BROWN UNIVERSITY
Department of Computer Science
Master's Project
CS-96-M2

"Mini-Distributed System (MDS)

by

Jasper Y. Wong

Mini-Distributed System (MDS)

Jasper Y. Wong
Department of Computer Science
Brown University
Providence, RI 02912, USA

August 31st,1995

Mini-Distributed System (MDS)

Jasper Y. Wong

Department of Computer Science
Brown University

Submitted in partial fulfillment of the requirements for the Degree of Master of Science in the Department of Computer Science at Brown University

August 1995

[illegible]

Professor Robert H.B. Netzer

Advisor

Table of Contents

Purpose

This paper serves as the project report to the Master's project Mini-Distributed System. The goal of the project is to bring some form of distributed computation to the home computing environment. The Amiga computer platform was used as the target system for this project.

The objective of Mini-Distributed System (MDS) is to mimic the internal message system used by the Amiga's operating system on the surface and exchange messages among networked Amiga's via TCP/IP socket interface. Thus, application programs can be developed to take advantage of distributed computation without knowing how to do TCP/IP socket programming. This report will show that our current implementation of MDS gives adequate performance on a 2-node network. Network with more nodes were not tested due to lack of available machines.

1 Introduction

As the availability of inexpensive processors has made possible the construction of distributed systems that were previously economically infeasible, networking personal computers is becoming a common trend in home computing. Thus, distributed programming no longer is a priviledge enjoyed only by those who program on high-end workstations; it has become potentially accessible to programmers of all levels. In light of the increased raw processing power provided by networked home computers, it seems appropriate to explore distributed computing on these computers.

1.1 Distributed Computing Systems

Distributed computing is a form of collaborated computing contributed by different computers connected by network. These computers communicate with one another by exchanging messages. A standalone computer can also be viewed as a distributed system with processes distributed internally.

Several advantages can be gained with distributed computation. A task can be divided up into several pieces and each piece be computed by a different processor such that less time is required for each task to be completed. Another advantage is the simulation of protected memory. Failure of one server does not have to bring down the whole network. Lost task can be redistributed to another server to perform. Resource can also be shared.

Overall, the goal of distributed computing is to combine resources and processing powers of interconnected computers such that a task can be performed faster and be able to access to more resources without the cost of upgrading a standalone system to provide equivalent performance.

1.2 Issues surrounding Distributed Systems.

Many issues arise in dealing with distributed systems. Synchronization, partial ordering, system failures, and distributed debugging represent just some of the issues MDS needs to address.

- Synchronization and Partial ordering

Processes are usually executed with different speeds, especially when they're executed on different computers, hence synchronization is often required to manage communication among processes. In order for two processes to communicate, these two processes must find some common memory block where both processes can access and detect. Then one process may arrange something to take place in that particular memory block so that another process may detect and response such that two way communication can be established. It is important that the events of performing an action and detecting an action are constrained to happen in that order. Synchronization can then be defined as a set of constraints on the ordering of events and synchronization mechanism can be implemented to satisfy these constraints by having the processes to wait for each other during execution. MDS relies heavily on the synchronization of its processes.

The ordering of events can be specified as an event **a** happened before an event **b** if **a** happened at an earlier time than **b**. But how can we tell if an event happened before another event? Some form of observable events within the system are needed to determine the relationship between the two events in question. A physical clock is psychologically the most ideal candidate. However, networked computers do not share the same physical clock, nor can they keep precise physical time without network delay even if they do share one physical clock. Thus, the "happened before" relation is usually defined without using clocks and is denoted by "->". Partial ordering is then defined as a "happen before" relation at any point of time satisfying the following conditions: (Figure 1 illustrates the four conditions)

(1) If **a** and **b** are events in the same process, and **a** comes before **b**, then **a->b**.

(2) If **a** is the sending of a message by one process and **b** is the receipt of the same message by another process, then **a->b**.

(3) If **a->b** and **b->c** then **a->c**.

(4) Two distinct events **a** and **b** are said to be concurrent if **a** !->**b** and **b** !-> **a**.

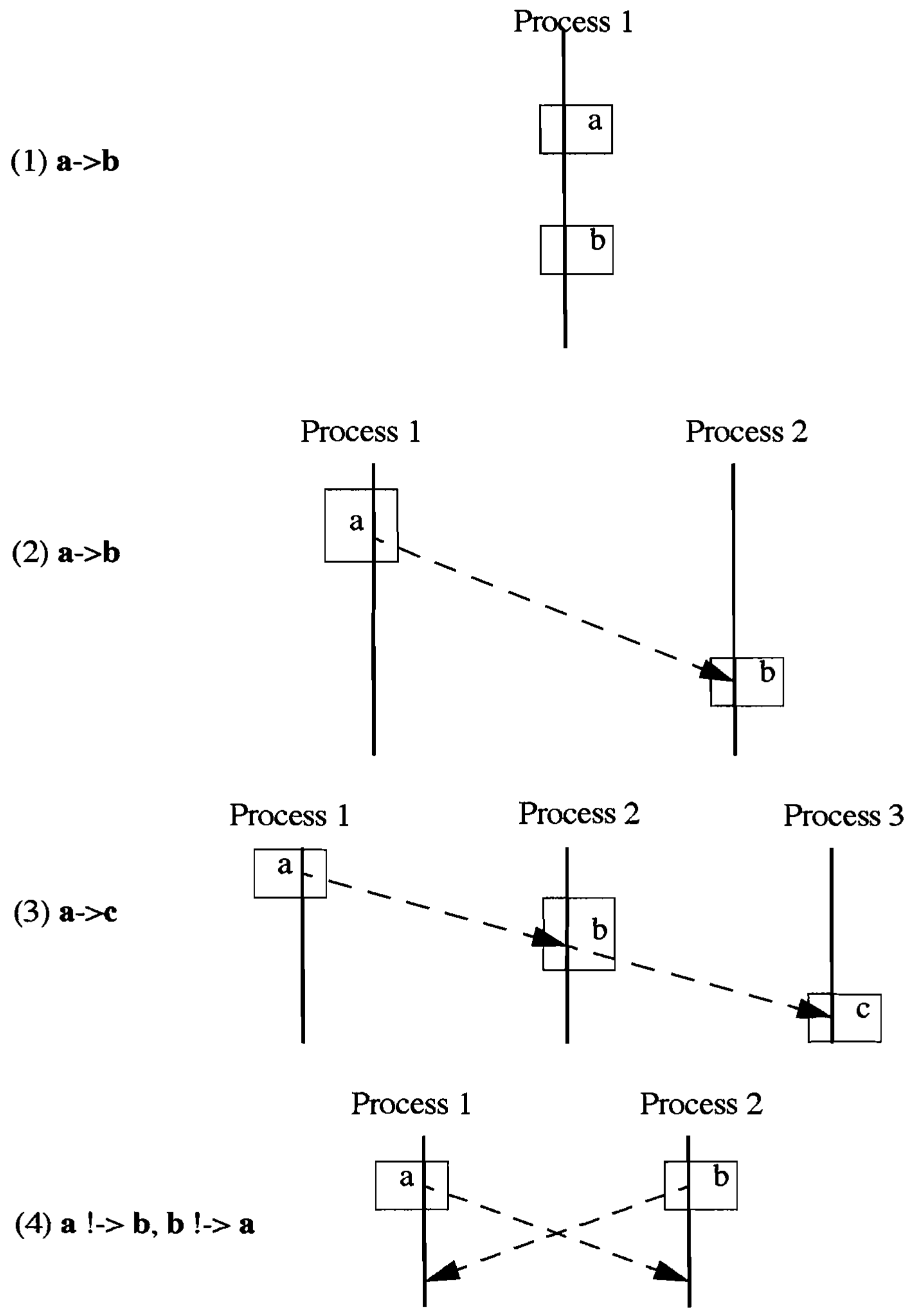

Figure 1. Different forms of partial ordering

Partial ordering serves as the fundmental building blocks for the design of synchronization mechanism as well as other topics related to distributed system and it's essential to developing on MDS.

- Distributed debugging

Program which is designed to run on standalone system exhibits same behavior when executed repeatedly. Distributed program, on the other hand, may generate different results when executed more than once. Its nondeterministic behavior can be caused by variations in process scheduling and message latencies.

There exists several strategies for nondeterministic behavior detection and they can basically categorized into the followings:

Compile time analysis

It examines program semantics and explores all possible executions.

Trace analysis

It examines execution logs generated by various exections of program in question to determine nondeterminacy.

On-the-fly detection

It examines information collected during program execution and detects nondeterminacy by message tracing and other methods.

In general, when debugging mechanisms are made of program operation actual execution behavior can be perturbed. If the pertubation effects of the instrumentation cannot be quantified by a perturbation model, information obtained through the debugging instrument could be inaccurate. Ideally, the debugging mechanism should be a separate entity from the actual execution model. The perturbation effects can also be quantified by implementing controlled execution in the underlying mail system, allowing the debug technique to be nonintrusive.

The future design of MDS allows for message logging and automatic replay thus allows for easier implementation of these debug strategies.

- System failure

Networked computers are frequented by system failure. A loose connection, nameserver failure, system overload, or simply system failure could all lead to a halt in a distributed environment. When a distributed program encounters system failures, procedures should be established to ensure the integrity of the program.

System failures fall into two main categories: (1) Server failure and (2) Server delay.

(1) A distributed task could be lost when a server is down and disappeared from the network. A failure with the nameserver may lead to connection failure to all remote servers. Thus, an ability to recover from system failure is needed when implementing distributed programs.

(2) Server delay can be considered as a form of server failure with remote tasks eventually routed back to the host server when completed. If a host server assigned remote task **a** to server **1**, and then server **1** became offline for a period of time such that the host server reassigned the task **a** to server **2** to perform, then a conflict may appear when server **1** became back online and send

back the result of task **a**, thus there exists two results from two completed task **a**'s.

1.3 Distributed Environment on micro-computers

With the increased popularity of home grown local area network (LAN), it has become beneficial to low-end computers to tap into the power of distributed computing as well. However, unlike their superset counterpart, most operating systems used on low-end computers weren't designed with distrbuted environment in mind, and built-in networking software has just begun to take shape on these operating systems. Thus, providing distributed computing to these systems requires careful planning and cooperation from application softwares in order to ensure efficiency and OS-friendliness.

One important feature is required of the operating system in order to provide a functional minimal distributed environment. A network protocol capable of exchanging messages is the sinlge most important feature needed to distribute tasks. Other features such as remote printing, task distribution, shared file system...can be implemented with a capable network protocol. Most distributed systems, such as PVM and some others, are based on some form of UN*X environments, thus internal multitasking can be assumed. However, quite a few operating systems for micro computers weren't designed to take upon more than one task at a time. These systems require a system level add-on which provides some form of time-slice scheduler in order to accept remote distributed requests as well as to serve as a regular computer (such as DesqView for MS-DOS computers). Implementation of distributed system is not recommended on these systems as failure in one task may lead to instability to the entire system.

1.4 Overview

In this section we describe in brief the structure of a distributed system and some related issues affecting the design and implementation of such a system. In section 2 we describe Mini-Distributed System and how it is integrated with the Amiga's OS architecture. Then in Section 3 we examine the performance of MDS and discuss further improvements. Section 4 we explore a more complete design of MDS. Section 5 gives a brief walk-thru on how to write MDS applications. Finally, Section 6 gives a summary and conclusion.

2 Mini-Distributed System

Our goal is to design and implement a distributed system that is friendly to the Amiga operating system used on some micro computers. That is, to bring high-level connectivity and the power of distributed computing to these computers without the need to perform complex socket programming. Such a system can be viewed as a general and flexible distributed resource that supports a message-passing scheme of computation. The simulation of message-passing permits system adaptability while retaining high level of transparency with the underlying operating system.

Application programs written under Mini-Distributed System (MDS) are structured as master/slave model in which a set of sub-processes performs task for the master process. Processes communicate with each other via an enhanced message structure. Several restrictions were imposed to ensure overall efficiency and transparency.

2.1 Amiga Operating System

In order to design a distributed environment based on the Amiga operating system it only makes sense to understand some essential elements of the Amiga operating system such as its multitasking environment, its lack of memory protection...etc.

2.1.1 Operating System Versions

The Amiga operating system has undergone several major revisions. The latest revision is Release 3.1 (corresponds to library 39 and above). MDS requires OS release 3.1 or above to function properly.

2.1.2 Multitasking

The core of the Amiga's operating system is *Multitasking Executive*, usually known as *Exec*. *Exec* serves as a central system in the Amiga and other systems in the Amiga rely on it to perform correctly. Tasks such as multitask scheduling, message-based interprocess communication management, the allocation/deallocation of system resources and their accessibilities are all controlled by *Exec*.

Traditional micro computer's operating system often sent the computer to perform some non-computational intensive tasks which require spending a lot of its time waiting for something to happen. Wait for user to press some keys on the keyboard, wait for the mouse to click on an icon, wait for data to come in through the serial port, wait for text to go out to a printer...just to name a few. It is, therefore, logical to think that in order to make efficient use of the CPU's time, the operating system should have the CPU carry out some other tasks while it is waiting for some particular events to occur.

By switching to another program when the current one needs to wait for some events to occur or when a set time limit is reached, a multitasking operating system minimizes the amount of time it stays in idle stage. A multitasking operating system allows multiple tasks, or programs, to be executed simultaneously. These tasks can be run independently from each other, allowing each task to treat itself as if it's the only program executed on the computer, thus having the computer all to itself.

The Amiga's multitasking system relies on switching the task with the highest priority to use the CPU to achieve multitasking, but only if the task is ready to run. A task can be in one of three states: *ready*, *sleeping*, or *running*.

- *Ready Task*

The task is considered to be waiting for the processor but not currently using the CPU. A list is kept by Exec which includes all the tasks that are ready. Task priority serves as the sort key which Exec uses, thus Exec wastes no time in searching for ready task with the highest priority as the task would be the one on the top of the list. Exec then switches the task that currently has control of the CPU, and sends the ready task with the highest priority to the CPU.

- *Sleeping Task*

The task is not currently running and is waiting for some event to happen. Exec will move the sleeping task into the list of ready tasks according to its priority when that event occurs.

- *Running Task*

The task is currently using the CPU. It will remain the current task until one of the following conditions occur:

* A higher priority task becomes ready, so the OS preempts the current task and switches

to the higher priority task.

* The current running task needs to wait for an event, so it goes to sleep mode and Exec switches to the highest priority task in Exec's ready list.

* There is another task of equal priority ready to take control of the CPU and the current running task has had control of the CPU for at least a preset time period called a *quantum*.. In this case, Exec will preempt the current task for the ready one with the same priority; a technique known as *t*ime-slicing. When there is a group of tasks of equal priority on the top of the ready list, Exec will cycle through them, letting each one use the CPU for a quantum (a slice of time).

The generic conecept of task is often represented interchangeably by the terms "task" and "process". On the Amiga, Exec considers both of them to be tasks.

2.1.3 Libraries of Functions

Similar to most of the modern operating systems, quite a few of the routines that make up the Amiga's operating system are organized into groups called shared libraries. Unless the library has already been open, in order to call a function on the Amiga one must first open the library, or re-enter the library, that contains the function. For example, if one wants to call the Read() function to read data from disk one must first opens the DOS library.

The system's master library, called Exec, is always open. Exec keeps track of all the other libraries and is in charge of opening and closing them. One Exec function, OpenLibrary(), is used to open all the other libraries.

There exists another type of library known as a link library. Even though a link library is a group of functions just like a run-time library, there are some major differences in the two types (current version of MDS is implemented as a link library):

- Run-time libraries

A run-time, or shared library is a group of functions managed by Exec that resides either in ROM or on disk. A run-time library must be opened before it can be used. The functions in a run-time library are accessed dynamically at run-time and can be used by many programs at once even though only one copy of the library is in memory. A disk based run-time library is loaded into memory only if requested by a program and can be automatically flushed from memory when no longer needed.

- Link libraries

A link library is a group of functions on disk that are managed by the compiler at link time. Link libraries do not have to be opened before they are used, instead one must link one's code with the library when compiling a program. The functions in a link library are actually copied into every program that uses them.

Following is the Amiga system software hierarchy

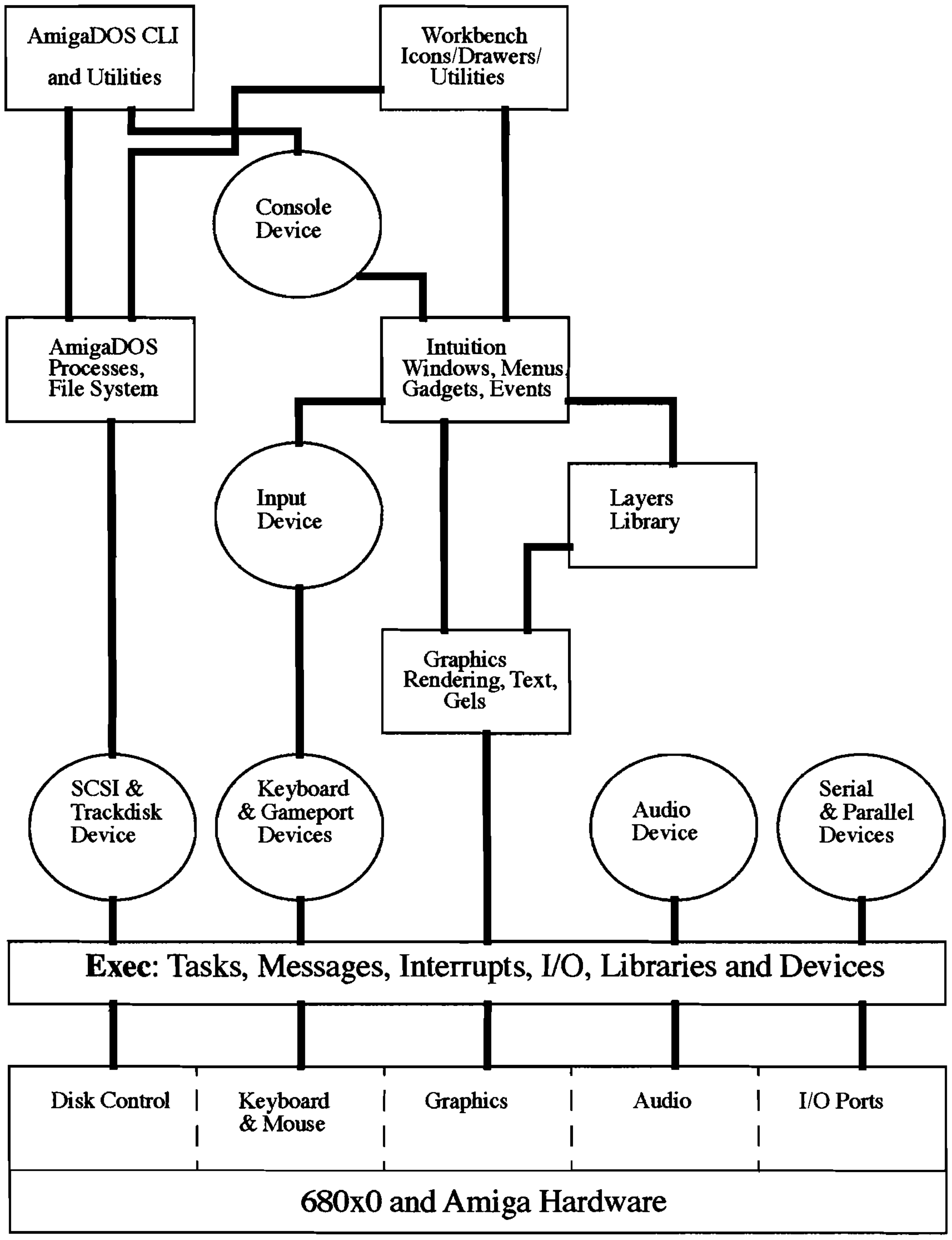
AmigaDOS CLI and Utilities
Workbench Icons/Drawers/ Utilities
Console Device
AmigaDOS Processes, File System
Intuition Windows, Menus, Gadgets, Events
Input Device
Layers Library
Graphics Rendering, Text, Gels
SCSI & Trackdisk Device
Keyboard & Gameport Devices
Audio Device
Serial & Parallel Devices
Exec: Tasks, Messages, Interrupts, I/O, Libraries and Devices
Disk Control
Keyboard & Mouse
Graphics
Audio
I/O Ports
680x0 and Amiga Hardware

Figure 2. Amiga System Software Hierarchy

2.1.4 Messages and Ports (interprocess communications)

Interprocess communication serves as the foundation for distributed computing, and Exec provides a consistent, expandable but somewhat lacking mechanism of messages and ports. This mechanism is used to pass message structures of arbitrary sizes from task to task, interrupt to task, or task to software interrupt. In addition, messages are often used to coordinate operations between cooperating tasks.

A *message* data structure has two parts: system linkage and message body. The system linkage is used by Exec to attach a given message to its destination. The message body contains the actual data of interest. The message body is any arbitrary data up to 64K bytes in size. The message body data can include pointers to other data blocks of any size within the same computer. MDS simulates *message* data structure but both system linkage and message body are sent to its destination.

A predetermined destination *port.* is always required when messages are to be sent. At a port, incoming messages are queued in a first-in-first-out (FIFO) order. There are no system restrictions on the number of ports or the number of messages that may be queued to a port.

Messages are always queued by *reference*, i.e., by a pointer to the message. It can be considered as a praise or a curse that for performance reasons message body duplication is not performed. However, it also presents a problem when we try to adapt network message. In essence, a message between two tasks is a temporary license for the receiving task to use a portion of the memory space of the sending task; that portion being the message itself. This means that if A sends a message to task B, the message is still part of task A context. Task A, however, should not access the message until it has been *replied*; that is, until task B has sent the message back, using the ReplyMsg() function. This technique of message exchange imposes important restrictions on message access but helps us in enforcing synchronization.

Message ports are redezvous points at which messages are collected. A port may contain any number of outstanding messages from many different originators. When a message arrives at a port, the message is appended to the end of the list of messages for that port, and a prespecified arrival action is invoked.

2.2 Design of Mini-Distributed System

MDS takes advantage of the multitasking ability of the Amiga's operating system to provide a functional distributed environment, it focuses on the following key elements to ensure usability, efficiency, and transparency.

- Simulation of standard Amiga's message interface on TCP/IP network
- Local and remote process tracking
- Network resource tracking

2.2.1 Simulation of Amiga's message interface

Section 2.1.4 described the data structure of message used on the Amiga operating system. Its structure has two parts: system linkage and message body. This data structure presents a few problems when a process running on computer A trying to receive message from another remote computer B. First of all, the two computers are connected via TCP/IP protocol thus standard operating system's messages and ports cannot be used. Secondly, even if the standard method works, the inability for the Amiga computers to share memory prevents computer A from accessing the message body on computer B. Therefore, designing a distributed message interface requires simulating the data structure of message via TCP/IP and the ability to pass both system linkage and message body.

The MDS system takes the host IP address, destination IP address, socket information and create a static link between the host and the remote servers. The body of a message is transmitted as opposed to just the system linkage. The original message body is disposed such that no excessive memory space can be wasted. The message body can be regained when it's been replied. Application programming interface for message exchanges has also been enhanced to handle the additional information. An API's server is also used to handle the translation of messages from standard messages to TCP/IP's socket messages and vice-versa.

Message body is splited into chunks of data and transmitted through the network as a special type of message or being written to a temporary file shared by the destinated computer via some form of shared file system.

2.2.2 Local and remote process tracking

In order to ensure adequate distributed performance, it is important not to overload any remote servers and to maintain a healthy track of local processes. Good process tracking and message tracking also allows for flexible distributed debugging by providing execution information when needed.

2.2.3 Network resource tracking

Server failure or the additional of new servers may happen any time during the uptime of the MDS system. It is, therefore, reasonable to keep track of the availability of servers as well as other network resources. A resource manager is used to broadcast changes in the network.

2.2.4 Structure of Mini-Distributed System

The Mini-Distributed System is broken into the following pieces:

- *Task Manager*
- *Resource Manager*
- *API library*
- *MDS Daemon*

Figure 3 outlines their relationship and the flow of message exchange.

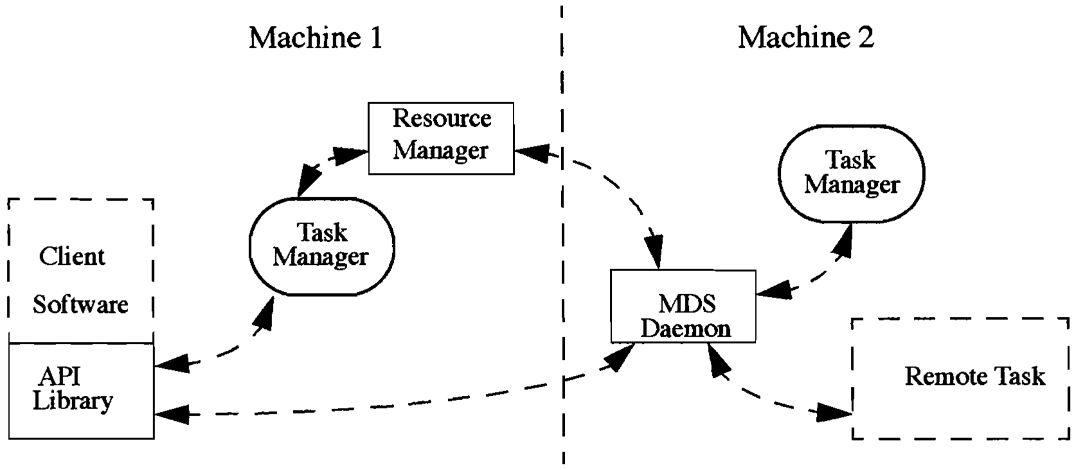

Figure 3. Overview of Mini-Distributed System

2.2.4.1 Task Manager

Task Manager resides on each computer and keeps a list of current registered local processes and a list of registered remote processes. It serves as the main communication center on each computer responsible to arrange communication among the resource manager, the API library, and the MDS daemon. The following flow chart outlines Task Manager's operation.

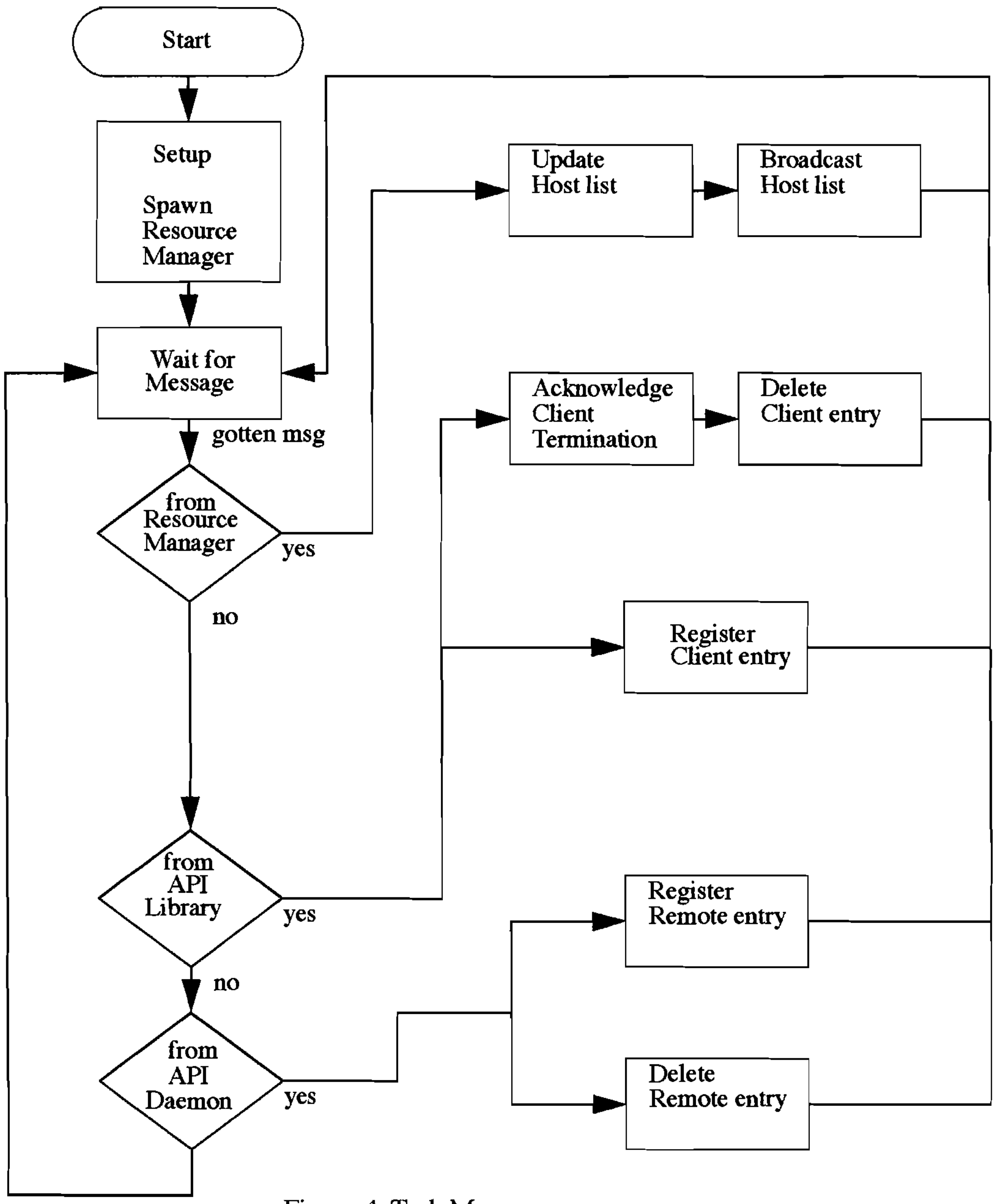

Figure 4. Task Manager

2.2.4.2 Resource Manager

Resource Manager keeps track of available servers and their CPU loads. It broadcasts its availability to the network and listens for broadcasts from other resource managers. CPU load is based on the number of local and remote processes tracked by Task Manager. The following flow chart outlines Resource Manager's operation.

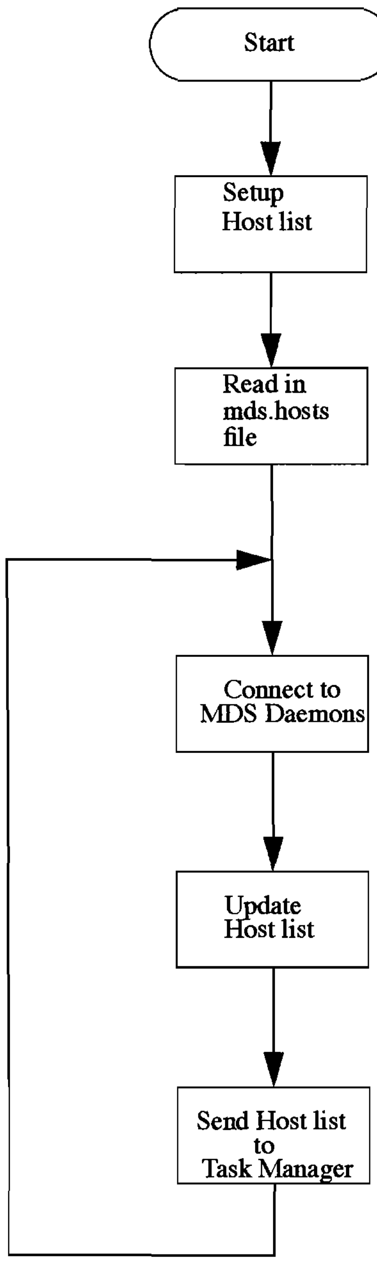

Figure 5. Resource Manager

2.2.4.3 API library

API library is a linkable library which provides an interface to MDS and its message system. A simple call to InitMDS() registers itself with the Task Manager and opens up a communication channel with remote MDS Daemons. A call to CloseMDS() shutdowns the API server and notifies the Task Manager of its desire to terminate. Following is a list of functions offered by API library, a more detailed description can be found on the Reference Guide.

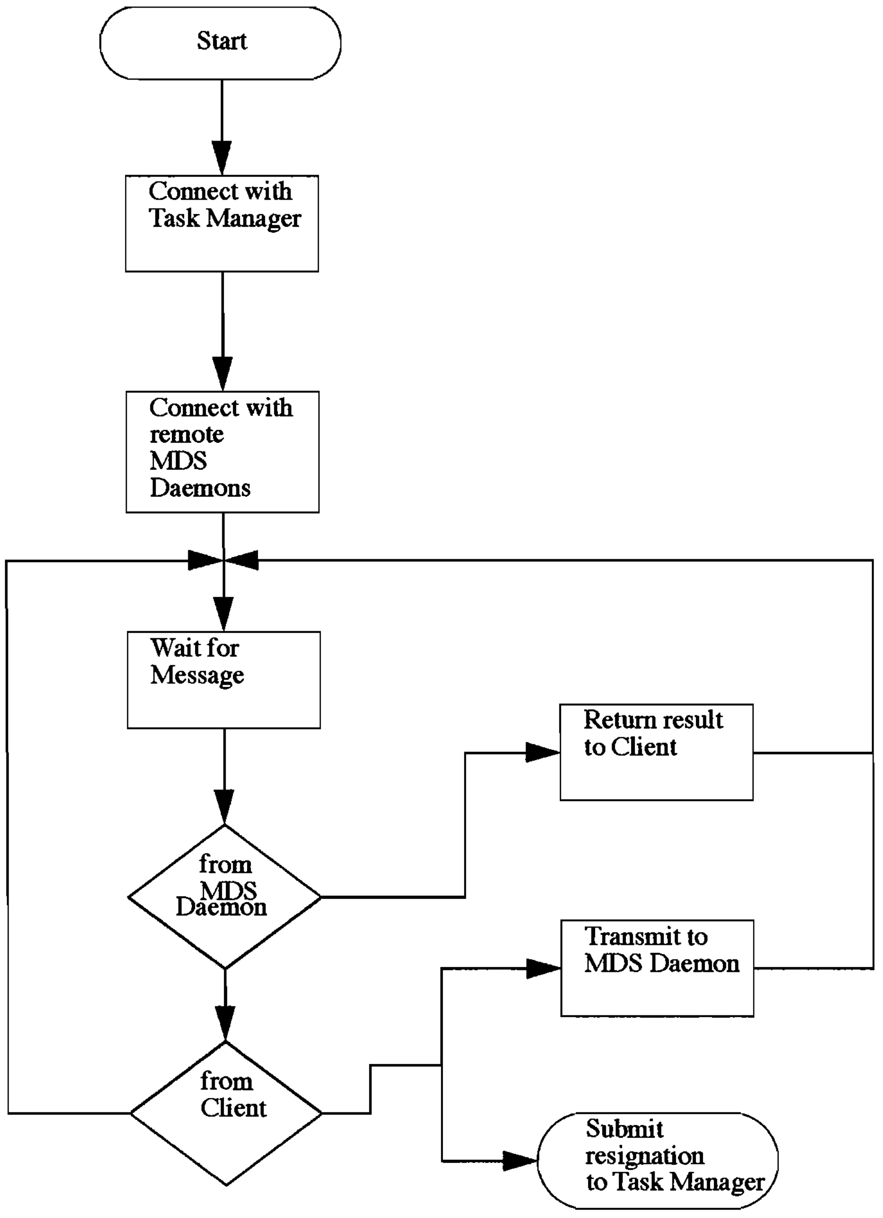

Figure 6. API Library

2.2.4.4 MDS Daemon

The MDS Daemon serves as a client daemon under the Internet Super Server (INet). It sets up a network port listening to connection requests from remote Resource Managers and API Libraries. Linkage among resouce managers are arranged by API Daemon. Remote tasks are carried out by the API Daemon by invoking the client's remote process and arranges message exchanges with the API Library. The following flowchart illustrates the operation of API daemon.

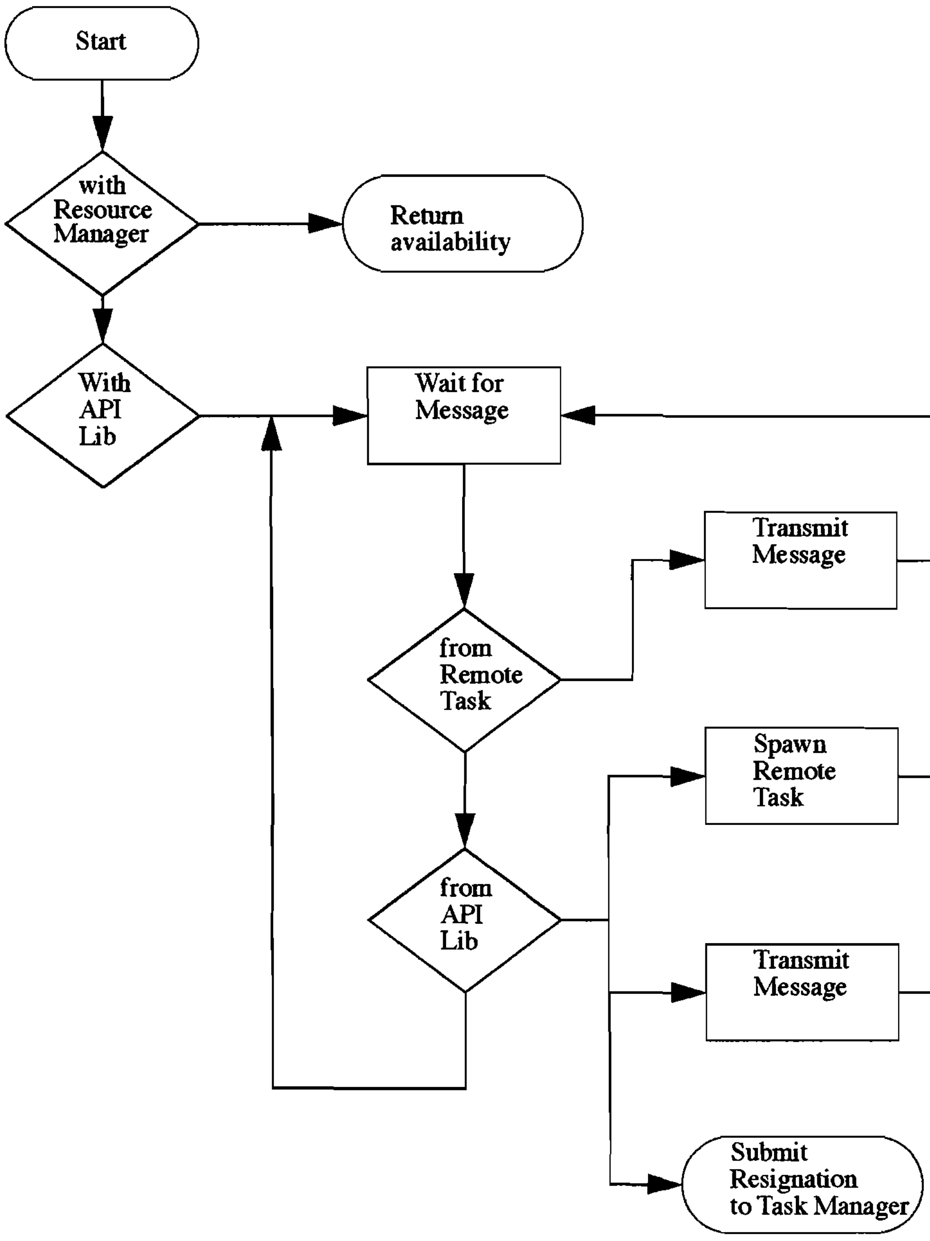

Figure 7. MDS Daemon

test1.c

```
#include "mds-api.h"
#include "client.h"

void main(int argc,char *argv[])
{
        int pid, servers;
        long lbegin, lend, ctrx, ctry, num, prim[5000];
        BOOL isprime;
        struct Special_Message *smsg;
        struct mMsgPort *cport;

        initdebug(DEBUG5);
        mds_output = Open("CON:10/320/520/80/Remote Client",MODE_OLDFILE);
        if (mds_output==NULL) return;
        mds_do_output=TRUE;
        if (!InitREMOTETASK(argc,argv)) {
                EWrite("Failed to initialize Remote Task\n");
                EWaitRETURN();
                Close(mds_output);
                return;
        }
        EWrite("Finding parent port\n");
        cport = mFindPort(-1);
        EWrite("Waiting for message\n");
        smsg = (struct Special_Message *)mGetMsg(cport);
        EWrite("Getting server ID\n");
        pid = GetServerID();
        servers = smsg->servers;
        lbegin = (smsg->total / servers) * pid + 1;
        lend = (smsg->total / servers) * (pid+1) - 1;
        num=0;
        for (ctrx=lbegin; ctrx<=lend; ctrx=ctrx+2) {
                isprime=TRUE;
                for (ctry=3;ctry < ((ctrx-1)/2);ctry=ctry+2)
                        if ((ctrx % ctry)==0) {
                                isprime=FALSE;
                                ctry=lend;
                        }
                if (isprime)
                        prim[num++]=ctrx;
        }
        smsg->total = num;
        mReplyMsg((struct mMessage *)smsg);
        for (ctrx=0; ctrx < num; ctrx++) {
```

References

1. Farshad Nayeri, Ben Hurwitz, and Frank Manola "Generalizing Dispatching in a Distributed Object System", Proc. of the 8th European Conference on Object-Oriented Programming, Springer-Verlag, 1994.
2. Jan Vitek, and R. Nigel Horspool "Taming Message Passing: Efficient Method Look-Up for Dynamically Typed Languages", Proc. of the 8th European Conference on Object-Oriented Programming, Springer-Verlag, 1994.
3. Peter C. Lockemann "Aggregate Behavior of Loosely Coupled Objects: Architectural Principle for Heterogeneous and Distributed Systems", Technical Report 0218-11-93-165, GTE Laboratories Incorporated, November 1993.
4. Gregory R. Andrews, and Fred B. Schneider "Concepts and Notations for Concurrent Programming", ACM 15, 1 (March 1983).
5. Eric Leu, Andre Schiper, and Abdelwahab Zramdini "Efficient Execution Replay Technique for Distributed Memory Architectures", Technical Report 20-5495.88, Ecole Polytechnique Federale de Lausanne, Departement d'Informatique, Switzerland.
6. Christian Kuhnert, Stefan Maelger, and Johannes Schemmel "Amiga Intern", Abacus.
7. Commodore-Amiga, Inc. "Amiga ROM Kernel Reference Manual: Libraries", Addison-Wesley Publishing Company, Inc.
8. "UMAX 4.3 Programming Guide", Encore Computer Corporation, 1989.
9. "UMAX 4.3 Programmer's Reference Manual", Encore Computer Corporation, 1989.

Mini-Distributed System (MDS) Reference Guide

Jasper Wong
Department of Computer Science
Brown University
Providence, RI 02912, USA

August 30, 1995

1 Purpose

Mini-Distributed System Reference Guide describes the content and use of each of the MDS functions available to the SAS/C and C++ programmer under AmigaDOS 3.1+.

2 Conventions

This section covers the typographical and syntax conventions used in this guide.

2.1 Typographical Conventions

The MDS reference guide use special fonts to depict specific types of information. These typographical conventions are as follows:

times	is the basic type style used for most text.
bold	is used to show example statements or programs. Bold is used also for items specific to the C and C++ languages, such as the names of functions, header files, and keywords.
italic	is used for terms that are defined and for arguments or variables whose values are supplied by the user. For example, you should enter an appropriate filename when you see filename.

2.2 Syntax Conventions

This guide uses the following conventions for syntax:

bold	indicates commands, keywords, and switches that should be spelled exactly as shown. These arguments may or may not be optional, depending on whether they are enclosed in square brackets.
italic	indicates arguments for which you supply a value.
[]	indicates an optional argument when they surround the argument.
...	indicates that you can repeat the argument or group of arguments preceding the ellipsis any number of times.
+	means to choose one item from a group of items separated by the pluses.

The following example illustrates these syntax conventions:
show [*function* + *integer*]

show
is a command name, so it appears in bold type.

function
is a function for which you supply the name, so it appears in italic type.

[*function* + *integer*]
are both optional, so they are enclosed in square brackets.

function + *integer*
indicates that you can specify only one of the items separated by the plus sigh.

3 Using the MDS Library

Current version of MDS library was compiled as an linkable object file mdsapi.o. If a program file named test.c is to be linked with the MDS library, you can compile, link, and run test.c with the following Shell commands:

```
sc LINK OBJ test.o mdsapi.o test.c
test
```

The header file mdsapi.h is needed for every program wishing to take advantage of MDS's distributed power.

init_MDS	Initialize MDS to be used with current application

Synopsis

```
#include “mdsapi.h”

d = initMDS(p, e, l);

int d;  /* number of available remote servers */
int p;  /* number of requested servers */
char *e;  /* Remote task to be executed */
BOOL l; /* boolean variable indicating to log the task or not */
```

Description

This function registers the current task with the Task Manager and awaits for connection with the API library. Its requet for the number of servers and the execution of remote task are then submitted to the API library.

Portability

SAS/C

Returns

The number of remote servers assigned is returned if the call is successful. A zero is returned if there’s no available remote server or the call failed.

Example

```
#include “mdsapi.h”

void main(void)
{
        if (init_MDS(10,”c:compute”,FALSE)) {
                /* Find 10 servers to run the task compute */
                ...
        }
        Close_MDS();
}
```

See Also

Close_MDS()

Close_MDS Shutdown MDS

Synopsis

```
#include “mdsapi.h”

Close_MDS();
```

Description

This function submits resignation with the Task Manager and notifies all remote MDS daemons to terminate. It then closes all the socket connections.

Portability

SAS/C

Returns

Nothing is returned.

Example

```
#include “mdsapi.h”

void main(void)
{
        if (init_MDS(10,”c:compute”,FALSE)) {
                /* Find 10 servers to run the task compute */
                ...
        }
        Close_MDS();
}
```

See Also

init_MDS()

CloseREMOTETASK Shutdown remote task

Synopsis

```
#include "mdsapi.h"

CloseREMOTETASK();
```

Description

This function submits resignation with the Task Manager and then sent for disconnection with its parent process.

Portability

SAS/C

Returns

Nothing is returned.

Example

```
#include "mdsapi.h"

void main(void)
{
        if (InitREMOTETASK(argc,argv)) {
                /* From here on it behaves as a normal program */
                ...
        }
        CloseREMOTETASK();
}
```

See Also

InitREMOTETASK()

InitREMOTETASK Initialize remote task to be used with parent process

Synopsis

```
#include "mdsapi.h"

d = InitREMOTETASK(argc, argv);

int d;  /* success or failure */
int argc;  /* number of arguments */
char *argv[];  /* actual arguments */
```

Description

This function registers the current task with the Task Manager and awaits for connection with its parent process. It then sets up all the necessary socket information.

Portability

SAS/C

Returns

1 if successful or 0 otherwise.

Example

```
#include "mdsapi.h"

void main(void)
{
        if (InitREMOTETASK(argc,argv)) {
                /* From here on it behaves as a normal program */
                ...
        }
        CloseREMOTETASK();
}
```

See Also

CloseREMOTETASK()

GetNumServers Checks to see how many servers were initialized by the parent process

Synopsis

```
#include “mdsapi.h”

d = GetNumServers();

int d;  /* number of servers */
```

Description This function can be used by the remote task to find out how many servers were used for task distribution.

Portability SAS/C

Returns It returns the number of servers used.

Example

```
#include “mdsapi.h”

void main(void)
{
        if (InitREMOTETASK(argc,argv)) {
                ...
                d = GetNumServers();
                ...
        }
        CloseREMOTETASK();
}
```

See Also None

GetServerID Check to see the current process ID number

Synopsis

```
#include “mdsapi.h”

d = GetServerID();

int d;  /* server’s ID */
```

Description This function returns the current server’s ID.

Portability SAS/C

Returns -1 if parent process or current server’s ID (positive integer).

Example

```
#include “mdsapi.h”

void main(void)
{
        if (InitREMOTETASK(argc,argv)) {
                ...
                d = GetServerID();
                ...
        }
        CloseREMOTETASK();
}
```

See Also None

mFindPort Find remote message port

Synopsis

```
#include "mdsapi.h"

d = mFindPort(p);

struct mMsgPort *d;  /* pointer to remote message port */
int p;  /* server's ID number */
```

Description

This function returns a pointer to a static message port associated with the server pointed to by the server's ID. -1 can be used to find parent process.

Portability

SAS/C

Returns

pointer to remote message port or NULL if not found.

Example

```
#include "mdsapi.h"

void main(void)
{
        struct mMsgPort *d;
        struct mMessage *msg;

        if (InitREMOTETASK(argc,argv)) {
                ...
                d = mFindPort(-1);
                if (d)
                        msg = mGetMsg(d);
                ...
        }
        CloseREMOTETASK();
}
```

See Also

mPutMsg(), mReplyMsg(), mGetMsg()

mPutMsg Put a message to the given message port

Synopsis

```
#include "mdsapi.h"

mPutMsg(d, m, len);

struct mMsgPort *d;  /* pointer to remote message port */
struct mMessage *m;  /* pointer to message needed to be sent */
long len; /* length of the message */
```

Description This function sends the message to the message port and disposes the current message.

Portability SAS/C

Returns None.

Example

```
#include "mdsapi.h"

void main(void)
{
        struct mMsgPort *d;
        struct mMessage *msg;

        if (init_MDS(4,"c:client",FALSE)) {
                ...
                d = mFindPort(2);
                if (d)
                        mPutMsg(d, msg, sizeof(struct mMessage));
                ...
                msg = mGetMsg(d);
                ...
        }
        Close_MDS();
}
```

See Also mFindPort(), mReplyMsg(), mGetMsg()

mGetMsg Get a message from the given message port

Synopsis

```
#include "mdsapi.h"

m = mGetMsg(d);

struct mMsgPort *d;  /* pointer to remote message port */
struct mMessage *m;  /* pointer to message needed to be sent */
```

Description This function waits for a message to arrive at the given message port and return a pointer to the message.

Portability SAS/C

Returns A pointer to a message or NULL if none.

Example

```
#include "mdsapi.h"

void main(void)
{
        struct mMsgPort *d;
        struct mMessage *msg;

        if (init_MDS(4,"c:client",FALSE)) {
                ...
                d = mFindPort(2);
                if (d)
                        mPutMsg(d, msg, sizeof(struct mMessage));
                ...
                msg = mGetMsg(d);
                ...
        }
        Close_MDS();
}
```

See Also mFindPort(), mReplyMsg(), mPutMsg()

mReplyMsg Reply to a received message

Synopsis

```
#include “mdsapi.h”

mReplyMsg(m);

struct mMessage *m;  /* pointer to a message */
```

Description

This function replies a message previously received with mGetMsg(). Message is then disposed.

Portability

SAS/C

Returns

None.

Example

```
#include “mdsapi.h”

void main(void)
{
        struct mMsgPort *d;
        struct mMessage *msg;

        if (InitREMOTETASK(argc,argv)) {
                ...
                d = mFindPort(-1);
                if (d)
                        msg = mGetMsg(d);
                ...
                mReplyMsg(msg);
        }
        CloseREMOTETASK();
}
```

See Also

mPutMsg(), mFindPort(), mGetMsg()