One is socket communication method for remote clients, the other is UNIX socket communication for local clients, which is one of the inter process communication methods on UNIX. The reason why middleware kernel uses a separate method for local clients without using socket method is that inter-process communication is 2 or 3 times faster than remote communication such as socket communication method.

UNIX socket method uses the same type of API functions as general socket interface method but has different implementation internally. Message Queue is another option for implementing message exchanges between user application tasks, which shows excellent performance in the speed of communication. But only maximum 16 message queues can be created in one system and one message queue can stores maximum 32K bytes. So, message queue communication method is not adequate for middleware in which several 64K bytes data have to be stored and exchanged.

we decided to use UNIX socket for supporting message exchanges between local clients because UNIX socket communication method shows similar performance to the performance of message queue.

2.2 Timer Management

Timer provided by middleware kernel is categorized according to whether timer event is issued periodically or not. And exact time of issuing timer event is specified as absolute time or offset from the current time of request.

If a user application task requests timer service through middleware API, the proxy thread in middleare kernel corresponding to the application task records type of the timer and time of issuing timer event in the proxy thread's timer management table . When calling timer service request, User can specify type of timer, time of issuing timer event and the application task which will receive the timer event message.

In the case of expiring the time, the proxy thread of which application task requested timer service, sends timer event to the application task which will receive the timer event message. If the user application task requested non-periodic timer service, the proxy thread handling the timer service deletes timer service entry in their timer management table after the first issuing of timer event. But in the case of periodic timer, the entry is not deleted until user application task calls API for deletion of timer service.

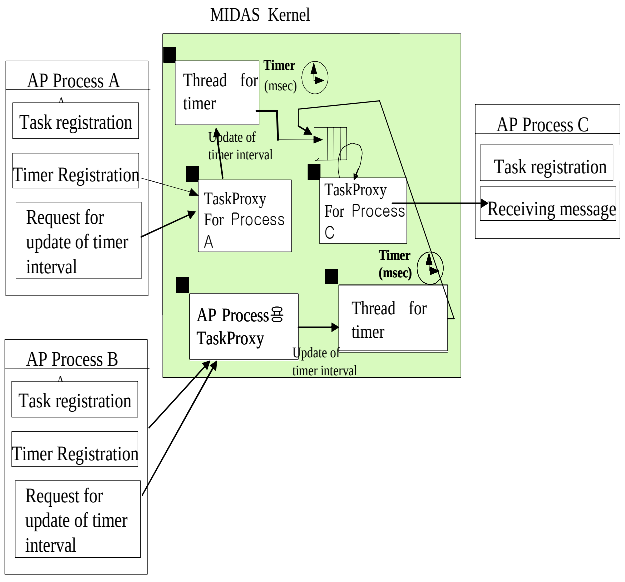


Fig. 2. Control flow of timer management

2.3 File Management

Middleware provides users with three types of file called record file, cyclic file and index file according to file access methods and storage medium. Common characteristic of every type of files is that the smallest unit of read/Write operation is a record. As we can see the structure of file provided by kernel in FIG.3, MIDAS file is composed of header area, sub header area and record area regardless of file types. Record area can varies in size by the number of records and record sizes. Record file is accessed by record number while cyclic file can be accessed by read/write pointer. And index file can be accessed by key value of a record. So index file is very efficient file type when users need to quickly search the record with specific key value.

To implement read/write pointer of cyclic file and key index of index file, a record in FIG.3 is composed of two fields. One is key field which stores key value uniquely identifying the record of index file or time stamp of read and write operation of the record in the case of cyclic file. And the other is record field which stores record values. Time stamp stored in key field contains information about the time when write and read operation of record happened. So, system can recognize the record which can be read next time and the empty positon where new records can be written.

All three types of files are physically stored in storage medium in three different ways. In one way, record is read and written directly from disk. In second way, record is read from memory and written into memory while middleware is running and then finally written into disk when middleware kernel stops its operation. In third way, record is repeatedly written into both disk and memory but read operation of record is only done on memory.

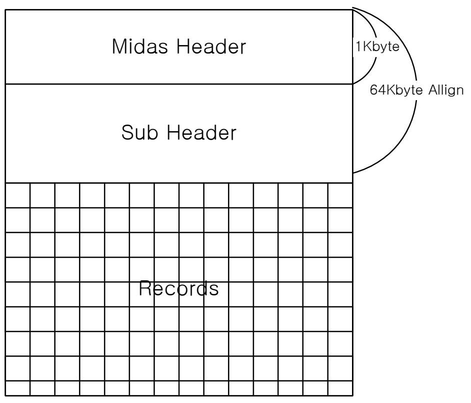


Fig. 3. The structure of file provided by MIDAS kernel

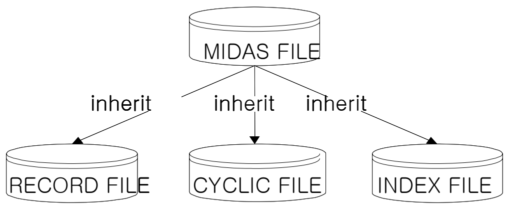


Fig. 4. Class structure of middleware file system

All three types of files inherit from the abstract class “midas file” as child class of midas file. Midas file class has common characteristics of three file types and implements basic file operations which files of child class internally use when handling user’s request of file operation such as file open/close, record read/write and file or record lock/unlock. Although results of file operation are similar to each other regardless of file types, little difference exists when executing the operation. The following items are explaining file operation classified by file types.

- File open and close operation : opening and closing file specified by user.
- Record read and write operation: reading and writing a record or several records from file
  - Record file: A record or several records can be read and written by record number specified by user. And also several records can be read or written starting from the position located in the offset from the specific record number.
  - Cyclic file : Read and write operation of records begin from the position pointed by read/write pointer. After read /write operation of record, read/write pointer increases by one. Addtion to read / write operation of records, users can get the number of valid records which are already written in file but not yet read.
  - Index file: Read/write operation of record is done by key value specified by user. As record searching has to be done by key value before reading or writing operation, registration of key values are needed when records are written.
- Record or file lock/unlock: locking or unlocking records and files.

In the case of index file, record searching by key value has to be done before read or write operation. So, key searching is vital to the fast access to records. In middleware, key value is stored in sorted array and binary searching method is used for fast key value – record matching.

2.4 Memory Management

Users can share data among tasks using shared memory provided by middleware. Middleware makes shared memory using unix function, shmget() during kernel initialization, referencing header file which defines record structure of shared memory. The header file usually is defined by users. Users have to also define key number of shared memory in the procedure of defining structure of records which will be stored in shared memory. This key number will be recorded in shared memory mapping table with id and start address of shared memory and used to get the physical address of shared memory.

3. PERFORMANCE TEST

3.1 Performance of Middleware System

UNIX based middleware system was developed on SUN solaris and IBM AIX operating system. So, Performance of middleware system was tested on both operating systems. The UNIX servers with the following specificaiton was used in the performance test.

[IBM AIX 5.1]

- CPU: 1.2GHz Power4
- Memory: 1 Gbytes DIMM
- Hard Disk: 36.4 Gbytes,Ultra3 SCSI, 10,000 RPM

[SUN Solaris8]

- CPU: 1.2GHz UltraSparcIII
- Memory: 1 Gbytes, DIMM
- Hard Disk: 73 Gbytes,Ultra/Fast/Wide SCSI, 10,000 RPM

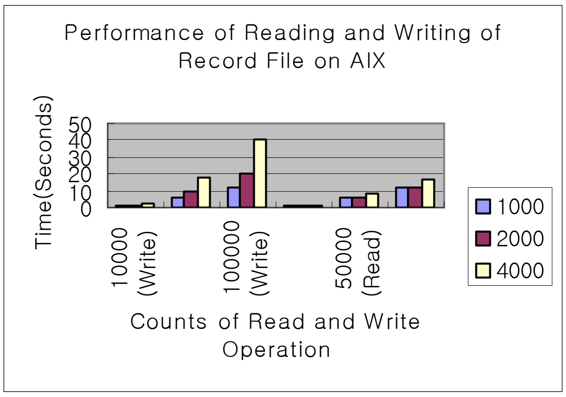
client applications for test was developed in middleware C API on the client system with the following specification.

- OS: Linux Redhat9.0
- CPU: Intel Xeon Dual CPU 2.8GHz
- Memory: 1GB

Middleware system resides on the UNIX servers. Client applications using services of middleware can reside in the same system or in remote system. Performance of middleware system can be categorized as performance of file system, message exchange between tasks and timer accuracy.

3.2 Performance of File Management System

Performance of middleware file system was tested with the number of read and write operations being increased from 10000 to 100000. And time of file read and write opreations was measured in the cases of twenty seven combinations of three kinds of file types such as Record, cyclic and index file, file



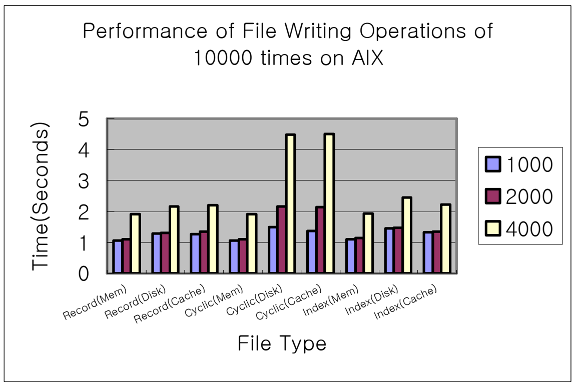


Fig. 5. Results of test for performance of file system

size(1Kbytes, 2Kbytes and 4Kbytes) and storage type of records such as memory, disk and cache type.
As a result of the test , we can get that it taskes 0.4 milli seconds to complete one write operation of a record with record size of 4Kbytes in record file on AIX . In the case of test on sun solaris, one write operation of a record with 4kbytes was completed in 0.35milli seconds. And a write operation costs more time than a read operation when file format and storage type are the same.
Analyzing the performance of file system of middleware in regard with storage type, 10000 times of write operations of records in memory type file show the least processing time than any other file type but shows a little difference because of the overall overhead of communication between middleware server and application requesting file system service.
Read and write operations of cyclic file costs more time than record file or index file. Test of record and index file shows similar result.

3.3 Performance of Task Management

Performance test of task management was executed in the environment where only testing system (a server and two clients) was connected on one network hub without no background network traffic.
User applicatons locally communicating with middleware was programmed in middleware API for UNIX(aIX, solaris) and gcc v3.2 on UNIX system where middleware server resides. And User applicatons remotely communicating with middleware was programmed in middleware API for linux and gcc v3.2

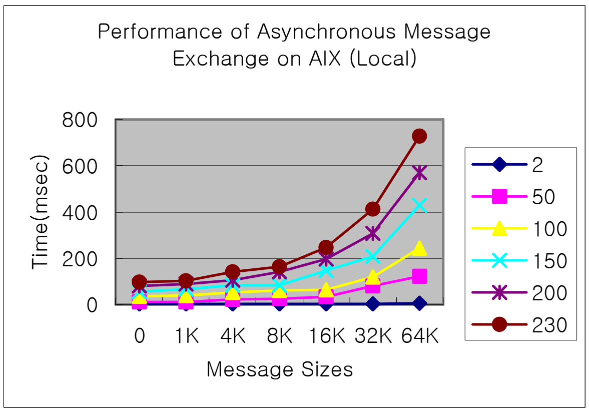
Test for Performance of Message Exchanges among Tasks

Test in this section shows how system performance varies as the number of task running and message sizes increase when tasks constantly exchange messages without taking a break. One return time of message exchange is defined as the time which takes for task A to receive the response from task B after sending a message to task B. Task B immediately sends a response to task A. All tasks have a message queue with the size of 100. For precise test, one return time of message exchange was calculated from average time of 100 numbers of message exchanges from 101th to 200th message exchanges.

As you can see in the graph, test of message exchanges through middleware system shows performance within 800 milli seconds in the case of middlware and communicating applications on the same system.
As number of tasks and message size increases, time spent in communication linearly increases as shown in FIG.6. In local communication between middleware and applications, better performance of middleware on AIX can be explained by difference of thread scope which were set in the time of development of middleware on Operating system. –AIX thread scope was single in which each user threads have correspoding kernel threads inside kernel layer while several user threads match onto one kernel threads in multi thread scope. AIX threads are allocated CPU time evenly with other processes while child threads of a process on solaris shares

CPU time allocated to parent process among child threads. Middleware kernel creates a proxy thread whenever requests are coming from user applications. Therefore, middleware for AIX shows better performance than AIX.

Another thing we have to notice is that time suddenly rises when message size increases from 4K to 8K bytes in remote communication between middleware and application. This phenomenon happens when task



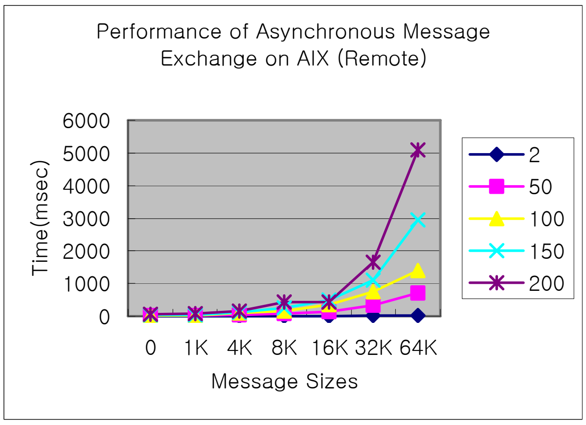


Fig. 6. Performance of message exchange among tasks

numbers are 50 ~ 230. One reason is that retransmission of messages occurs very often by ethernet collison as number of tasks increases and too many messages are transmitted on network. Actually we could find out that network utilization was over 50% for more than 50 tasks exchanging messages each other.

3.4 Performance test of timer accuracy

Table 1 Test result of accuracy of timer service of middleware

Interval Number of tasks	50 ms	100 ms	500 ms	1000 ms
1	49.98	99.97	499.9	999.8
20	49.97	99.96	499.9	999.8
64	49.96	99.95	499.9	999.8

To test the accuracy of timer, two kinds of test was performed. One is to test if timer event from middleware was sent to the application requesting timer service within timer inteval when serveral other applications also get timer events from middleware. The result of irst test was shown in figure. Another is to test if timer event from middleware was sent to the application requesting timer service within timer inteval when serveral other applications are exchanging messages in busy way through middleware communication service.

As seen in the table, we can see that tasks requesting timer service can get timer service of middleware within timer interval although the number of application requesting timer service increases

4. CONCLUSION AND FUTURE PLAN

UNIX based middleware was developed for built-up of cost-effective open control system with convenience of maintenance and development and standardized middleware with platform independence. Middleware has several management functions of task, timer, file and memory and adds-on utilitys such as logviewer and maintenance system.

To review performance of middleware, file system and message exchange among taks was tested. Time for message exchanges increases linearly as the number of tasks and message size becomes larger. Performance of UNIX based middleware goes down in the case of consecutive message exchange among over 100 tasks by processing overhead. And time suddenly rises when message size increases from 1K to 4K bytes and number of tasks is over 50 ~ 100 for UNIX based middleware. This is caused by several retransmission of messages by ethernet collison as too many messages are transmitted on network. We can also get the result from performance test of message exchanges that best performance of message exchange can be maintained , not giving CPU processing overhead regardless of number tasks if given delay time of 5 msec between each messages exchanges. And we can also make sure that queue overflow never happens when messages are exchanged one to one if size of message queue is over 5. After performance test of file system on middleware, we can get the result that types of file doesn’t play a major role in overall performance of file system on middleware because communication overhead of middleware is too large enough.

UNIX based middleware developed in this project will be used as middleware for open process control system to be newly installed in steel works. And mddleware will be added with new functionality such as on-line backup functionality and support for databases.

REFERENCES

Kangje,Cho (1997). development of network functionality of PC middleware for PCS, POSLAB POSCO.
FSMLabs Inc. (2001). *Getting Started with RTLinux*.
Gavin Smith (2002). *Linux Kernel Programming*, chapter 2, O’Reilly