PROCESS MONITORING IN UNIX

Dissertation submitted to Jawaharlal Nehru University in partial fulfillment of the requirements for the award of the degree of

Master of Technology
in
Computer Science and Technology

By

SRUJAN DEEP DEVARA

SCHOOL OF COMPUTER & SYSTEMS SCIENCES
JAWAHARLAL NEHRU UNIVERSITY
NEW DELHI - 110067

JANUARY 2001

005.42 TH
D491 Pr

TH10246

जवाहरलाल नेहरू विश्वविद्यालय
JAWAHARLAL NEHRU UNIVERSITY
NEW DELHI-110067
SCHOOL OF COMPUTER & SYSTEMS SCIENCES

CERTIFICATE

This is to certify that the dissertation entitled **"PROCESS MONITORING IN UNIX"** submitted by ***Srujan Deep Devara*** to the School of Computer and Systems Sciences, Jawaharlal Nehru University, New Delhi in partial fulfillment of the requirements for the award of the degree of Master of Technology in Computer Science is a bonafide work carried out by him under the guidance and supervision of Asstt.Prof. R.C. Phoha.

The matter embodied in the dissertation has not been submitted for the award of any other degree or diploma.

Srujandeep Devara
SRUJAN DEEP DEVARA

5 Feb. 2001

Prof.C.P.Katti,
Dean, SC&SS,
Jawaharlal Nehru University,
New Delhi - 110067

Asstt.Prof. R. C. Phoha
SC&SS,
Jawaharlal Nehru University,
New Delhi - 110067

GRAM : JAYENU TEL. : 667676, 667557 TELEX : 031-73167 JNU IN FAX : 91-011-6865886

ACKNOWLEDGEMENTS

I would like to pay obeisance at the feet of my beloved parents for their blessings are always with me in all my aspirations including my academics.

I would like to sincerely thank my supervisor, Asst. Prof. R. C. PHOHA, School of Computer and Systems Sciences, Jawaharlal Nehru University for his help, encouragement and support extended in completion of this project.

I would like to record my sincere thanks to my dean Prof. C. P. Katti, Jawaharlal Nehru University for providing the necessary computing facilities.

I take this opportunity to thank all of my faculty members and friends for their help and suggestions during the course of my project work.

SRUJAN DEEP DEVARA

TABLE OF CONTENTS

1. INTRODUCTION 1

2. CONCEPTS OF PROCESS MONITORING 3

2.1 Process 3

2.2 Client-Server interaction 3

2.2.1 Client-Server model 4

2.2.2 Making contact 4

2.2.3 Direction of data flow 4

2.2.4 Transport protocol and client-server interaction 4

2.2.5 Identifying a particular service 5

2.2.6 Connection oriented and connectionless transport 6

2.2.7 UDP client-server 7

2.2.8 Overhead required for opening a TCP connection 9

2.3 Socket interface 10

2.3.1 Application program interface 10

2.3.2 Socket library 10

2.3.3 Socket communication and UNIX I/O 10

2.3.4 Procedures that implement the socket API 11

3. FUNCTIONAL SPECIFICATIONS

3.1 Process Monitoring 17

3.1.1 Overview 17

3.1.2 A high level view of the architecture 17

3.2 Functional requirements 19

3.2.1 Functional specification 19

4. HIGH LEVEL DESIGN 20

4.1 The monitor server 20

4.1.1 Design basic 20

4.1.2 The packets 21

4.1.3 Information maintained by the server 22
4.1.4 Processing done by the server 23
4.1.5 Use cases 24

4.2 The library design 24
4.2.1 The requirements from the library 24
4.2.2 Outline of the functions 25

5. LOW LEVEL DESIGN 26
5.1 The server 26
5.2 The watcher 29
5.3 The library 29
5.4 Implementation details 30
5.4.1 The packets 30
5.4.2 The timer setting 31
5.4.3 Getting the process name and machine name 31
5.5 File and functional details 32
5.5.1 The files 32
5.5.2 The functions and declaration 34
5.6 Pseudo code 35

6. Interface control
6.1 The server process interface 55
6.1.1 Introduction 55
6.1.2 Context diagram
6.1.3 Packet structures 56
6.2 Server watcher interface 57
6.3 Conclusion 57

7. Test plan 58
7.1 Sever testing 58
7.1.1 Maintenance of the process structure 59

7.1.2 Maintenance of the time list structure 59
7.1.3 Resetting of the timer 60

7.2 Library testing 60
7.2.1 Signal testing 60
7.2.2 Checking the end monitoring packet 61
7.3 Testing the watcher 61
7.4 Testing the interfaces 61

CONCLUSION 62

REFERENCES 63

ABSTRACT

The dissertation deals with Process monitoring in Unix networking environment. UNIX is a networking operating system tightly integrated with TCP/IP protocols. The status of the processes on a Unix machine can be known by process monitoring commands (ps, time). But these utilities will give the status of the processes only on the current machine. In larger networks there are number of processes running on different machines at a time and these processes are critical and need to be monitored. Process monitoring can be defined as the procedure of taking care of the state of the different processes over a network individually, as per the user and the process requirements and specifications. This is an essential component of network monitoring and so network health management system, as monitoring of the processes helps to gain more knowledge about the running of the processes and hence the network. By getting the statistical distribution of the behavior of the processes it can help efficient utilization of the resources of the network. The design is based on Client - Server architecture and the client and server applications use TCP/IP protocol suite to communicate over the network.

CHAPTER 1
INTRODUCTION

The "information super highway " has received a lot of attention recently. Much of this "network of the future" is with us today. The Globe is shrinking and coming closer day by day as the networks kept expanding for example internet which is a collection of networks allows to connect ones computer to hundreds and thousands of computers world wide and thus has redefined communication. In such a scenario network health management is of paramount importance which is based on network monitoring.

The main purpose of the monitoring is for efficient utilization of the resources, for traffic management and to check unauthorized access attempts i.e. intrusion detection etc. Network monitoring in it's various forms involves monitoring at process level, traffic monitoring, resource monitoring and intrusion monitoring etc. In this process monitoring plays a very important role, as it is useful for efficient utilization of CPU i.e. to know how to make large, but not time- critical, tasks takes less of CPU time, to learn how to shutdown programs that have gone astray which helps in efficient use of resources, and to learn how to improve the performance of the machine.

Basically it is the UNIX software that connect hundreds of thousands of machines together in the INTERNET and USENET as Unix is a network operating system tightly integrated with TCP/IP networking protocols . We use processes on Unix every time we want to get some thing done. Each command (that isn't built into shell) that is run will run one or more new processes to perform the desired task. To get the most benefit out of Unix machine we need to monitor the processes that are running on it.

Unix is a multi-user, multitasking i.e. time sharing operating system. Many processes can be activated at any given time (the goal of time sharing system). Processes

have to share system resources (CPU, memory, and so on). In this CPU is switched rapidly between the processes to give an illusion of multiprocessing system and hence processes have to be monitored for efficient utilisation of system resources. The first step in controlling processes is to learn how to monitor them. By using the process monitoring commands in Unix , we will be able to find what programs are using CPU time , find jobs that are not completing , and generally explore what is happening to the machine.

Any process at a given time can be in different states like sleeping, waiting, running etc. This information can be obtained using the process monitoring commands (ps, time, which) .But this gives at a time only the information about the processes on the current machine. When processes over a network need to be monitored simultaneously ,then we need to use better methods.

In a larger network there are number of processes running on different machines at a time. A lot of these processes are critical and they need to be monitored. By monitoring it is meant that the state of the process at any point of time like running , sleeping , waiting, etc. needs to be known. Sometimes there are long processes which have critical sections and need to be monitored whenever those sections are running. The project " Process Monitoring in Unix " addresses these requirements.

The dissertation is divided into seven chapters: the first chapter gives a brief introduction to process monitoring. Chapter two explains the conceptual basis which includes an introduction to the concept of Process, Client- Server architecture and Socket interface. Third chapter is the functional specifications defined for process monitoring. Fourth chapter describes the high level design of the architecture in line with the functional requirements. Fifth chapter describes the low-level design of the library and the server in line with the high level design. Sixth chapter deals with three different entities, which interface with each other i.e., the server-process interface and the server-watcher interface. Finally the seventh chapter describes the test plan for the process monitoring software developed.

Chapter 2
CONCEPTS OF PROCESS MONITORING

2.1 Process

A fundamental entity in a computer network is a process. A process is a program that is being executed by the computer's operating system. When we say that two computers are communicating with each other, we mean that two processes, one running on each computer, are in communication with each other.

2.2 Client - Server Interaction

The primary pattern of interaction among co-operating applications is known as the Client-Server paradigm. Client server interaction forms the basis of most network communication.

2.2.1 CLIENT -SERVER MODEL

The term server applies to any program that offers a service that can be reached over a network. A server accepts a request over the network, performs its service, and returns the result to the requester. An executing program becomes a client when it sends a request to a server and waits for a response. Usually servers are implemented as application programs. The advantage of implementing servers as application programs is that they can execute on any computing system that supports TCP/IP communication. Thus, the server for a particular service can execute on a timesharing system along with other programs, or it can execute on a personal computer. Multiple servers can offer the same service, and can execute on the same machine or on multiple machines.

Two important points that are generally true about Client-Server interaction.

The first concerns the difference between the lifetime of severs clients:

A server starts execution before interaction begins and continues to accept requests and sends responses without ever terminating. A client is any program that makes a request and awaits a response; it (usually) terminates after using a server a finite number of times.

The second point, which is more technical, concerns the use of reserved and non-reserved port identifiers

A server waits for requests at a well-known port that has been reserved for the service it offers. A client allocates an arbitrary, unused, non-reserved port for its communication.

2.2.2 Making Contact

Instead of waiting for an arbitrary message to arrive, an application that expects communication must interact with protocol software before an external source attempts to communicate. The application informs the local protocol software that a specific message is expected, and then the application waits. When an incoming message matches exactly what the application has specified , protocol software passes the message to the application.

2.2.3 Direction of data flow

Information can pass in either or both direction between a client and a server .Typically, a client sends a request to a server , and the server returns a response to the client. In some cases a clients sends a series of requests and the server issues a series of responses (e.g., a database client might allow a user to look up more than one item at a time). In other cases, the server provides continuous output without any request as soon as the client contacts the server, the server begins sending data.

2.2.4 Transport protocols and client -server interaction

A client and server use a transport protocol to communicate. For example, fig below illustrates a client and server using the TCP\IP stack

As the figure shows, a client or server application interacts directly with a transport-layer protocol to establish communication and to send or receive information. The transport protocol then uses lower layer protocols to send and receive individual messages.

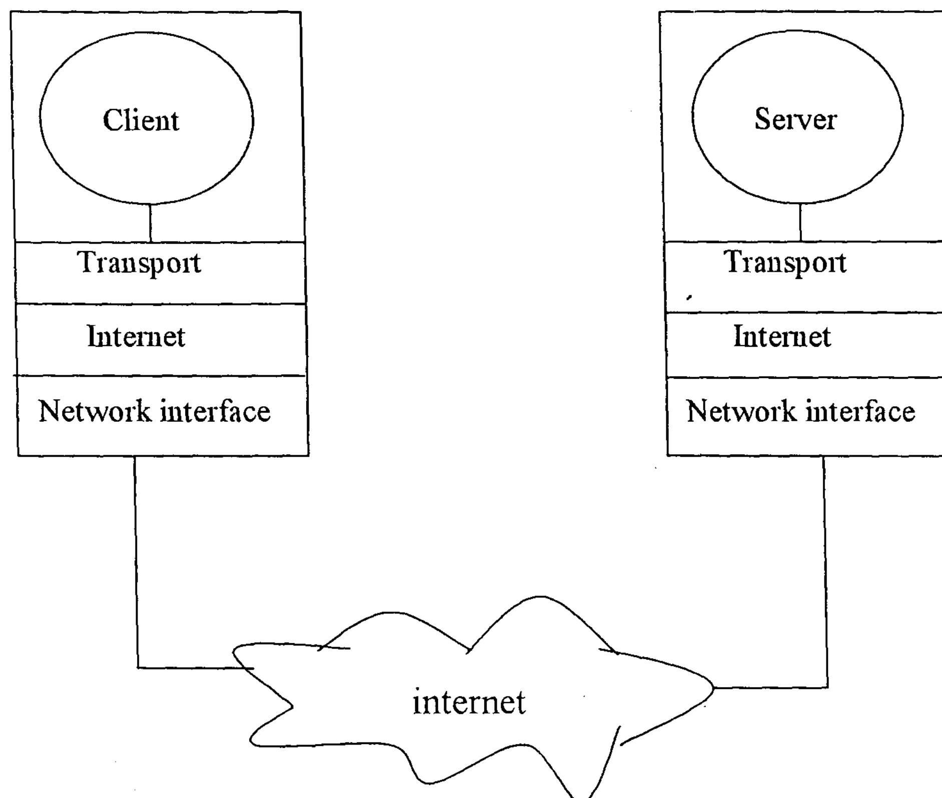


Fig A client and server using TCP/IP protocols to communicate across an internet.

2.2.5 Identifying a particular service

Transport protocols provide a mechanism that allows a client to specify unambiguously which service is desired. The mechanism assigns each service a unique identifier , and requires both the client and the server to use the identifier. When a server begins execution, it registers with local protocol software by specifying the identifier for the

service it offers. When a client contacts a remote server , the client specifies the identifier for the desired service. Transport protocol software on the client's machine sends the identifier to the server 's machine when making a request. Transport protocol software on the server's machine uses the identifier to determine which server program should handle the request.

As an example of service identification ,TCP uses 16-bit integer values known as protocol port numbers to identify services, and assigns a unique protocol port number to each service. A server specifies the protocol port number for the service it offers , and then waits passively for communication. A client specifies the protocol port number of the desired service when sending a request.

2.2.6 Connection - Oriented and Connectionless Transport

Transport protocols support two basic forms of communication: connection-oriented or connectionless. To use a connection-oriented transport protocol, two applications must establish a connection, and then send data across the connection. For example, TCP provided a connection-oriented interface to application. When it uses TCP, an application must first request TCP to open a connection to another application. Once the connection is in place, the two applications can exchange data. When the applications finish communicating, the connection must be closed.

The alternative to connection-oriented communication is a connectionless interface that permits an application to send a message to any destination at any time. When using a connectionless transport protocol, the sending application must specify a destination with each message it sends. For example, in the TCP/IP protocol suite, the User Datagram Protocol (UDP) provides connectionless transport. An application using UDP can send a sequence of messages, where each message is send to a different destination.

Clients and servers can use either connection-oriented or connectionless transport protocols to communicate. When using a connection-oriented transport, a client first

forms a connection to a specific server. The connection then stays in place while the client sends requests and receives responses. When it finishes using the service, the client closes the connection.

Clients and servers that use connectionless protocols exchange individual messages. For example, many services that use connectionless transport require a client to send each request in a single message, and the server to return each response in a single message.

2.2.7 UDP Client - Server

The client sends a request to the server, the server processes the request and sends back a reply.

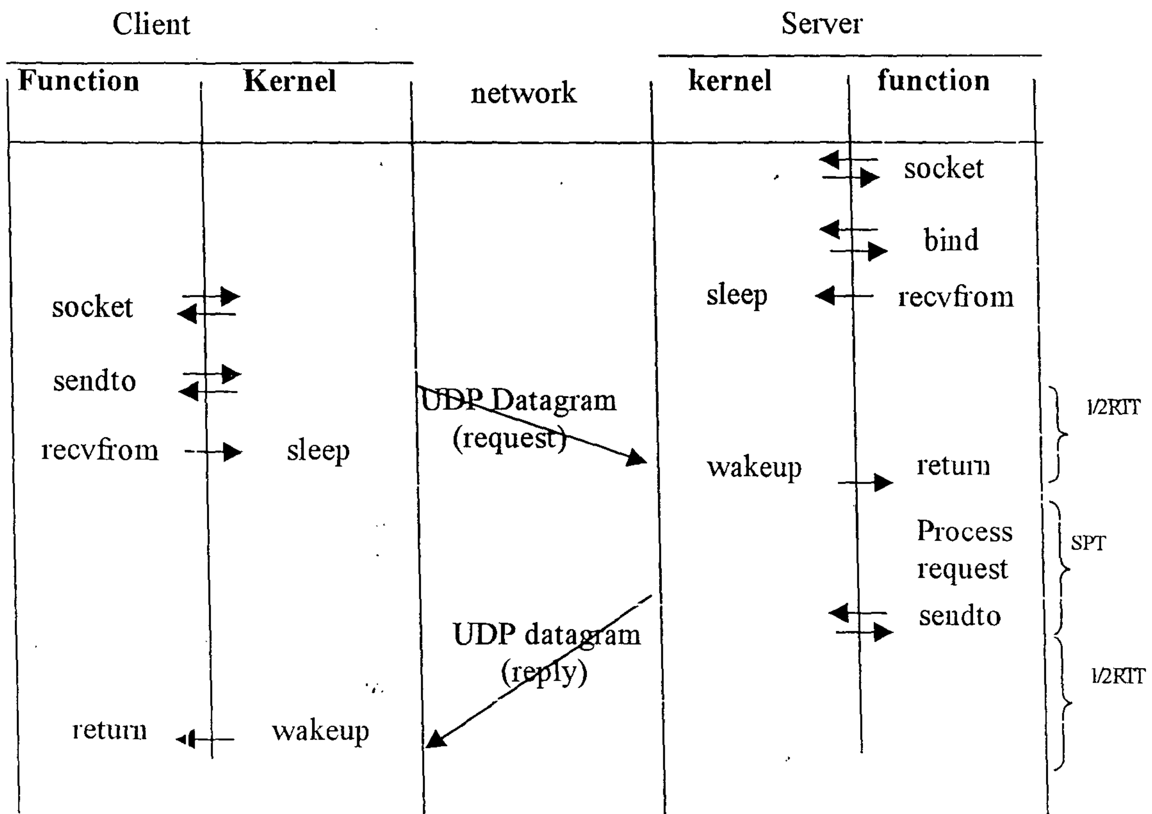


Fig Time line of UDP client - server transaction

If we watch the packets that are exchanged when a client sents the server a request, we have the time line shown in fig above. Time increases down the page. The server is started first, shown in right side of the diagram, and the client is started sometime later.

We distinguish between the function call performed by the client and server, and the action performed by the corresponding kernel. We use two closely spaced arrows, as in the two calls to socket, to show that the kernel performs the requested action and returns immediately. In the call to *sendto,* although the kernel returns immediately to the calling process, a UDP datagram is sent. For simplicity we assume that the sizes of the resulting IP datagrams generated by the client's request and the server's reply are both less than the network's MTU (maximum transmission unit), avoiding fragmentation of the IP datagram.
In this figure we also show that the two calls to *recvfrorm* put the process to sleep until a datagram arrives. We denote the kernel routines as sleep and wakeup.

Finally, we show the times associated with the transaction. On the left side of fig above we show the transaction time as measured by the client: the time to send a request to the server and receive a reply. The values that comprise this transaction time are shown on the right side of the figure: RTT + SPT, where RTT is the network roundtrip time, and STP is the server processing time for the request. The transaction time for the UDP client-server, RTT +SPT, is the minimum possible.

Since UDP is an unreliable protocol, datagrams can be lost, reordered, or duplicated, and a real application needs to handle these problems. This normally involves retransmitting the request. If a timeout is going to be used, the client must measure the RTT and update it dynamically, since RTTs on an internet can vary widely and change dramatically over time. But if the reply was lost, instead of the request, the server will process the same request a second time, which can lead to problems for some types of services. One way to handle this is for the server to save the reply for each client's latest request, and retransmit that reply instead of processing the request another time. Finally, the client typically

sends an identifier with each request, and the server echoes this identifier, allowing the client to match responses with requests.

While many UDP applications add reliability by performing all of these additional steps (timeouts, RTT measurements, request identifier, etc.), these steps are continually being reinvented as new UDP applications are developed.

2.2.8 Overhead required for opening a TCP connection

First we need to go through several steps of opening a connection. This takes quite a bit of time. once the connection is open, sending and receiving data each involve several steps. Each of these steps adds some time and data overhead to the transaction. If we are sending large amounts of data that must absolutely arrive at its destination, we use the TCP protocol. However, if all we want to do is quickly send a simple, short message, all of this work may not be worthwhile.

Over an IP network such as the internet. a protocol called UDP(the unreliable Datagram Protocol)is used to transmit fixed-length datagrams.

Datagrams have a couple of Advantages.

Speed :

UDP involves low overhead. With TCP you want to go through the hassle of setting up and tearing down a connection which takes time. For small amounts of data, it may not be worth it. the overhead time for setting up a connection may be greater than the amount of time it takes to send a small chunk of data. In many cases we could send and retry to s end a datagram several times before a TCP connection could be opened

Message Oriented Instead Of Stream Oriented

If we have a simple data structure such as a database record with fixed length fields, it might be easier to simply send the chunk of bytes.

2.3 THE SOCKET INTERFACE

The chapter considers how an application uses protocol software to communicate, and explains an example set of procedures that an application uses to become a client or a server, to contact a remote destination, or to transfer data.

2.3.1 Application Program Interface

Client and server applications use transport protocols to communicate. When it interacts with protocol software, an application must specify details such as whether it is a server or a client (i.e., whether it will wait passively or actively initiate communication). In addition, applications that communicate must specify further details (e.g., the sender must specify the data to be sent, and the receiver must specify where incoming data should be placed). The interface an application uses when it interacts with transport protocol software is known as an Application Program Interface (API). An API defines a set of operations that an application can perform when it interacts with protocol software. Thus, the API determines the functionality that is available to an application as well as the difficulty of creating a program to use that functionality. Usually, an API contains a separate procedure for each basic operation. For example, an API might contain one procedure that is used to establish that is used to establish communication and another procedure that is used to send data.

2.3.2 Socket Library

A socket library can provide applications with a socket API on a computer system that does not provide native sockets. When an application calls one of the socket procedures, control passes to a library routine that makes one or more calls to the underlying operating system to implement the socket function.

2.3.3 Socket communication and UNIX I/O

Sockets are integrated with I/O - an application communicates through a socket similar to the way the application transfers data to or from a file. Thus, understanding sockets requires one to understand UNIX I/O facilities. UNIX uses an open-read-write - close paradigm for all I/O; the name is derived from the basic I/O operations that apply to

both devices and files. For example, an application must first call open to prepare a file for access. The application then calls read or write to retrieve data from the file or store data in the file. Finally, the application calls close to specify that it has finished using the file. When an application opens a file or device, the call to open returns a descriptor, a small integer that identifies the file; the application must specify the descriptor when requesting data transfer (i.e., the descriptor is an argument to the read or write procedure). Socket communication also uses descriptor approach. The system returns a small integer descriptor that identifies the socket. The application then passes the descriptor as an argument when it calls procedures to transfer data across the network; the application does not need to specify details about the remote destination each time it transfers data.

2.3.4 Procedures that Implement The Socket API

The socket procedure creates a socket and returns an integer descriptor:

Descriptor = socket (protofamily, type, protocol)

Argument protofamily specifies the protocol family to be used with the socket. For example, the value PF_INET is used to specify the TCP/IP protocol suite. Argument type specifies the type of communication the socket will use. The two most common types are a connection-oriented stream transfer (specified with the value SOCK_STREAM) and a connectionless message-oriented transfer (specified with the value SOCK_DGRAM). Argument protocol specifies a particular transport protocol used with the socket. Having a protocol argument in addition to a type argument, permits a single protocol suite to include two or more protocols that provide the same service.

The Close Procedure

The close procedure tells the system to terminate use of a socket. It has the form:

Close(socket)

Where socket is the descriptor for a socket being closed. If the socket is using a connection-oriented transport protocol, close terminates the connection before closing the socket. Closing a socket immediately terminates use - the descriptor is released, presenting the application from sending more data, and the transport protocol stops accepting incoming messages directed to the socket, preventing the application from receiving more data.

The Bind Procedure

When created, a *socket* has neither a local address nor a remote address. A server uses the bind procedure to supply a protocol port number at which the server will wait for contact. Bind takes three arguments:

Bind(socket, localaddr, addrlen)

Argument socket is the descriptor of a socket that has been created but not previously bound; the call is a request that the socket be assigned a particular protocol port number. Argument *localaddr* is a structure that specifies the local address to be assigned to the socket, and argument *addrlen* is an integer that specifies the length of the address.

The Listen Procedure

After specifying a protocol port, a server must instruct the operating system to place a socket in passive mode so it can be used to wait for contact from clients. To do so, a server calls the listen procedure, which takes two arguments:

Listen(socket, queuesize)

Argument socket is the descriptor of a socket that has been created and bound to a local address, and argument *queuesize* specifies a length for the socket's request queue. The operating system builds a separate request queue for each socket. Initially, the queue is empty. As requests arrive from clients, each is placed in the queue; when the server asks to retrieve an incoming request from the socket, the system returns the next request from the queue. If the queue is full when a request arrives, the system rejects the request. Having a queue of requests allows the system to hold new requests that arrive while the server is busy handling a previous request.

The Accept Procedure

All servers begin by calling socket to create a socket and bind to specify a protocol port number. After executing the two calls, a server that uses a connectionless transport protocol is ready to accept messages. However, a server that uses a connection-oriented transport protocol requires additional steps before it can receive messages: the server must call listen to place the socket in passive mode, and must then accept a connection request. Once a connection has been accepted, the server can use the connection to communicate with a client.

The Connect Procedure

Clients use procedure connect to establish connection with a specific server. The form is:

Connect(socket, saddress, saddresslen)

Argument socket is the descriptor of a socket on the client's computer to use for the connection. Argument *saddress* is a sockaddr structure that specifies the server's address and protocol port number. Argument *saddresslen* specifies the length of the server's address measured in octets. The connect procedure, which is called by clients, has two uses. When used with connection-oriented transport, connect establishes a transport connection to a specified server. When used with connectionless transport, connect records the server's address in the socket, allowing the client to send many messages to the same server without requiring the client to specify the destination address with each message.

The Send, Sendto, And Sendmsg Procedures

Both clients and servers need to send information. Usually, a client sends a request, and a server sends a response. If the socket is connected, procedure send can be used to transfer data. Send has four arguments:

Send(socket, data, length, flags)

Argument *socket* is the descriptor of a socket to use, argument data is the address in memory of the data to send, argument length is an integer that specifies the number of octets of data, and argument flags contains bits that request special options

Procedures *sendto* and *sendmsg* allow a client or server to send a message using an unconnected socket; both require the called to specify a destination. *Sendto*, takes the destination address as an argument. It has the form:

Sendto(socket, data, length, flags, destaddress, addresslen)

The first four arguments correspond to the four arguments of the send procedure. The final two arguments specify the address of a destination and the length of that address. The *sendmsg* procedure performs the same operation as *sendto,* but abbreviates the arguments by defining a structure. The shorter argument list can make programs that use *sendmsg* easier to read:

Sendmsg(socket, msgstruct, flags)

Argument msgstruct is a structure that contains information about the destination address, the length of the addresss, the message to be sent, and the length of the message.

The Recv, Recvfrom, and Recvmsg Procedures

A client and a server each need to receive data sent by the other. The socket API provides several procedures that can be used. For example, an application can call *recv* to receive data from a connected socket. The procedure has the form:

Recv(socket, buffer, length, flags)

Argument socket is the descriptor of a socket from which data is to be received. Argument buffer specifies the address in memory in which the incoming message should be placed, and argument length specifies the size of the buffer. Finally, argument flags allows the caller to control details (e.g., to allow an application to extract a copy of an incoming message without removing the message from the socket). If a socket is not connected it can be used to receive messages from an arbitrary set of clients. In such cases the system returns the address of the sender along with each incoming message. Applications use procedure *recvfrom* to receive both a message and the address of the sender:

Recvfrom(socket, buffer, length, flags, sndaddr, saddrlen)

The first four arguments correspond to the arguments of recv; the two additional arguments, *sndraddr* and *saddrlen*, are used to record the sender's IP address. Argument *sndaddr* is a pointer to a *sockaddr* structure into which the system writes the sender's

address, and argument *saddrlen* is a pointer to an integer that the system uses to record the length of the address. *Recvfrom* records the sender's address in exactly the same form that *sendto* expects. Procedure *recvmsg* operates like *recvfrom* , but requires fewer argument. It has the form:

Recvmsg(socket, msgstruct, flags)

Where argument *msgstruct* gives the address of a structure that holds the address for an incoming message as well as locations for the sender's IP address.

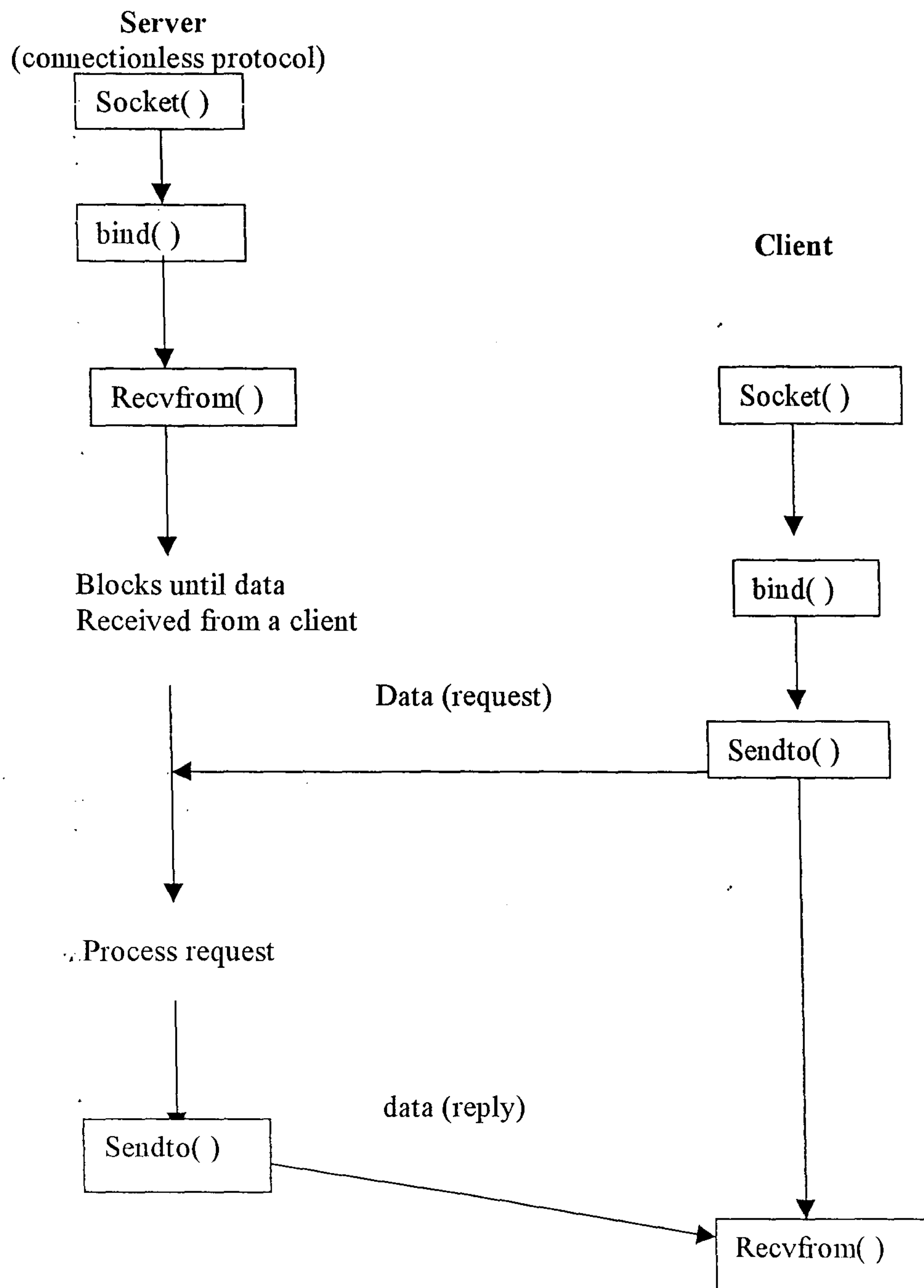


Fig Socket system calls for connectionless protocol

For a client server using a connection less protocol the client does not establish a connection a with a server . Instead , the client just sends a datagram to the server using the sendto system call , which requires the address of the destination(the server) as a parameter . similarly , the server does not have to accept a connection from a client . Instead , the server just issues a *recvfrom* system call that waits until data arrives from some client. The *recvfrom* returns the network address of the client process along with the datagram , so the server can send its response to the correct process.

CHAPTER 3
FUNCTIONAL SPECIFICATIONS

3.1 Process Monitoring

3.1.1 Overview

In a large network there are many processes running on different systems at a time. Any process at a given time can be in any of the many different states like sleeping, waiting, running etc. This information can be obtained using the ps utility of Unix. But this gives at a time only the information about the processes on the current machine. When processes over a network need to be monitored simultaneously then we need to use better methods. Thus for effective management over a network and to take care of the various critical systems we design this process monitoring software.

Thus we can define process monitoring as " the procedure of taking care of the state of the different processes over a network individually as per the user and the process requirements and specifications".

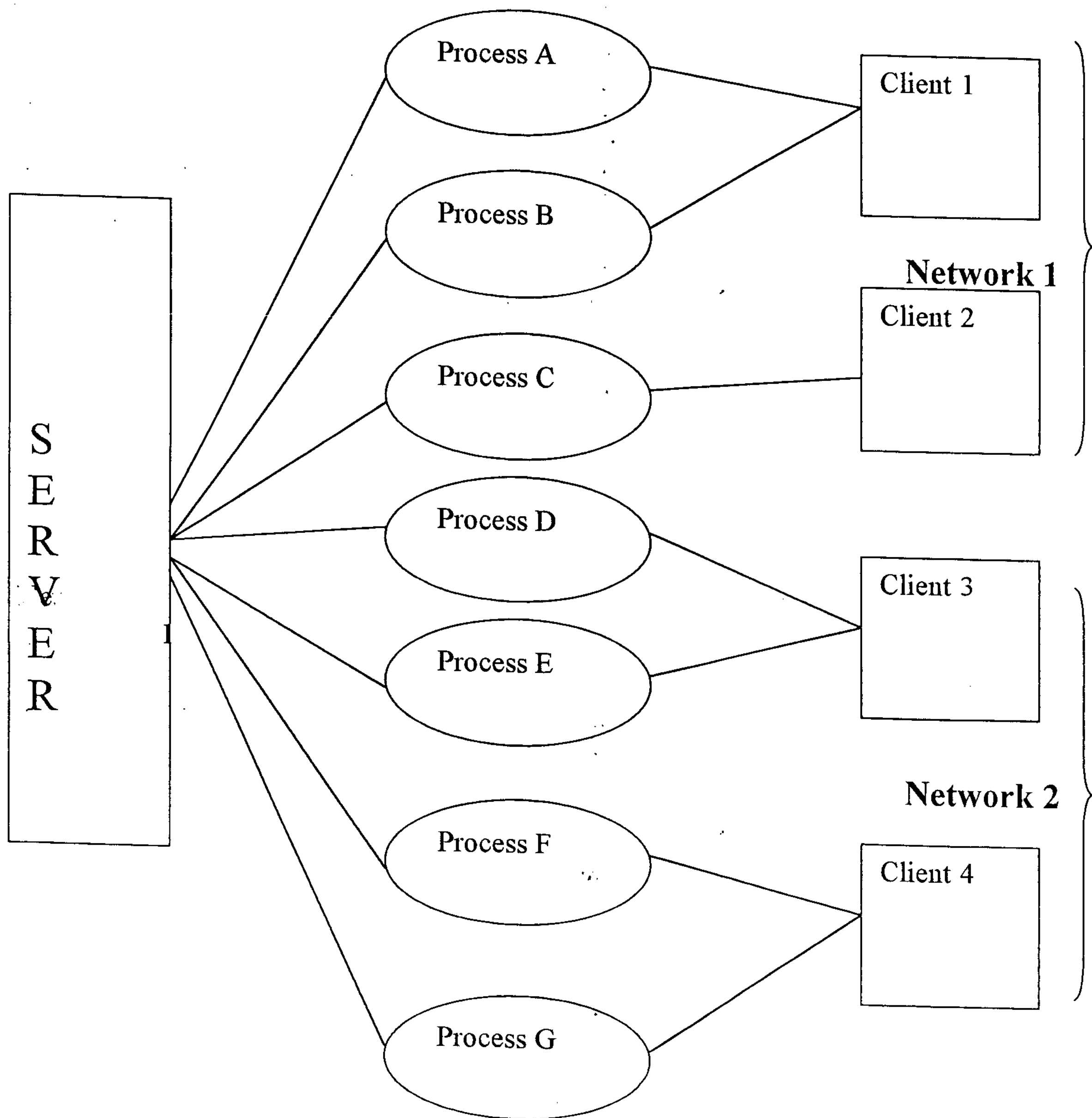
The keywords in the above definition are "individually" and "user and process requirements and specifications."

3.1.2 A High Level View of the Architecture

The architecture consists of a monitoring server which runs at a specific place over the network. The processes that want to be monitored intimate the same to this process monitor server. This intimation consists of other information about the process like the process ID and the time slice etc. Now the monitor server maintains the list of processes

being monitored and expects a signal by each of these processes at the specified times(indicated by the time slice for each process and the starting time) failing to receive which it changes the process status from UP to DOWN in the maintained table.

Basic Architecture



3.2 Functional Requirements

3.2.1 Functional Specifications

- A library which can be used by the application programs to tell the process monitor server that it wants to get monitored and also about its state. The states can be *RUNNING / STOPPED.*

- The library updates the process monitor server about its state using a custom protocol after a fixed interval of time.

- The different processes should be differentiated on the basis of some unique name. Each process has a unique identification number, but two processes on different machines can have same identification number, therefore to differentiate each process machine number is also taken into account.

- The process monitor server should also be monitored. For this to be carried out a monitor server watcher process is also run, which keeps track of the monitor server.

- At any point of time the monitor server could also go down, therefore the current status of the server should also be stored in text file.

- The process to be monitored must be defined in some configuration file.

CHAPTER 4

HIGH LEVEL DESIGN

The Design

Introduction

The software can be basically divided into two parts :-

1. The Process Monitor Server Design
2. The Library Routine Design(the functions used by processes to communicate with the server)

4.1 The Monitor Server

4.1.1 Design Basic

This is the server which runs on a fixed machine and actually monitors all the processes. It maintains the details of all the processes that are being monitored or that have been monitored. It is this server which receives packets from the various processes and handles them. The various processes connect to this server through the provided Library Routines. The server is assumed to be running by the processes which are using it to get themselves monitored.

The server needs information about the processes it needs to monitor. This information is provided to the server by two methods :

- A configuration file that is read by the server whenever it is switched on. This file contains the list of the processes with their machine names which it is supposed to monitor.
- The processes which want to get themselves monitored send a UDP packet to the server having necessary information.

The server maintains the information of all the processes it is monitoring or has monitored and constantly updates it depending on

- the various packets it receives
- packets which it expects but does not receive.

Thus the server needs to maintain information of two kinds – one about the processes which are being *currently* monitored and the other about all the processes. This is maintained in a list data structure by the server.

The server also needs to keep information in a file which is read by the user. This file is called the process file and is updated by the server from time to time. The information in this file is not as exhaustive as that maintained in the structure as it is used only to know the state of the processes while the structure also keeps data for monitoring.

TH-10246

4.1.2 The Packets

There are three types of packets. Whenever the server receives any of these packets it updates the structure and the files according to type of packet and the information received in that packet.

- Information Packet : This is the packet which is sent by the a process which wants itself tobe monitored. The server creates an entry of that process in its structure and the file it maintains and starts monitoring it also. This packet contains information like the process name, machine name, the time slice of the process, the ending criteria of the process and the process state.
- Message Packet : This is the packet which the processes send from time to time informing the server about their state. This is the packet which the server expects from the processes time to time depending on the time slice set by the process. The server updates information at various places according to the information received.This packet contains the process name, machine name and the proces state.
- End Monitoring : This packet is sent by a process when it wants the server to stop monitoring it. The server again updates the information. This packet contain the process name, machine name and the final process state.

4.1.3 Information Maintained by the Server

As stated in the basic design the server maintains information in the structure and the process file. Following are the details of what are the various fields maintained and when are they updated.

The Structure

The fields which data structure conatins alongwith when they are updated is given below

- The Process Name

  This field is created whenever information packet is received or when the configuration file is read. It is not changed thereafter.

- The Machine Name

  This field is created along with the proces name and is not changed thereafter.

- The Time Slice

  This field is updated with every message packet received.

- The Time to Receive Next Packet

  This is updated from time to time whenever a packet is received and is set to the current time plus the time slice. This is also updated whenever there is a timeout and packet is missed and more misses are allowed.

- The State of the Process

  This is updated whenever packet is received as every packet contains the state of the process.

- Number of Packet Misses still allowed

  This is updated whenever there is a packet miss or whenever a packet is received setting it to the ending criteria.

- Ending Criteria for the Process

  This is received in the information packet and not changed hence forth.

The File Maintained

The file contains the following fields for every process :

- The Process Name
- The Machine Name

- The Time Slice
- The Time last packet received
- The State of the Process

Updating in the file is done whenever there is any change in any of these fields.

4.1.4 Processing done by the Server

Except from maintaining all this information in the structures and files and receiving the UDP packets the server does a lot of processing which not only includes parsing these packets but also keeping track of any packet misses to update the information it maintains. For keeping track of any packet misses the server sorts the structure it maintains with respect to the time it should expect the next packet from a process. This helps the server keep track of when to assume that a packet has been missed. Thus the server sets a timer which times out at the required time and signals the server to make the necessary updations.

Many issues arise here. Following are the requirements or issues and their solution from this timer which is being used by the server for detecting packet misses.

- This timer needs to time out at the earliest expected packet.

  The structure maintained by the server is kept sorted according to the time when the next packet is expected and the timer is always set to the value for the header of the list.

- The timer needs to be reset whenever there is new process to be monitored with small time slice.

  Whenever a new process is added to the list the timer is reset to the header time.

- The timer needs to be reset whenever the packet is received in due time. The new time at which the timer is to be set needs to be known.

  Whenever a packet is received the next expected time for that process changes and so does its position in the list. The resetting of the timer to the time value of the header again solves the problem.

- As packets can be received at unexpected times the timer setting depends on the actual time rather than the elapsed time.

  This is handled in the same way as the last case.

Thus the timer efficiently takes care of any packet misses and the server is able to keep the various information updated.

4.1.5 Use Cases

Following is a listing of the various situatons that may arise while running of the server and how are they being handled.

Use Case I

Pre-Condition : Since the process file contains only the process recognition information how does the server start monitoring it.

Handling : The server does not assume these processes to be up unless it receives a packet from them. These processes are till then kept in a default state. The server assumes some default values for the different fields.

Use Case II

Pre-Condition : The server while running crashes due to some unforseen circumstances.

Handling : To handle such a case there is a *monitor watcher* which which is a process monitoring the server. Whenever the server crashes the watcher dumps all the previous information and exits.

Sub Case

Pre-Condition : The server crashes and the data that was being maintained by the server in its structure is lost.

Handling : The structure was being maintained by the server for efficient processing. The relevant data is being constantly updated in the process file. Thus that structure is anyway not needed once the server crashes.

4.2 The Library Design

The following sections give a basic design of the Library and the functions in it.

4.2.1 The Requirements from the Library

The Library will have functions that should be able to do the following :

- When the process wants itself to be monitored then it sends a check packet to the monitor server. This packet just contains the process recognition information.
- The process sends its monitoring information to the server through an information packet. This packet contains information like the time slice etc.
- The process needs to send a message packet to the server every time after a time slice has lapsed. This is done by setting a timer which times out to send a signal to the process to send a message packet to the server.
- The process sends a end packet when it wants the server to end monitoring.

4.2.2 Outline of the Functions

Thus library will contain the following functions which will perform the following actions:

- CONNECTION

  It will open a socket and bind a address to it.
- INFORMATION_PACKET

  This function will prepare a information packet about the process (process name, machine name, time slice, etc.) and will send this packet to the server.
- MESSAGE_PACKET

  This function will prepare a message packet like the information packet and send it to the server from time to time on receiving a signal from the kernel.
- ALARM_SIGNAL

  This function will set an alarm clock for the number of seconds specified by the process after that much seconds it will receive a signal from the kernel and call the message packet function.
- END_MONITOR_PACKET

  If the process in between does not want itself to get monitored then it will send a message to the server to ends monitoring.

CHAPTER 5

LOW LEVEL DESIGN

The Low Level Description

5.1 The Server

Introduction

First an outline of the control flow of the server is given. This is followed by the details of each step through the different struuctures used and processing done. Here all the various functions which the server performs have been finally integrated and a complete design has been given.

Control Flow

The server basically does the following three things in order :

1. Open a socket and binds it to a universally identified port. This is done to receive packets from processes which are to be monitored.
2. Read the configuration file and construct the structure for the processes listed there.
3. Wait for the packets at the binded port and handle them by updating the structure.

This was the basic control flow of the server. A detailed description of the structure which the server maintain follows. The handling of the various packets will be described after that.

The Structure

Requirements

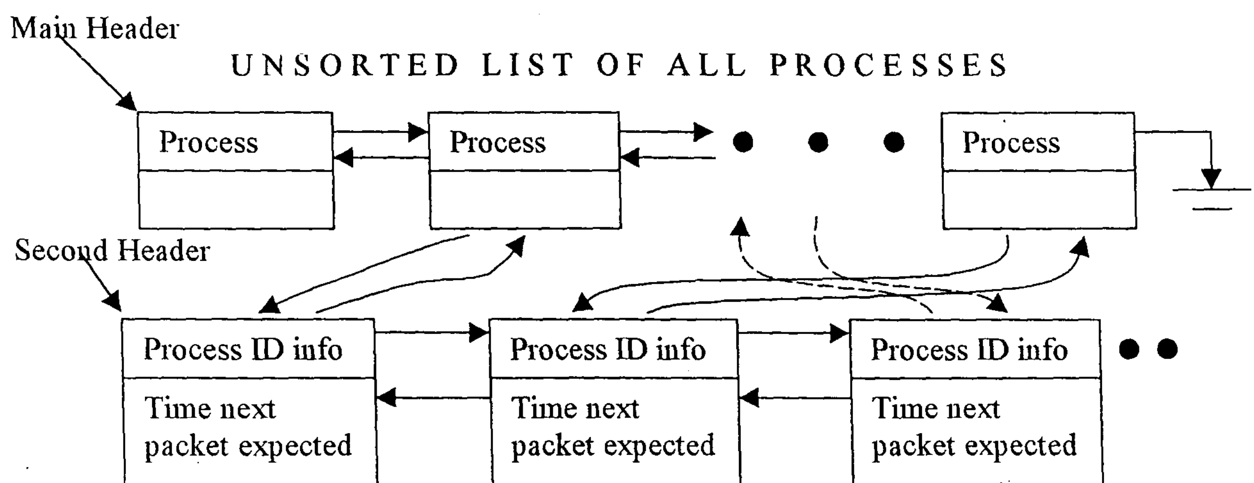
- The structure should be dynamically allocated as it keeps the details of all the processes it has been monitoring. Thus the structure will be a pointer maintained list structure.
- The structure needs to keep all the information about every process it has been monitoring since a down process can also go up.

- The structure needs to keep a sorted list of message expecting time. This will be used to detect any messages missed.

Thus a list of all the processes which are being monitored need to be maintained. The process may be up or down. Another list of processes from which packets are expected needs to be manitained. This excludes processes which are down. This list needs to be sorted according to when the next packet is expected from the process. The process state determines whether to expect packets from the process.

Details

Hence two lists will be needed. The processes which are down are kept differently from those which are sending packets. But this will necessaite keeping onle list sorted always and also moving all information from one list to other again and again. Thus the following structure is proposed.



SORTED LIST OF ALL PROCESSES BEING CURRENTLY MONITORED

The second list is sorted depending on the next expected packet receive time. The first list contains all processes which have been monitored. Thus the blocks having information about processes which are down are not pointed to by from the second list. Also since all information about a process may not be available hence there is a default value for every field. The first block in the second list has the least or the nearest time when to expect

packet from any process. A timer which times out at this time actually informs the server about any missed packets. The timer has to be updated whenever there is a change in the first block in this list.

The Packets and their Handling

After reading the configuration file the server updates the structure. As of now since there is not much information about the processes thus the default values are kept. The process state is given default values till packets are received from them. Their information is also updated in the process file.

The structure of the different types of packets and their handling follows :-

Information Packet : This is the packet which a process sends when it wants itself to get monitored. A process which has not sent this is not considered to have gone up.

This packet contains the following fields :

- Process Name
- Machine Name
- Process State
- Process Ending Criteria
- Time Slice

After receiving this packet the processes in the first list is searched. If it exists then its information is updated and according to the updated information various actions are performed. If updation had occurred in the first element of the second list then timer is reset, if there is a state change then the process file is updated. All these updations are done.

If the process is not present in the first list then a new block is added. Depending on the process state the various updations are performed. Like additions in both the lists and updation of the timer, process file.

Message Packet : This is the packet sent by the processes time to time to keep the server informed about their states.

This packet contains the following fields

- The Process Name
- The Machine Name
- The State of the Process

Whenever this packet is received the process it comes from is searched in the second list. If not found it is searched in the first list. If not found then the packet is ignored.
If found the process info is updated. This includes :

- Resetting the state of the process
- Resetting the time values
- Changing the number of misses allowed to the maximum allowed.
- Adjusting the place of the process block in the second list due to change in time.
- Resetting the timer when required
- Updating the process file if there is a state change.

End Monitoring : This packet is received when a process wants its monitoring to end. The process state is changed to "Ended" and it is removed from the second list. This is again followed by updations like the timer.

5.2 The Watcher

This Watcher has its own code. It does not share code with the server. This watcher is basically a monitoring process like the server but it has the following specifications also :

- It monitors only one process at a time – the server.
- It only takes care if that process(the server) is up or down. This is done by receiving packets in the same way as the server receives packets from the processes it monitors.

This is done to take care if server crashes. A fixed time slice is set and the sends message to the watcher whenever that time slice elapses. If continuously some number of messages do not come(a default setting) then watcher considers the server to have crashed and proceeds.

5.3 The Library

The library which can be used by the application programs to tell the process monitor server that it want to get monitored and also about its state. The states can be up state or the process can send a message to the server that it wants to end its monitoring. For these things to carry out the process first sends a information packet to the server containing all the information and after that sets an alarm for a specified number of

seconds so that the process can send message to the server about its status. And whenever the process wants to ends it monitoring it will kill the previous alarm and sends a end monitoring packet to the server.

5.4 Implementation Details

Introduction

This chapter gives details of what are the standard functions which will be used to carry out the processings listed in the last chapter. The implemantations issues have also been addressed. The implementation is being done in C.

5.4.1 The Packets

The packets are sent by the different processses to the server. The packets are sent after packaging them into a structure which is defined in the header. There are three types of packets as discussed earlier.

The information packet conatins the maximum information. It contains the time slice and the ending criteria which is used by the server for initializing the process details and monitoring it. But these two parameters are a global constant for any process. Once specified they remain fixed. Moreover they are only intergers and increase the size of the packet by only a small amount. All these put together and the fact that handling two different types of packets increases the complexity of the server by a large amount leads to the conclusion that only one type of packet may be kept. This packet will contain the same fields as the information packet. The message packets will also contain that extra information but since that information is anyway a global constant for the process it will not be difficult for the processes to send it again.

With this the processing on the server becomes quite easy. This also gives the processes an added flexibility. The processes can change their time slice and ending criteria from time to time. Thus this change makes things easier but better.

To further reduce complexity the end-monitoring packet also can be sent in the same structure as the other packet. Only the state can be made "Ended" to let the server

understand the type of the packet. As this packet is expected to be rarely used hence the extra information does not make a difference(which still is the same two integers).

5.4.2 The Timer Setting

In order to keep track of the packets missed the server is using a timer as already discussed. This timer should have the following properties :

- It should be non-blocking
- It should be asynchronous i.e. the timer should signal the process on timeout at arbit intervals as required.
- It should be possible to reset the timer whenever required cancelling the previous setting.

All the above requirements are satisfied by the alarm and signal function calls.

The alarm function takes as input an integer and timesout after the number of seconds sending a SIGALRM signal to the process it was called from. The signal function call handles any signal coming to the process asynchronously and calls a function specified in its arguments whenever the specified signal is received. The signal call takes the types of signal and the corresponding function to be called as input. The alarm gets automatically reset if called again before time out.

5.4.3 Getting the Process and Machine Name

The Library functions used by processes need to send the process and the machine name to the Monitor Server. This information needs to be extracted by the Library functions themselves. The following two standard C functions are used :-

gethostbyname : This function takes in a register pointer in which it returns the name of the process from which it called.

gethostname : It gives the name of the machine on which it is running into the string parameters it takes.

Using these two functions the library routines send the required packets to the server.

The time slice and the process ending criteria are to be provided by the process.

5.5 File and Function Details

Introduction

This chapter describes all the files in which the code will be structured and the functions that will be implemented along with the details of their functionality. The details of which file will have which function is also given.

The Pseudocode given in the next chapter completely describes each file and its function. The main code will be exactly in line with this Pseudocode.

5.5.1 The Files

The Header Files

The header files should be able to provide enough functions to the the main server such that it does not need to code any functions which deal with a specific type of processing. The processing and declarations done are as follows :

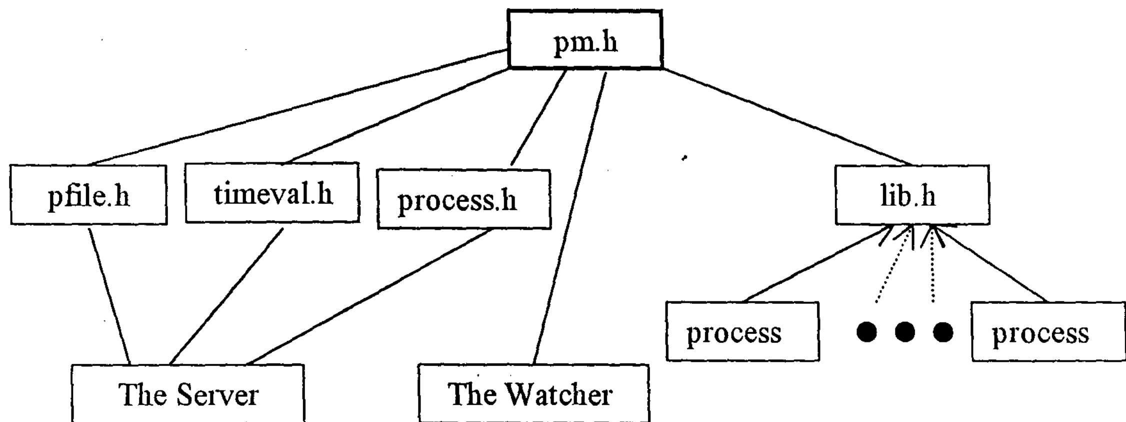
1. Global declarations for the server port, address, sizes of different strings like process name etc.
2. Global declarations of different structures like the packet, the process_info structure, the time list structure etc.
3. The processing to be done on the list structures like creating new blocks from information received, inserting them into the structure, checking for existence etc.
4. The processing that need to be done on the file that maintains the records of the processes.
5. The processing that needs to be done with respect to the timer setting and time manipulation.
6. The declaration of the various library functions that are used by the processes.

There needs to be a file for each of the above. The first two declarations can be done in one file as they will be needed by all the source files. The names of the files will be as follows

1. pm.h (all major declaration for process monitoring)
2. process.h (for processing on process_info)

3. pfile.h (for file manipulation)
4. lib.h (functions for the processes)
5. timeval.h (for processing on time)

The include structure of the files will be as follows :



The Source Files

There will be one source file which will be the server.c file,containing all the server processing and maintaining of the data. This file will compile into the main server which will receive all the packets and monitor the processes.

The other source file will contain the watcher code which monitors the server.

The Executables

There will be one executable of the server. This will not take any command line arguments and will run on a previously defined fixed machine.

There will be another executable of the watcher. This will be run whenever the server is run and will monitor the server.

5.5.2 The Functions and Declarations

The pm.h File

The following need to be declared

- The global port and IP address
- The various string lengths
- The process_info, time list and the packet structure

The process.h file

This file contains the functions for all the processing of the structure which includes

- Existence of a given process in the Process List
- Existence of a given process in the Time List
- Creating new process block from the packet information
- Creating new time block from the packet information
- Inserting a time block into the time list
- Inserting a process block into the process list
- Inserting both blocks together(this needs adjustment of pointers between them)
- Displaying the process list
- Displaying the time list

The pfile.h file

This is the file having the functions used for all the file manipulations and updations by the server. There will basically be one function which takes as input a process and its state and updates the file which includes adding the process if it is already not present.

The lib.h file

This is the main library file included by the process. It has the following functions.

- Opening a socket and binding the socket.
- Sending the information packet to the server.

- Setting an alarm for specified number of seconds and handling the signals from the kernel.
- Sending a message packet to the server from time to time.
- Sending an end monitoring packet to the server.

The server.c file

This is the main server source file. This will have the following functions:

- Opening and binding the socket
- Waiting for Packet
- Handling a Packet
- Handling a signal
- Updating the time list for resetting the alarm
- Read the configuration file
- The main

5.6 Pseudo Code

Introduction

This chapter gives the pseudo code of all the major files. The actual code will be in line with this pseudo code.

The Main Header File(pm.h)

Declarations

```
#define Process Name Size
#define Machine Name Size
#define The Global Server Port
#define The Process State Size
#define Maximum Packet Size
struct process_info
{       Process Name
        Machine Name
```

State

Time Slice

Process Ending Criteria

Allowed Misses

Other pointers

};

struct time_proc

{ Process Name

Machine Name

Next Packet Time

Other Pointers

};

struct mes_pack

{ Process Name

Machine Name

State

Time Slice

Process Ending Criteria

}

The Time Header File(timeval.h)

myalarm

Function Prototype

void myalarm(int tt);

Arguments

Tt	[IN]	Number of seconds for which the alarm is to be set

Description

This function sets an alarm for the specified number of seconds

Pseudo Code

void myalarm(int tt)

{

set an alarm for tt seconds.

}

get_cur_time

Function Prototype

long get_cur_time( );

Description

This function returns the current time time in seconds elapsed as returned by the gettimeofday function.

Pseudo Code

```
long get_cur_time( )
{
    get current time using gettimeofday in a timeval structure
    return the tv_sec part of the structure as the current time
}
```

The File Manipulation Header (pfile.h)

update_process_file

Function Prototype

void update_process_file(process name,machine name,state);

Arguments

Process name [IN]	The process to update
Machine name [IN]	The machine on which process is running
State [IN]	The current state of the process

Description

This function updates the state of the input process to the input state in the process file. If the process does not exist then makes a new entry.

Pseudo Code

```
void update_process_file(process name,machine name,state){
        open the process file for reading
        open a temporary for writing
```

```
        while (not end of file)
        {
                read the file contents of process file line by line
                if the input process found then copy it into temporary file
                        with the new state otherwise copy as it is
        }
        if process not found then copy it into temporary file as a new entry
        remove the process file and copy temporary into it
}
```

The Main Library Header(lib.h file)

This chapter describes all the function defined in lib.h file. These are all the functions included by the processes being monitored.

connection

Function Prototype

connection( )

Arguments

No Arguments

Description

This function opens a UDP socket and assigns a address.

Pseudo Code

```
connection( )
{
        get the process name using the gethostbyname function
        copy process name into packet field
        use gethostname to get the machine name directly in to packet field
        open a UDP socket using AF_INET family and SOCK_DGRAM as parameters;
        fill in cli_addr structure with AF_INET family, SERV_ADDRESS, port 0;
        bind the above socket;
        fill in the structure serv_address with AF_INET, server address and the server
        port
}
```

information_packet

Function Prototype

information_packet (int time_slice_in, int proc_end_crit_in);

Arguments

Time_slice [IN]	Identifies the number of second after which the next packet is to be send.
Proc_end_crit_in [IN]	Identifies the number of packets after which to assume the process is down

Description

This function prepares a information packet with the state "tostart" and sends it to the server.

Pseudo Code

```
information_packet (int time_slice_in, int proc_end_crit_in )
{
        assign time_slice_in to the packet field
        assign proc_end_crit_in to packet field;
        copy "tostart" in packet state;
        send the packet to server using sendto call
}
```

alarm_signal

Function Prototype

alarm_signal( )

Arguments

No arguments

Description

This function sets an alarm for the number of seconds specified by the process in packets time slice. Also this function handles the SIGALRM signal from the kernel and call the message_packet function.

Pseudo Code

```
alarm_signal( )
{
```

call the signal system call using SIGALRM and message_packet as parameters.

(message_packet is function called when SIGALRM signal is generated)

set the alarm using alarm for the specified time slice in the packet

}

message_packet

Function Prototype

message_packet( )

Arguments

No Arguments

Description

This function prepares a message packet with the message "up" and sends it to the server

Pseudo Code

message_packet( )

{

copy state as "up" in the packet state field

send this packet to the server

}

end_monitoring

Function Prototype

end_monitoring( )

Arguments

No Arguments

Description

This function sets an another alarm for zero seconds and handles the signal from the kernel and then calls the function end_packet

Pseudo Code

end_monitoring( )

{

call the signal system call to handle SIGALRM and call the function end_packet

set an alarm for one second. }

end_packet

Function Prototype

end_packet( )

Arguments

No Arguments

Description

This function prepares an end monitoring packet and sends it to the server.

Pseudo Code

```
end_packet( )
{
        copy the state "toend" in the packet state variable
        send this packet to the server.
}
```

The process.h file

This chapter describes all the functions defined in process.h file which are used for processing on the structures maintained by the server.

disp_proc_test

Function Prototype

```
void disp_proc_test(
                    struct process_info *pro_hd;
                   );
```

Arguments

pro_hd [IN]	Contains all the information

Description

This function displays the process_info structure.

Pseudo Code

```
void disp_proc_test( pro_hd )
        struct process_info *pro_hd;
{
        while (not end of list){display all the process list fields }
```

}

const_proc_block

Function Prototype

```
truct process_info* const_proc_block(
                        char *pro_nm;
                        char *mach_nm;
                        long  time_sl;
                        char  *pro_sl;
                        int   pro_end_mon;
                       );
```

Arguments

pro_nm	[IN]	This contains the process name
Mach_nm	[IN]	This contains the machine name
Time_sl	[IN]	This is the time slice after which next packet will come
pro_st	[IN]	This contains the process status
pro_end_mon	[IN]	This contains the process end criteria

Description

This function makes the process block from the packet data.

Pseudo Code

```
struct process_info* const_proc_block(
                        char *pro_nm;
                        char *mach_nm;
                        long  time_sl;
                        char  *pro_sl;
                        int   pro_end_mon;
                       )
{
        struct process_info *temp_proc_hd;
        malloc memory for temp_proc_hd;
        copy process name into temp_proc_hd->process_name;
        copy machine name into temp_proc_hd->machine_name;
```

copy process state into temp_proc_hd->state;

assign time_sl to temp_proc_hd->time_slice;

assign pro_end_mon to temp_proc_hd->alwd_misses;

return the temp_proc_hd pointer

}

const_time_block

Function Prototype

```
struct time_proc* const_time_block(
                                char  pro_nm[];
                                char  mach_nm[];
                                long  time_sl;
                              );
```

Arguments

pro_nm [IN]	This contains the process name.
Mach_nm [IN]	This contains the machine name.
Time_sl [IN]	This is the time after which the next slice is coming.

Description

This function creates the block of the time list given the information

Pseudo Code

```
struct time_proc* const_time_block(
                                char  pro_nm[];
                                char  mach_nm[];
                                long  time_sl;
                              );
{
        struct time_proc *temp_time_hd
        allocate the memory for temp_time_hd
        copy process name into temp_time_hd->process_name
        copy machine name into temp_time_hd->machine_name
        get current time in cur_time using gettimeofday
        assign temp_time_hd->time_next the current time added to the time slice}
```

insert_into_time

Function Prototype

```
void insert_into_time(
                        struct time_proc**  time_hd_hd;
                        struct time_proc*   time_blk;
                      );
```

Arguments

Time_hd_hd [IN]	Address of the time list head
Time_blk [IN]	Address of the time block to be inserted

Description

This function inserts into the time list a block containing all the information about the process. This function uses insertion sort.

Pseudo Code

```
void insert_into_time(
                        struct time_proc**  time_hd_hd;
                        struct time_proc*   time_blk;
                      );
{
        struct time_proc  *time_prev_hd, *time_hd;
        if (new time list)
        {
                make this block as the time list
        }
        else
        {
                if ( block time < header time)
                {
                        insert block at the head of the list
                }
                else
                {
```

```
            take time header into time_hd;
            take previous header into time_prev_hd

            while ( time_hd  != NULL )
            {
                    if current time header's time > time block's time then insert
                    time block here
            }
            if time block still not inserted then insert at end;
        }
    }
}
```

insert_into_proc

Function Prototype

```
void insert_into_proc(
                    struct  process_info**  pro_hd_hd;
                    struct  process_info*   pro_blk;
                    );
```

Arguments

pro_hd_hd	[IN]	Address of the process list head
pro_blk	[IN]	Address of the process block to be inserted

Description

This function inserts into the process list a block containing all the information about the process.

Pseudo Code

```
void insert_into_proc(
                    struct  process_info**  pro_hd_hd;
                    struct  process_info*   pro_blk;
                    );
```

```
{
  if (new process list)
  {
        make process block the process list;
  }
  else
     {
          insert process block at the process list header
     }
}
```

insert_blocks

Function prototype

```
void insert_blocks(
                    struct process_info**  pro_hd_hd;
                    struct process_info**  pro_blk;
                    struct time_proc**     time_hd_hd;
                    struct time_proc*      time_blk;
                   );
```

Arguments

pro_hd_hd [IN]	Address of the process list head
pro_blk [IN]	Address of the process block
Time_hd_hd [IN]	Address of the time list head
Time_proc [IN]	Address of the time block

Description

This function inserts a given time block and its corresponding process block into their respective lists.

Pseudo Code

```
void insert_blocks(
                    struct process_info**  pro_hd_hd;
                    struct process_info**  pro_blk;
```

```
                    struct time_proc**      time_hd_hd;
                    struct time_proc*       time_blk;
                    );
{
   adjusts pointers between time and process block;
   call insert_into_proc(pro_hd_hd, pro_blk);
   call insert_into_proc(time_hd_hd,time_blk);
}
```

exists_proc

Function Prototype

```
struct  process_info *exists_proc(
                                struct process_info*  proc_hd;
                                char                  proc_name[];
                                char                  mh_name[];
                                );
```

Arguments

Proc_hd	[IN]	Address of the process list header
Proc_name	[IN]	This contains the process name
mh_name	[IN]	This contains the machine name

Description

This function checks for the existence of a given process in the process list

Pseudo Code

```
struct  process_info *exists_proc(
                                struct process_info*  proc_hd;
                                char                  proc_name[];
                                char                  mh_name[];
                                );
 {
         while ( (not end of list) and (not found))
         {
```

match current machine and process name with given

if matched then found else move to next block in the list

}

if not found then return NULL else return the current pointer.

}

exists_time

Function Prototype

```
struct time_proc* exists_time(
                        struct time_proc*  time_hd;
                        char               proc_name[];
                        char               mh_name[];
                        );
```

Arguments

Time_hd	[IN]	Address of the time list header.
Proc_name	[IN]	This contains the process name.
Mh_name	[IN]	This contains the machine name.

Description

This function checks for the existence of a given process in the time list

Pseudo Code

```
struct time_proc* exists_time(
                        struct time_proc*  time_hd;
                        char               proc_name[];
                        char               mh_name[];
                        );
{
    while ( (not end of list) and (not found))
    {
            match current machine and process name with given
            if matched then found else move to next block in the list
    }
    if not found then return NULL else return the current pointer.}
```

The Main Server Code(server.c)

open_bind

Function Prototype

void open_bind( )

Arguments

No arguments

Descriptions

This function opens a UDP socket and binds it to a

Pseudo Code

```
void open_bind( )
{
        open  a UDP socket using AF_INET family and SOCK_DGRAM as parameters
        fill in cli_addr structure with AF_INET family, SERV_ADDRESS, port 0
        bind the above socket;
}
```

update_time_list

Function Prototype

void update_time_list( )

Arguments

No arguments

Descriptions

This function updates the time list by considering all those processes which have timed out.

Pseudo Code

```
void update_time_list( )
{
        get the current time in time_current.
        while ( not end of list & alarm not set)
        {
                if (header process timed out)
```

```
        {
                updates its time_next field.
                remove and reinsert in time list.
        }
        else
        {
                set alarm for the header process
        }
    }
    if (end of time list reached)
    {
        prompt "no process is being monitored".
    }
}
```

handle_pack

Function Prototype

```
void handle_pack(
                struct mes_pack* recv_pack;
                );
```

Arguments

Recv_pack [IN]	Address of the received packet

Descriptions

This function handles the packet that are coming.

Pseudo Code

```
void handle_pack(
                struct mes_pack* recv_pack;
                )
{
    if (received process exists in time list)
    {
        update process fields according to the received packet
```

```
            ending criteria = received ending criteria
            allowed misses  = ending criteria
            time slice = time slice received
            next  time = current time  +  time slice
        if state changed then update process file also
        now remove and reinsert in the time list.
        update time list
}
else
{
        if (received process exists in the process list)
        {
                update the process fields according to the received packet
                        ending criteria = received ending criteria
                        allowed misses  = ending criteria
                        time slice = time slice received
                        process state = received state
                if (received state not "down" or "ended")
                {
                        create new time block according to received packet
                        adjust pointers with the corresponding process
                        insert into the time list
                        update time list
                }
        }
        else
        {
                consider the received packet as an information packet
                create new process and time blocks
                insert them into the lists
                update the time list
```

```
        }
    }
}
```

receive_pack

Function Prototype

void receive_pack( )

Arguments

No arguments

Descriptions

This function receives a UDP packet at the server port.

Pseudo Code

```
void receive_pack( )
{
    for()
    {
        receive a UDP packet in the structure form at the SERVER_PORT
        call the handle_pack function to handle this packet
    }
}
```

handle_signal

Function Prototype

void handle_signal( )

Arguments

No arguments

Descriptions

This function handles the time out of a process and does the necessary updating in the time and process list

Pseudo Code

```
void handle_signal( )
{
    get current time into time_current
```

```
while (list not ended and alarm not set)
{
        if (header process timed out)
        {
                if (no misses allowed)
                {
                        update the time fields
                                change state to down
                        remove the process from the time list
                }
                else
                {
                        decrease allowed misses by one
                        update the time_next field
                        remove and reinsert in the time_list
                }
        }
        else
        {
            reset alarm to the time_list header value
        }
}
if (end of list reached)
{
        prompt "no process being monitored"
}

signal(SIGALRM,handle_signal)
call receive_pack for more packets
}
```

read_config_file

Function Prootype

```
void read_config_file(
                        char *fn
                    );
```

Arguments

fn [IN]	This contains the file name

Descriptions

This function read the process and machine name from the configuration file and set defaults values of other fields of the structure.

Pseudo Code

```
void read_config_file(
                        char *fn
                    );
{
        open the config file
        while (end of file not reached)
        {
                read the process and machine name from the file
                set default values of other fields as specified in header
                create new blocks and insert in the lists
                update the time
        }
}
```

CHAPTER 6

INTERFACE CONTROL

The Interface

Introduction

The process monitoring software contains three different entities which interface with each other. These are :-

- The Monitor Server
- The Watcher
- The processes being monitored.

The server communicates with the other two while the watcher and the processes monitored do not communicate. The following sections describe both the interfaces in detail.

6.1 The Server-Process Interface

6.1.1 Introduction

Every process which is monitored inerfaces with the same server. This is the main interface of the software. The following communication is to take place between the server and the monitored process :-

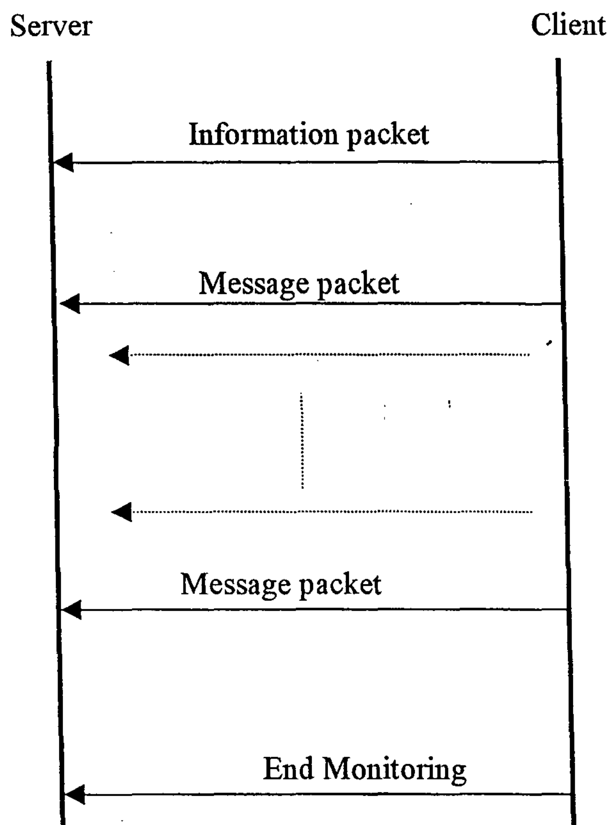
- The process sends packets to the server when itwants the server to start monitoring it
- The processes being monitored send a packet from time to time to the server
- The server receives these packets and processes them

The server does not send any packet to the processes. The processes assume the server to be always up. The context diagram of the interface is given in the next section.

6.1.2 Context Diagram

The context diagram is given on the next page. The process which is being monitored sends the different types of packets to the monitor server. Thus the interface is simple and

direct. Since the same packet can be considered to be a message or a information packet or a end-monitoring packet (as explained in the Low Level Design) hence the order in which the packets arrive does not matter to the server(the context diagram just shows the expected ordering of the packets). Moreover though the diagram shows server inteface with only one packet there are many processes interfacing with the server at a time.



Ports Used

The port used by the server to listen is as defined in the main header file. The client uses any arbitrary port to send its message. The server is bind to a fixed machine which is defined in the main header file.

6.1.3 Packet Structures

Following is the structure of the packet which the processes send to the server.

Process name
Machine name
State of the process
Time slice
Process end criteria

The above information is packed into a structure and sent to the the server.

6.2 The Server-Watcher Interface

The server and watcher communicate in the same way as the processes and the server only that the role being palyed by the server in the server-process interface is now playes by the watcher and the role of the process is played by the server.

6.3 Conclusion

This concludes the interface detail of the software. Both these interfaces are controlled by the server making it the major interface control process of the software. The processing being done both on the server and the watcher side is such that any packet misses are taken care of.

CHAPTER 7

THE TEST PLAN

Introduction

The test plan can be divided into the different types of test cases as follows :

- Testing the Server
- Testing the Library
- Testing Watcher
- Testing the Interfaces

Since the software is integrated in such a way that functioning is not possible without the all the different modules running hence most of the testcases come under the Integrated Testing section.

7.1 Server Testing

Introduction

The server needs to be rigorously tested broadly for the following sections :

✓ Maintenance of the Process List Structure
✓ Maintenance of the Time List Structure
✓ Resetting of the Timer

7.1.1 Maintenance of the Process List Structure

Purpose

To check whether a process state in the list is changed whenever a process sends a different state.

Procedure

1. Make the server start monitoring a process which is running and sending packets.
2. Now make the process send a packet with a different state

Expected Result

The process state should change in the process list.

7.1.2 Maintenance of the Time List Structure

Purpose

To check whether a new process is added at the right place in the list

Procedure

1. Make the server monitor a process of small time slice
2. Now start a process with a small time slice
3. Now start a process with a modearate time slice such that it times out after the second but before the first process

Expected Result

At step 2 the new process should be added at the start of the time list. At the step 3 the new process should be added in the middle of the time list

7.1.3 Resetting of the Timer

Test Case 1

Purpose

To test whether the timer is reset when a new process sends packet with small time slice

Procedure

1. Make the server monitor one running process with a large time slice
2. Start a new monitoring process with a small time slice with early time out.

Expected Result

The timer should be reset for the new process

Test Case 2

Purpose

To test whether the timer is set to next timeout rather than time slices

Procedure

Keep two processes having co-prime time slices being monitored by the server

Expected Result

The alarm setting should be for the next elapsing of the time slice.

Test Case 3

Purpose

Check whether the time list is updated correctly when two processes time slice at the same time

Procedure

1. Start two processes at the same time(least count of seconds) which have time slices which are multiples.
2. Now stop both the processes

Expected Result

Everytime the alarm times out at a common mutiple both packets should have reduction in their allowed misses.

7.2 Library Testing

7.2.1 Signal Testing

Purpose

To verify that the signal is received by the process on time

Procedure

1. Include this functions in program which has a infinite loop
2. Print all the packet structure values on standard screen and check the time value
3. Run the program

Expected Result

Each time the value should be printed after the time interval set.

7.2.2 Checking the End Monitoring Packet

Purpose

To verify that after sending the end monitoring packet that no more packet is being sent

Procedure

1. Include these functions in a program which has a infinite loop.
2. In the infinite loop get a particular character from the user after getting that character call end_monitoring function.
3. Check the standard screen

Expected Result

After pressing the desired character no more packet data should be printed on the screen.

7.3 Testing the Watcher

Purpose

To check whether the watcher behaves conrrectly when server crashes.

Procedure

Just make the server crash in between of normal processing

Expected Result

The watcher should dump all the information in the process file into another file and exit after creating another file which gives information about the time of crash.

7.4 Testing the Interfaces

Purpose

To check whether packets are being received from different machines on the network

Procedure

Start different monitored processes on different machines

Expected Result

The server receives packets from all the processes at the right time intervals

CONCLUSION

The project involved the creation of process monitor server which monitors the processes over the network individually, as per the user and the process requirements and specifications. The project code was successfully tested according to the test plan. As the current testing was done using dummy clients further testing can be carried out to check the robustness of the server.

This approach helps the network manager to remotely log on to the machine on which the process monitor server is running and collects the data about the behavior of the various processes. These statistics help in restricting the processes using the system resources massively.

Further enhancements can be done on the server side by adding mutual exclusion as signals can time out between list processing. Similarly enhancements can be carried out in the API by using better methods for re-sending packets rather than SIGALRM signal as it can be used only once and processes which are already using that signal cannot use the API in the present form.

An alternative approach to the work done in this dissertation is instead of using a single server to monitor the processes in the network we can connect servers of the different local area networks to the process monitor server. This approach could help in reducing the load on the single process monitor server.

References

1. Andrew S. Tanenbaum, "Computer Networks," Third Edition, Prentice Hall of India, 1999.
2. B. Kernighan and R. Pike, "UNIX Programming Environment," Prentice Hall of India, 1998.
3. Douglas E. Comer, "Computer Networks and Internet," Second Edition, Prentice Hall of India, 2000.
4. Douglas E. Comer, "Internetworking With TCP/IP," Vol 1: Principles, Protocols, and Architecture, Third Edition, Prentice Hall of India, 1999.
5. Douglas E. Comer and David L. Stevens, "Internetworking With TCP/IP," Vol 2: Design, Implementation, and Internals, Prentice Hall International Editions, 1991.
6. Gray. R. Wright and W. Richard Stevens, "TCP/IP illustrated, Volume 2 The Implementation," First ISE 1999.
7. M. Schulze and Craig, "Network Monitoring and Visualization Tools, " http://coast.cs.purdue.edu/pub/tools/unix/netmon/netman.
8. Paul M. Sittler, "A Web based UNIX network monitoring and notification system," February 1997.
9. R. Stevens, " UNIX Network Programming," Prentice Hall of India, 1999.
10. W. Richard Stevens, "TCP/IP Illustrated," Volume 3: TCP for transaction, HTTP, NNTP, and the UNIX domain protocols, Addison Wesley, 1999.