Development of a Control Process for the Berkeley UNIX Distributed Programs Monitor

Cathryn Marcia Macrander

Computer Systems Research Group
Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720

Abstract

Conventional programs have a single stream of execution. Distributed programs expand on this notion, having multiple streams of execution that interact with each other. This expansion increases the complexity of a program's behavior. Existing program monitors do not provide enough information to deal with all of the problems of a distributed computing environment. This paper is concerned with the development of a programming tool for Berkeley UNIX whose goal is to characterize the performance of distributed programs.

The Distributed Programs Monitor (DPM) monitors specifically the interactions between the processes of a distributed program and provides routines to analyze the resulting data. DPM is a tool composed of independent subtools that work together to monitor a distributed program. This paper will present a overview of DPM including a bit of its history and a bit of its experimental use, but it deals primarily with the development of a control process for the monitor. The design and the implementation of this process is described. A major issue of the design addresses how to provide distributed process management in a nondistributed processing environment.

Research supported by the National Science Foundation grant MCS-8010686, the State of California MICRO program, and the Defense Advance Research Projects Agency (DoD) Arpa Order No. 4031 monitored by Naval Electronic System Command under Contract No. N00039-82-C-0235. The views and conclusions contained in this document are those of the author and should not be interpreted as representing official policies, either expressed or implied, of the Defense Advance Research Projects Agency or of the United States Government.

ACKNOWLEDGEMENTS

This Masters project was completed with the help, support, and guidance of several individuals.

First of all, I would like to acknowledge a great debt to Bart Miller who directed this research project. He was an excellent teacher and a valuable source of motivation and encouragement.

I would also like to thank my research advisor, Domenico Ferrari, and all of the members of the Progres research group. Working with them was an extraordinary learning experience. In particular, I would like to thank Stuart Sechrest who worked with me on the project. He deserves much credit.

During my stay at Berkeley, I was awarded a Graduate Opportunity Fellowship from the Graduate Division of the University of California. I also received financial support in the form of a research assistantship from the department of Electrical Engineering and Computer Sciences. I would like to express my gratitude to the people who made it possible.

Finally, I would like to thank my family for their constant support during my college years. Without their help I could not have come this far.

1. Introduction

As technology advances, computing power is becoming a more abundant resource. Multiprocessor systems and networks of computers are providing programmers with powerful computing environments. The potential of such environments encourages the notion of parallelism within a program. Concurrent execution, however, introduces obvious and subtle complexities into program behavior. Conventional program debuggers and performance measurement tools do not provide enough information to deal with the added complexity. There is an apparent lack of instrumentation with the required capabilities. Therefore, tools are needed that specifically address the programming problems introduced by a distributed computing environment. The Distributed Programs Monitor for Berkeley UNIX is one such tool.

The Distributed Programs Monitor (DPM) is a monitoring tool whose goal is to characterize the performance of distributed programs. Based on the observation that process interactions are just as important to a distributed program's execution as the functioning of the individual processes, it is the process interactions within a distributed computation that DPM monitors. DPM provides the user with the means for obtaining a trace of the process events that occur during the execution of a distributed computation. The events we monitor are those that signal process creation and process termination, describe the creation and destruction of communication paths, and relate to the sending and receiving of messages.

In addition to providing a trace of process events, DPM includes analysis routines to interpret trace data. Although the analysis routines will only be mentioned briefly in this paper, they are an essential part of DPM. To date, analyses that have been made include the simple problem of summarizing message statistics. They also include the more complex problems of determining a measure of the amount of parallelism, or speed up, achieved in the program's execution, and determining the flow of control through a computation[Miller 84].

DPM does not depend on the existence of a global, network-wide clock to establish an ordering of events adequate for most analyses. However, it does not preclude the existence of such a clock, either. Berkeley UNIX provides a fairly accurate approximation of a network wide time[Gusella & Zatti 83]; thus, the independence of DPM from a global time is not dictated by necessity. If a global time is not assumed, however, certain analyses such as that of parallelism mentioned above may require some information that cannot be extracted from the trace. The availability of a global clock would have the effect of making such analyses simpler.

Transparency is a key feature of DPM. DPM is explicitly not a distributed programs debugger, even though the information it provides may be very valuable in debugging. It is a passive monitor that observes a computation with no intent to modify its execution. Furthermore, programs do not have to be recompiled in order to be monitored. This transparency allows system processes as well as user processes to be monitored.

Organizationally, DPM is tool composed of smaller tools. Meters, filter processes, and a control process are the individual pieces of the tool that cooperate during a monitoring session. Analyses are run on the trace after the computation has completed.

In this paper, we are primarily concerned with the design and the implementation of Berkeley UNIX DPM's control process. The role of a control process is to serve as a *liaison* between the different entities that are active during the monitoring of a program. It interacts heavily with the user, the different parts of DPM, and the operating system. In order to establish a context for the discussion of control process design, we will first present an overview of DPM, introducing information that is relevant to the discussion of the control process. A more detailed description of the other parts of DPM, such as the meter and the analysis routines, are found in[Miller *et al* 84, Miller 84].

In the following section, we will present definitions and goals that underlie the design of DPM. We assume some familiarity with the 4.2BSD interprocess communication (IPC) facilities. For a complete description of Berkeley UNIX IPC see [Leffler 83, Sechrest 84]. Section 3 will

present a short history of the tool. An overview of DPM will be given in Section 4, after which we will proceed to a detailed discussion of a control process design. Section 6 will follow the design details with the implementation details. We will conclude the paper with a presentation of an experiment, an analysis of DPM's implementation, and a look at how the tool may evolve in the future.

2. Foundations

2.1. Model of Computation

A program composed of multiple, interacting processes cannot be described by a traditional model of computation. Traditionally, a program involves a single stream of execution in which one program instruction is executed at a time. The model of a single-process program does not have to account for process interactions. Performance instruments for such programs provide metrics such as the number of page faults, context switches, and i/o operations. Such metrics do not provide all of the information we need to characterize a distributed computation. Interactions are a key to understanding a distributed program. To characterize a distributed computation, we have to characterize process interactions. Since traditional models do not have the notion of interactions, we must devise another, more general model of computation that does encompass this notion.

We define a distributed program as a collection of processes that cooperate in a computation. This model is depicted in Figure 2.1.

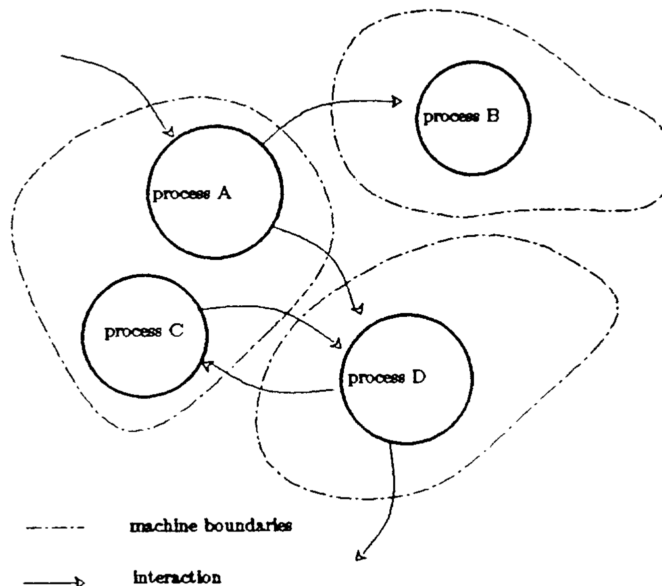

Figure 2.1 Example of a Distributed Computation

A process is an entity that computes and communicates. Processes have private address spaces and communicate with other processes via messages. A distributed computation executes on a network of processors, and its processes may be executing on the same or different processors. The type of the underlying network is unimportant as long as processes on different processors can communicate.

With this definition, we can model a computation that contains multiple concurrent streams of execution interacting with each other. In this model, a process sees the operating system as another process.

2.2. Measurement Goals

There are two objectives which we have maintained throughout the design and the implementation of DPM. These objectives are *consistency* and *transparency*. The first, consistency, is concerned with the class of events that we monitor. We believe the events DPM monitors should be consistent with how a user sees his or her program. When writing a program, a programmer uses various primitive functions. A user views the flow of control through a program as a sequence of these functions. Therefore, these primitive functions are the events DPM should trace for the user.

The second goal, transparency, is an important feature of DPM. DPM is meant to be a performance tool. As a performance tool, it should do as little as possible to modify the program's behavior. A monitor is a passive tool, not an active one. Use of the monitor should not alter the order in which process events occur. In fact, it is precisely this sequence of events that we would like to trace.

We also take transparency to mean that we should be able to monitor any program. Therefore, we should not require a program to be recompiled before it can be monitored. No subroutine calls to special run-time monitoring routines should have to be inserted into the program's source, and no special run-time libraries should have to be included. We want to provide a tool that is *applied* to a program, and does not have to be *integrated* into a program.

3. Evolution of DPM

DPM for Berkeley UNIX is a programming tool that has evolved and is still evolving. The original version of a distributed programs monitor based on the models and goals discussed in the previous sections was working as of September 1983[Miller 84]. The tool was first developed for the DEMOS/MP operating system[Baskett *et al* 77,Powell 77,Powell & Miller 83]. For several reasons, DEMOS/MP provides a much cleaner model of a distributed world than Berkeley UNIX. The DEMOS/MP environment is more uniform in nature. The implementation of the monitor in this environment produced a relatively simple tool structure. From this simple structure, DPM for Berkeley UNIX has evolved. In porting the distributed programs monitor from DEMOS/MP to the "uglier", less uniform UNIX environment, the basic structure of the tool remained simple, but additional complexity had to be introduced. The design of a control process discussed in section 5 will provide evidence of this added complexity.

3.1. DEMOS/MP

DEMOS/MP is a communication-based operating system. The interprocess communication mechanisms are implemented at the lowest level of the system. Most of the systems functionality is provided by system processes executing outside of the system's kernel. DEMOS/MP processes do not share memory. They have only one way to interface to the outside world, that is, via messages. Processes use messages to interface to the operating system, to system resources such as the file server and the name server, and to other user processes. Since a single interface is used between all DEMOS/MP processes, a process is encapsulated by the communications paths, or links, that it possesses. All of its external activity is visible through its links.

Another characteristic of DEMOS/MP that adds to the uniformity of its environment is that process and memory management are transparent across machine boundaries. We can manage a process located on a remote machine as easily as we manage a local process. The operating system processes for name service, process management, and file management provide an environment amenable to the implementation of a distributed computation.

3.2. Berkeley UNIX

The UNIX world looks quite different from that of DEMOS/MP. As with DEMOS/MP, UNIX processes do not share memory. In great contrast to DEMOS, UNIX is not a communication based operating system. The interprocess communication (IPC) mechanisms are

not implemented at the lowest level of the system. UNIX IPC facilities were an afterthought and as such were built on top of UNIX as a separate abstraction having its own name space. Also in contrast to DEMOS/MP, UNIX is a monolithic operating system; most of the system's functionality lies in the system's kernel. These functions are accessed via system calls, not via messages. Consequently, a process has different ways for talking to the external world.

There are also multiple ways for the outside world to talk to a process. For example, the signal mechanism addresses a process via its process identifier, but the IPC mechanism addresses a process via its own abstraction, an IPC socket. The consequence of multiple external interfaces is that there is no simple way to encapsulate a process so that we can observe all of its external interactions.

In further contrast, machine boundaries are not transparent in a UNIX system. Process and memory management mechanisms are not uniform across machines. The state spaces and identifiers of processes executing on a host cannot be directly addressed by the kernel of a remote host. Controlling processes across boundaries is therefore more complicated than controlling processes within a single machine. In addition, the current version of Berkeley UNIX (4.2BSD) has no distributed file system or distributed name server.

One result is that the lack of network transparency is a primary obstacle in providing a simple control process design.

4. Overview of Berkeley UNIX DPM

The process structure and organization of our monitor resembles the organization of METRIC[McDaniel 75]. First of all, as in METRIC, the monitoring detection mechanism is put at a low-level. It is at the system level, not at the language level. Second, the collection and storage of event information are separated from the analyses. This organization, which predates METRIC by many years (see for example[Ferrari 78]), allows for a great deal of flexibility.

It differs from METRIC on the issue of transparency. METRIC requires the insertion of special subroutine calls to monitoring functions, or *probes*, to be inserted into a program's source. This is a mechanism we want to avoid. METRIC is also dependent on the type of the underlying network, assuming that all messages are broadcast and thus can be overheard by everyone.

Based on our models and on our goals, the following structure was created, first for DEMOS/MP and then for Berkeley UNIX. There are three stages involved: (1) event detection and event record generation, (2) data collection and storage, and (3) data analysis. Program monitoring stage includes the first two stages, event detection and data collection. There are three types of processes or mechanisms involved in program monitoring: the *meters*, the *filter processes*, and a *control process*. The organization of these parts is shown in Figure 4.1.

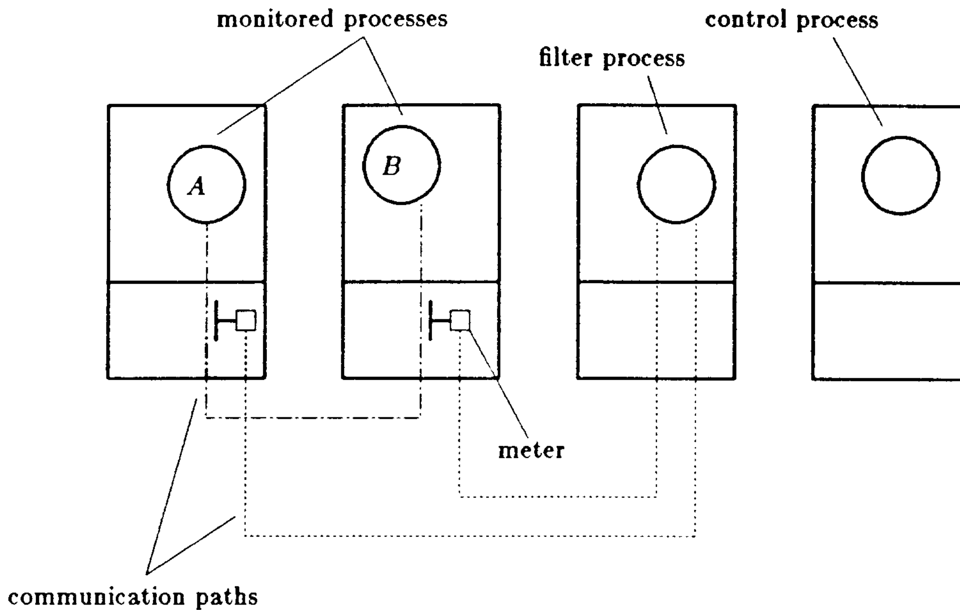

Figure 4.1 Organization of our Monitor

The meters are responsible for detecting events and creating event records. They are mechanisms located in the kernel of each machine. The filter processes are user processes that collect the event records produced by the meters and store these records in a trace file. A filter may also perform data selection and data reduction operations. The control process keeps the whole organization working together. It provides the user interface to DPM and it is responsible for making all of the needed connections between the different parts of the tool.

To date, the third stage of the measurement model, the analysis routines, do not participate in the monitoring session. Analyses are run on the trace following the collection of data. However, there is nothing in the design which restricts the analyses to be run only at this time. A filter process could pass data directly to an analysis routine, allowing for realtime monitoring. Such ideas are described in Section 8.3 on future research.

4.1. The Meters

The meter mechanism is located in the kernel. If a process produces an event which has been specified for monitoring, the meter will detect the occurrence of this event. An event message will be created with data extracted from system structures and sent over a communication path to a filter process for processing outside of the kernel. Each monitored process must be provided a communications path to a filter process.

In order to implement the kernel metering facility, three fields were added to the process structure in the process table as depicted in Figure 4.2.

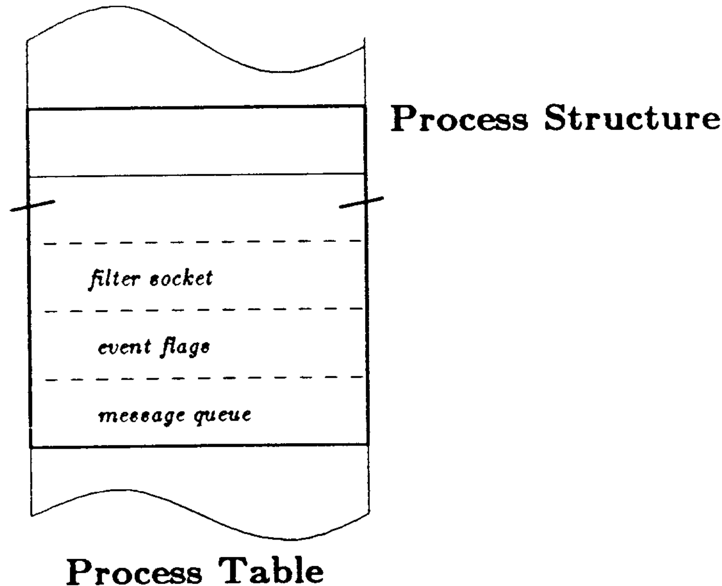

Figure 4.2 Augmented Process Table Entry

These three fields are (1) a pointer to a socket connected to a filter process, (2) a bit mask that indicates which events are to be monitored, and (3) a pointer to a list of buffered event records. When a sufficient number of records are buffered, a meter message containing the event records is sent to the filter process. This reduces the number of system generated messages caused by monitoring. The measurement strategy is defined be the values of these fields. The interface to the kernel meter is via the system call *setmeter()*. This interface is used by a control process when setting the measurement strategy for the user. A process is not monitored by default. Monitoring must explicitly be set for a process via a *setmeter()* call. If a monitored process forks, however, its child will inherit its measurement strategy. There are situations in which a server is used to create processes. Examples of such a class of servers are *rexec()*, file servers, or name servers. In these cases, only if the server is being monitored will a process it creates be monitored.

The events for which monitoring is currently available are:

ACCEPT	process accepts a connection
CONNECT	process initiates a connection
SEND	process sends a message
RECEIVECALL	process makes a call to receive a message
RECEIVE	process receives a message
SOCKET	process creates a socket
DESTSOCKET	process destroys a socket
FORK	process forks
TERMINATION	process is destroyed

Each of these system calls is examined by the kernel meter. When a monitored process makes one of these calls, a meter message is generated and sent across the filter connection. The design of the meter does not limit the monitored events to those listed above. These were chosen because of our primary interest in characterizing performance based on process interactions via messages. The meter could be extended to include more events. For example, we might like to detect file activity (for processes that communicate via a file). In this case, interesting system calls such as *open()*, *read()*, *write()*, and *close()* could be modeled as *<send(),receive()>* pairs.

The event records generated by the meter consist of a header and a record body. The header contains the machine number, the size of the message, the local clock time, the amount of CPU time the process has consumed, and the event type. The record body contains the process identifier, the value of the program counter, and the addresses of the source and/or destination sockets, depending on the event type.

4.2. The Filter Processes

The selection, reduction, and storage of trace data is the responsibility of a *filter* process. A filter is a process that executes in user space. It is the recipient of the measurement data that are generated by the meters. The user may only be interested in saving meter events which match certain criteria. For example, the user may only want to save event records of messages having a message length over a specified minimum, or event records of a process only when the CPU time used by that process is under a specified value. It is the filter that interprets the selection criteria and decides which event records are to be kept and which are to be discarded.

Filter processes do not exist by default in the measurement tool. The user must tell the control process to create a filter process. Many filter processes may exist simultaneously. Usually, there will be a filter process created per computation. A standard filter is provided by DPM. However, under only a few basic constraints, custom filters can be easily written.

4.3. The Control Process

The task of organizing the parts of the measurement system and providing a control interface to the user is performed by a *control* process.

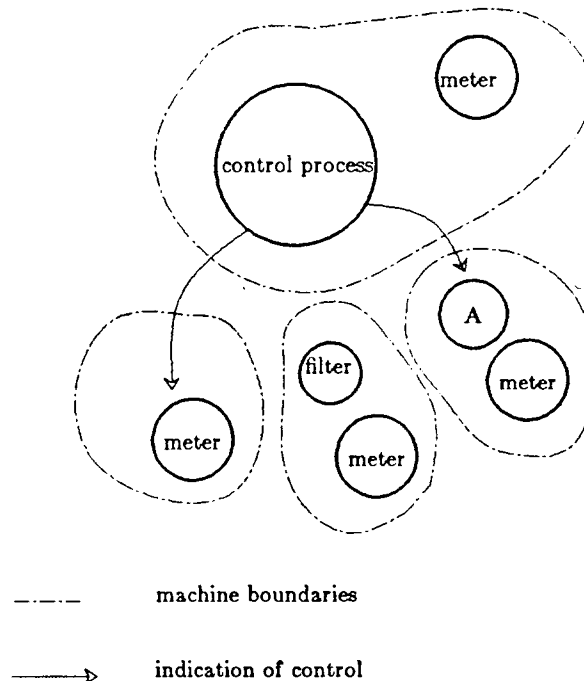

Figure 4.3 Role of Control Process

A control process provides the mechanisms for establishing the communication paths between all of the components of the measurement system. A control process is a command interpreter. It provides the user with a concise menu of commands to use in the measurement and control of one or more distributed computations. A control process, upon receiving a command, interprets the command and performs the requested function. Executing this request may require interacting with the other components of the measurement system and establishing communication paths

between the various components.

Control processes are user processes that exist on a per monitoring session basis. One user's control process is independent from another user's control process. A given user could have multiple control processes, just like he or she could have multiple shells. In the normal case each user will have one control process and may use it to monitor one or more computations.

5. Design of a Control Process

DPM is a collection of cooperating entities: meters, filter processes, control processes. The meters exist independently from the other parts of DPM. The meters are part of the system kernel and therefore are mechanisms that are available even if no instantiation of DPM exists. Filter processes, on the other hand, are created by the user. Once created, a filter process also exists independently from the rest of DPM. In addition to meters and filter processes, we have a computation or computations which are to be monitored. These computations know nothing about meters or filter processes.

Unless we devise some way to tell meters which processes to monitor and where to send the event records, we cannot use our tools to monitor a computation. Establishing the necessary communication paths between meters and monitored processes and filter processes is the responsibility of the control process. The control process creates the communication paths according to the specifications of the user. The programmer makes such specifications through an interface provided by the control process.

5.1. Functionality

Before discussing the issues that arise concerning how the control process will do what it is supposed to do (the mechanisms), we will first outline what functions we would like the control process to support.

Approaching this problem from the perspective of a potential user, we must be able to create a process as part of a computation. We would like a process to be created without starting execution. This is necessary to give us sufficient time to specify the desired measurement strategy and to provide some control over the time that various processes start executing.

Next, we must be able to specify and change the monitoring strategy for a computation. As well as setting this strategy for processes we have created, we would like to be able to monitor a process or computation that already exists (we have termed this action *acquiring* a process). This would give us the ability to monitor processes such as system servers. System server processes are always running (hence, they don't have to be created) and may be interesting to characterize. When we acquire an existing process for monitoring, no part of its environment or status should be changed other than the setting of the monitoring fields of its process table entry. The privilege of acquiring a process should only be given to us if we have the proper authority (e.g., only if we are root we should be allowed to monitor a system process).

In addition to needing a way to set the monitoring strategy, we must have a way to create a filter process for data collection. A standard filter process should be available if we do not need a special filter process. However, there should also be a way for more advanced users to experiment with different filter processes.

We would like to have basic process control, as in stopping and starting processes. We should be notified about process terminations and should receive the standard output produced by the processes we create. Finally, we must be able to retrieve the trace of process events that results from a monitoring session for use in subsequent analyses.

5.2. Evolution of Design Issues

5.2.1. Process Control

Process control is an issue of central importance to the design of a control process. Process control is required to create a process and to have it monitored, as well as to provide some control over a process's execution. Since no restrictions are placed on the processor locations of processes, DPM's control process must be able to manage processes on machines different from the one it runs on.

When our world is limited to a single machine, the level of process control we need is easily accomplished through existing system functions and utilities. To create a process, a standard UNIX system provides the *fork()* and *exec()* functions. To subsequently start, stop, and terminate the execution of a process, UNIX signals are used. Through the UNIX *signal()* system call, changes in a process's state (such as normal process termination) can be detected, and control can temporarily be given to signal handling routine for signal processing. The problem of process control has a readily available solution for a single machine environment running under Berkeley UNIX.

UNIX is not a distributed system, however, and the multiple machine world looks quite different. The process management facilities of a system are not aware of processes executing on a remote machine, and hence they have no authority over remotely executing processes. The identifiers of a process only have meaning for the operating system under which it is executing, making direct control of a process on a remote machine impossible. Since process control has to be initiated by a local process, the only alternative is to provide a mechanism for indirect control.

We need the ability to create processes without starting them, to set the monitoring strategy for processes, to signal processes to start, stop, and terminate, to detect process terminations, and to handle the standard I/O of a process. The system server *Rexec()* presents an example of a service that is used to create processes remotely. Since this is only a trivial subset of the services we must provide with the control process, *rexec()* was not used.

One design alternative is to use a mechanism resembling a remote procedure call. [Nelson 81] The implementation we would use would create a process on the remote machine for the purpose of servicing the call. When the call completes, a result is returned, and the process is destroyed. The advantage of this implementation is that a server exists only when it is needed. No system space is wasted to store the process state of an idle server.

We could use a remote procedure call as follows. When remote process management is needed, the control process could call a procedure on the remote machine that provides the functionality we need. This procedure, depending on the parameters, could create a process, create a filter process, set the metering flags for a process, stop a process, start a process, or remove a process. When creating a process, this procedure could make the needed modifications to the process's environment.

Although a remote procedure call is a powerful mechanism, providing us with the local processes we need in order to initiate process control, it does not provide us with a complete solution. Since process control is initiated by a local process, changes in process state have to be sensed by a local process. These changes in process state will occur at any time. In order to detect them, a process must exist for this purpose. A remote procedure call, as we have described it, does not provide a local process between requests, and therefore, we cannot use it.

To provide the process management facilities we need, we establish a server process on every host that supports the monitor. The addition of these servers is shown in Figure 5.1. These servers are *daemons*. They are processes executing independently from any terminal and any user, and whose services are always available. These servers are used by any control process to extend control into the realm of another system. A control process is a per user process; the servers are per system processes. With the assistance of the servers, the process management facilities available on the remote host can be used. These control servers have been called *meterdaemons*. The sole purpose of a meterdaemon is to serve control processes. A

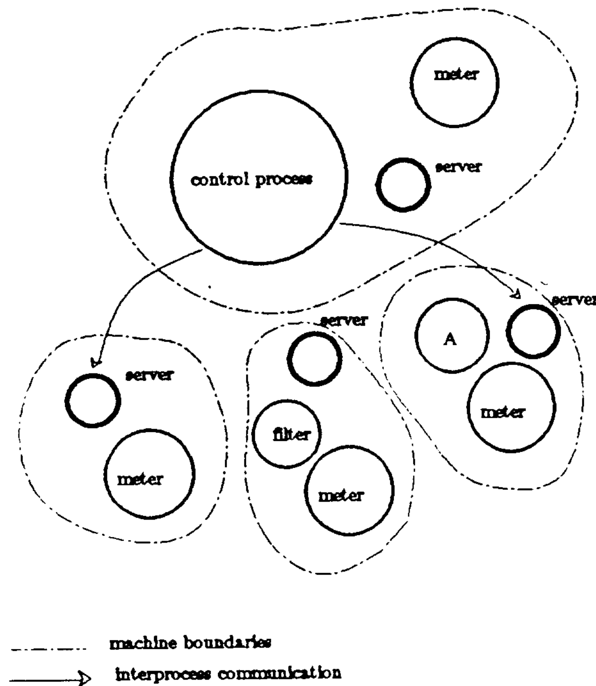

Figure 5.1 Role of Control Servers

meterdaemon is a slave of all control processes, spending most of its time waiting for requests. When it receives a request, it executes the appropriate functions and returns a reply to the requesting control process. A control process is not always the initiator of communications. When the termination of a process is signaled, the meterdaemon (not the control process) is the process which can detect the change in execution state. Therefore, the meterdaemon has to be able to initiate an interaction with the control process so that the control process can inform the user about the execution state of the processes in a computation.

A meterdaemon maintains information about the requests it receives. When it creates or acquires a process, it saves the process identifier and the location of the control process that issued the request. By keeping this information, the meterdaemon can handle requests from multiple control processes. Each request to the meterdaemon is independent from every other request. Information about the higher level structure of computations is irrelevant to the meterdaemon. The control process is responsible for maintaining this information.

As a list of requirements, our meterdaemon must:

- be available to accept requests from multiple control processes.
- provide the necessary functionality: process creation, process acquisition, filter creation, monitoring specifications, process control.
- maintain information about the processes that it creates or acquires for monitoring.
- be able to detect changes in the execution state of the processes it creates.
- be able to initiate an interaction with a control process.

The communications path between a control process (or *controller*) and a meterdaemon must be reliable. The lack of reliable datagram protocol in the current version of Berkeley UNIX forces us into using a stream connection for the controller/daemon communications. The

datagram protocol that is available is reliable if the processes communicating are executing on the same host, but it becomes unreliable when we cross machine boundaries. A disadvantage of stream protocols is that each stream connection uses a descriptor in the per process descriptor table. Since the size of this table is limited (the current size is 20 descriptors, three of which are used for stdin, stdout, and stderr), each stream connection between a control process and a meterdaemon would use a descriptor in each processes descriptor table. If the control process established a stream connection to each meterdaemon at the start of a monitoring session, the number of available descriptors would be quickly depleted. This is a limitation in flexibility that we would like to avoid.

The way in which we solved the reliability problem was to establish a temporary stream connection between a controller and a meterdaemon for each interaction. An interaction may be a request-reply pair from a control process to a meterdaemon or a one-way interaction from a meterdaemon to a control process. At the start of each interaction, a stream connection must be created. When the interaction has completed, the stream is destroyed. Although this increases the response time for each interaction, the frequency of these interactions in low enough that the cost in performance is not a significant deterrent.

5.2.2. Process I/O

Another issue involved in providing a complete control environment for the execution of a process is the redirection of a process's standard I/O across machine boundaries. This redirection should be done for any process the meterdaemon creates. For a process that is acquired for monitoring purposes, no changes are made to the handling of the process's I/O. This process continues executing in its existing environment even though it is now being monitored. This monitoring is transparent to the executing process. The user is not allowed to modify an acquired process's execution state or any aspect of its execution environment other than the monitoring strategy.

An example of redirecting process I/O is found in the implementation of the function *rexec()*. *Rexec()* is used to create a stream connection to a remotely executing command. The standard input and output of the command are redirected to this stream, allowing the user to communicate with the remote command. We use a similar strategy for our processes.

We have the ability to make a connection to a meterdaemon running on each machine. To complete the path from the process to the user, we must provide a facility for making a connection between the process being monitored and the meterdaemon. Given this facility, the standard output of the process is redirected to the meterdaemon, the meterdaemon forwards the message to the appropriate control process, and the control process displays the message to the user. The reverse path is traversed when sending standard input from the user to the process.

To implement this strategy, each meterdaemon creates an IPC socket. This socket is used as part of the bridge for forwarding I/O between the user and the process. It creates an additional IPC socket for each of its child processes. This socket is the child's end of the bridge. The child process, before executing the *exec()*, changes its standard input, standard output, and standard error file descriptors to the descriptor of the socket provided by the meterdaemon. Consequently, when the new process reads from or writes to standard I/O, it actually is reading from and writing to the meterdaemon.

Since the meterdaemon and the metered processes are executing on the same machine, datagrams can be used reliably for interprocess communication.

5.2.3. File Location

In order to create a process, the executable file must be accessible to the operating system on the machine where the process is being created. If we want a process to execute on a remote machine, but the file is local, either a remote file system must exist or we must copy the file to the remote machine. The lack of such a file system in 4.2BSD at this time forced us to implement the

latter alternative. The *rcp* utility was used to copy the files. This utility is accessed through a call to the *system()* function.

5.2.4. Internetwork Communication

The control process is responsible for making the required connections during a metering session. One example of a connection the control process must provide is the connection between a filter and a process that will be monitored. When a filter process is created, it opens an IPC socket and binds a name to the socket. This is the socket to which the meter messages will be sent. The control process must save the name of a filter's socket, and must give this name to the process which is to be monitored. Therefore, the name of an IPC socket will be exchanged between processes. These processes may be executing on different machines, and even on different networks.

A socket name is composed of a host address and a port number. A given host may be a member of two or more networks, and thus two or more different addresses may be used to access it by the other hosts in these networks. The implication of this inconsistency is that a socket name should not be exchanged between processes if this name will be used to make an IPC connection. Therefore, when communicating an address, the literal name of the host and the number of the port are exchanged. The receiving process then constructs the socket name using its own host address for the specified machine.

5.2.5. Protection

Protection in the measurement system is implemented according to the policy used in 4.2BSD. When using the measurement tool, a user is granted no special privileges; he or she has only those access rights dictated by his or her account. Access to any files involved in the creation of processes or in the storing of data are checked against the privileges associated with that account. An outside process, such as a system daemon, can only be metered by a user that has the appropriate access rights to such a process. To create a process on a machine, a user must have an account on that machine. This implies that, in order to meter a computation, a user must have an account on every machine involved in that computation.

5.2.6. Naming of Services

A meterdaemon and a control process are independent processes which must communicate with each other. A meterdaemon represents a service which is always available. A control process represents a client of the service. To use the service, the control process must establish a communication path to a meterdaemon. In order to do this, the control process has to know where (to which socket address) to send requests. This is the socket that was set up by the meterdaemon when it was created. Ideally, we would like to register the address of this socket with a name server, so that processes requiring the meterdaemon's service could locate the meterdaemon by asking the name server.

Berkeley UNIX does not yet provide a general, dynamic name server of this type. Limited name service is provided in the form of a static data base of services. This data base contains a list of services and the corresponding ports at which the services are located. The information in the data base is obtained through the system calls *getserverbyname()* and *getservent()*. The information about the meterdaemon service has been added to this data base.

6. Implementation of a Control Process

In the previous section meterdaemons were introduced. Figure 6.1 shows the structure of the monitor as it was augmented for Berkeley UNIX.

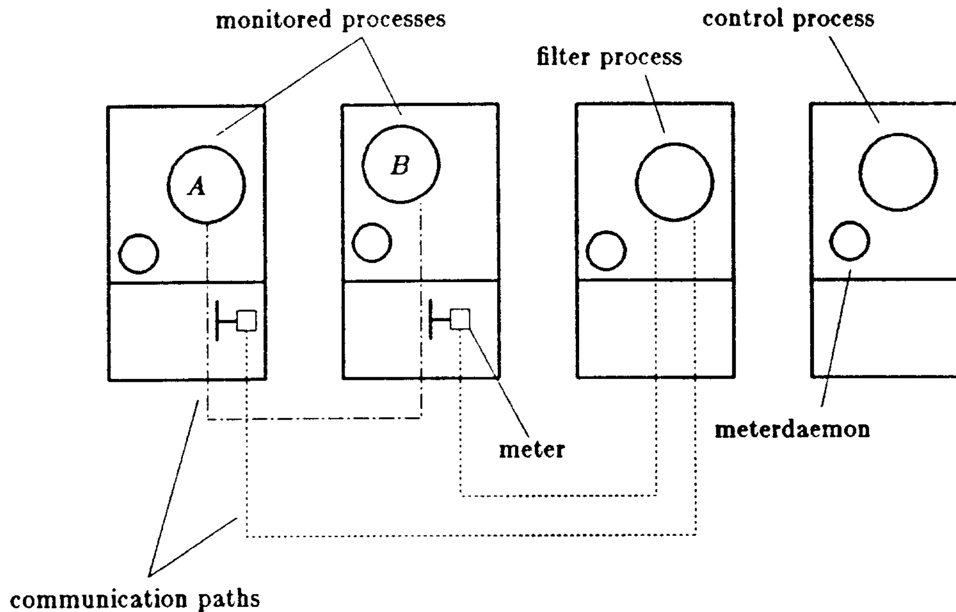

Figure 6.1 Augmented DPM Process Structure

Our "control process" is actually a per session control process, which we have called the controller, in conjunction with a team of meterdaemons. In this section, the implementation of the controller, the meterdaemons, and their communications protocol is described.

6.1. The Controller

6.1.1. Responsibilities

The controller is the front end to DPM. It is a command interpreter. The commands that are available are listed in Appendix A. During a user's session with DPM, the controller manages the state of the current computations and maintains a list of the currently available filter processes. A computation, or job, is created by assigning it a name, and then adding processes to it. Once the job name is assigned, the measurement strategy for the job can be specified. The controller keeps account of all of the processes created or acquired, and the name of the job to which they belong. The state of a job consists of a list of processes that are part of that computation, the address of the job's filter, and a list of the event flags that are set for the processes in the job.

Each process also has a state. The state of each process includes its process identifier, the name of the host on which it is executing, and its current state of execution. The process states maintained be the controller are *new*, *running*, *stopped*, *killed*, and *acquired*.

The processes making up a computation are tracked by the controller as they progress through these different stages of their lives. The state diagram representing a process's life cycle is shown in Figure 6.2. When a process has been newly created, it is in the *new* state. This state indicates that the execution environment has been set up, but the process is suspended prior to the execution of the first instruction. From the *new* state, a process can enter the *running* state by starting it, in which case the process begins execution. A process can also move from the new state to the *stopped* state, in which case it remains suspended, this transition occurring when the user stops the job. A process can switch between the running and the stopped states until the time it completes. When a process completes, it is moved from the running state to the *killed* state. It will remain in this state until the user removes the job. The killed state may also be entered from the stopped state. This occurs if the user decides to remove the job before it has completed execution. A process cannot be restarted once it has been killed. A process cannot move directly to the killed state from the new state. This restriction is enforced as a

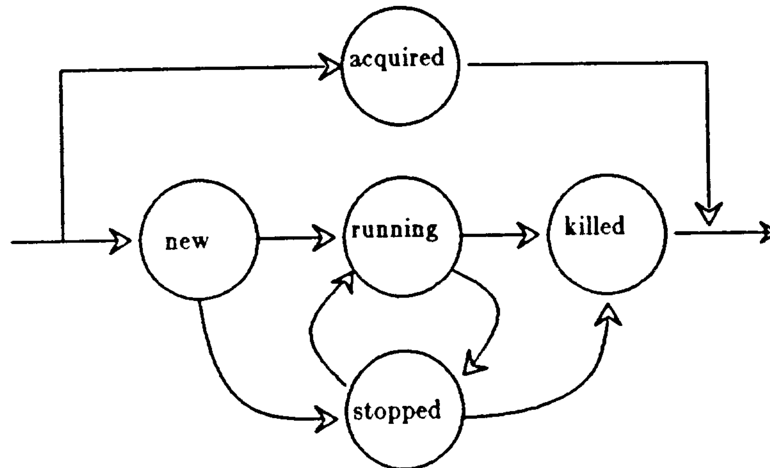

Figure 6.2 Process State Diagram

precautionary measure, ensuring that the user does not accidentally remove a computation that is in progress. If a previously existing process such as a system server is metered, it is moved directly to the *acquired* state. This is the only state such a process can be in. An acquired process cannot be stopped or killed, it can only be metered. The acquired status ensures this limitation of control.

As a command interpreter, the controller is always waiting for requests from the user. When a user request is received, the controller validates the command and the parameters. If the request can be handled locally (e.g., **HELP**, **NEWJOB**), the controller can satisfy the request without the assistance of a meterdaemon. If it is a request such as **ADDPROCESS** (process creation), the controller must use a meterdaemon. It initiates and establishes a communication path to the meterdaemon on the appropriate host, creates a request message indicating process creation, sends the request to the meterdaemon, and blocks waiting for the meterdaemon's reply. Upon receiving the reply, the communication path is closed and the controller takes actions based on the status of the reply. This activity may involve updating process and job records, and notifying the user.

The controller must also read messages coming from meterdaemons. These messages contain information about the current state of processes. Such information must be forwarded to the user and used to update the information about the status of the processes kept by the controller.

6.1.2. Communications

In order to participate in the necessary interaction with the user and with the meterdaemons, the controller must establish endpoints of communication. Requests from the user will be expected to arrive on the standard input of the controller, which is the terminal unless a file of control commands is being sourced (see Appendix A). To accept messages from meterdaemons, a special communications socket must be established.

The controller prompts the user and then uses the *select()* system call to block on a multi-way receive. In parallel, it waits for terminal input (for user requests) and for connection requests on the special socket. When one of these endpoints has something in its queue, the controller will unblock and can proceed to receive the message.

6.1.3. Data Structures

To represent the jobs and processes in the system, the data structures required are *job records* and *process records*. These records are use to store the state described in section 6.1.1, and their structures are pictured in Figure 6.3.

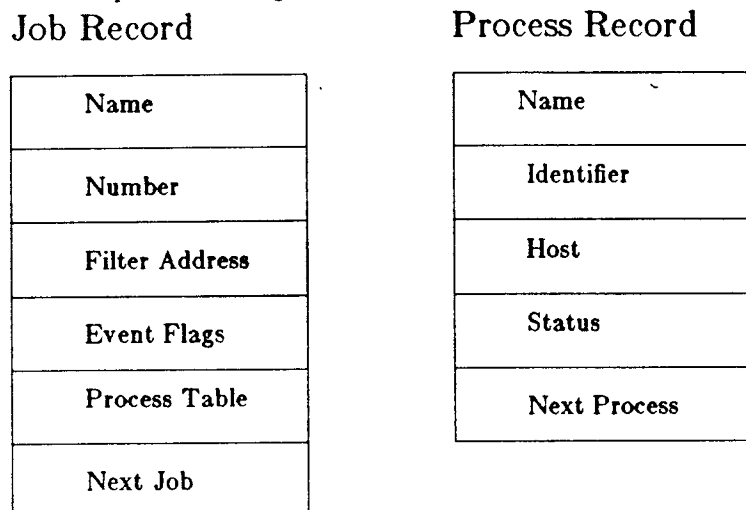

Figure 6.3 Job and Process Records

Information about filter processes is maintained using the job and process records. When a user uses DPM in a monitoring session, he or she enters a different command environment supported by the controller. When the controller initializes, it creates a job structure with the name *filter* to maintain information about filter processes. The process table pointer of this job record will point to the list of filter processes currently available for that user. To encompass the information needed for a filter process, a field was added to the process record. This augmented process record is shown in Figure 6.4.

The port field is used to store the *port* number. For all processes other than filter processes, the port number will be zero. For filters, it will be nonzero, and will be the number of the port used for the filter's end of a communication path.

Process Record

Name
Identifier
Host
Port
Status
Next Process

Figure 6.4 Process Record

The controller maintains the state of computations in a linked list structure of job and process records. An example of this state is shown in Figure 6.5. In this example, there are three jobs: the filter job and two computations. There are two filter processes, the *default* filter and one created by the user, *foo*. The computation *SORT* has two processes and the second computation, *DAEMON*, has only one process.

There are two other data structures that are part of the controller. A *command table* and a *flag table*. They are created when the controller is created. These tables are used by the command interpreter to internally translate and validate DPM commands and event flags.

6.1.4. Modules

The modules that implement the controller are written in the MODEL programming language [Morris 80]. MODEL is a strongly typed, high level language developed at Los Alamos Scientific Laboratory in 1973. It has PASCAL as its foundation. Pascal was extended to support the development of large application software and systems programming. MODEL allows for interfacing to existing libraries and subroutine packages, such as various C libraries. A characteristic feature of MODEL is its support of abstract data types.

The following are the main modules used to implement the controller:

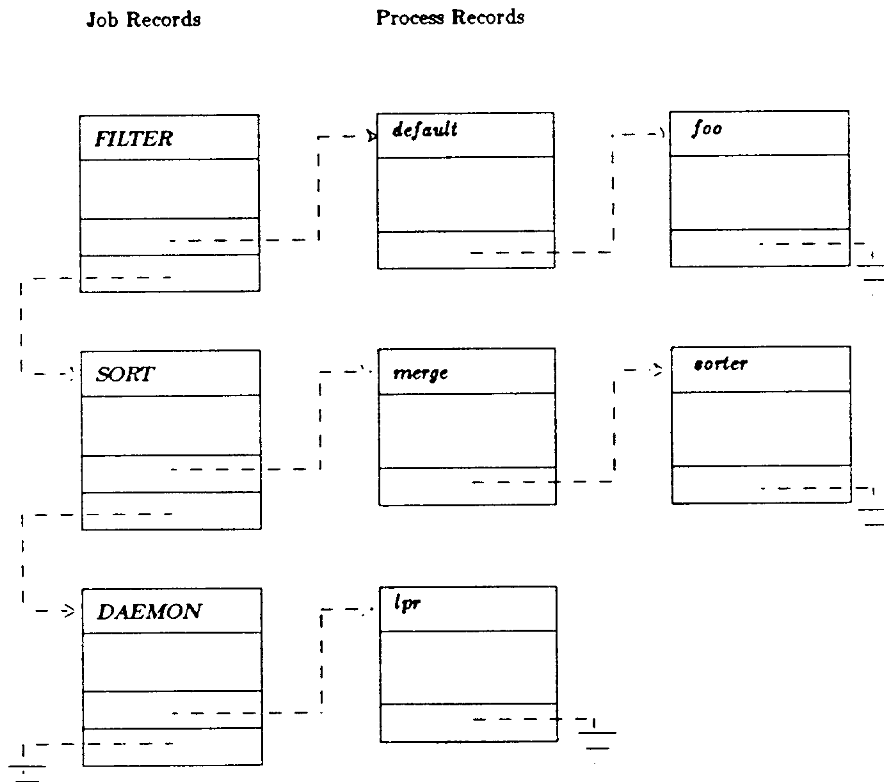

Figure 6.5 Current State of Computations

- control
- controlsubrs
- interfacesubrs
- controlscan
- controlstructs

Control is the main program. It contains part of the initialization of the command interpreter and the main loop. Initialization includes the creation of the command table and the flag table, creation of communication ports, jobtable initialization, filter job initialization, and creation of a default filter process. The main loop cycles through the three stages of command acceptance, command validation, and command switch.

The *controlsubrs* module contains the command procedures which are given control after the command switch. They take care of parameter validation and see to the execution of the specific functions.

The *interfacesubrs* implement the interface to the meterdaemons. They are called from the controlsubrs. They use the communication protocol for meterdaemon interactions. They establish the communications paths to the meterdaemon, package up a request message, receive the reply message, and then give control back to the calling controlsubr. Any necessary file transfers (system calls to *rcp()*) are executed at this level.

The reading and parsing of command lines received from the user is done through the *controlscan* routines. The module *controlstructs* contains all of the procedures that are used to manipulate the linked list structure of job and process records.

6.2. Meterdaemon

6.2.1. Responsibilities

A meterdaemon is a server whose clients are controllers. As a server, it must accept requests for services such as process creation, setting of a process's monitoring strategy, and signaling a process to start, to stop, or to terminate. For the processes it creates, the meterdaemon is responsible for handling their standard output, and for detecting changes in their execution state.

6.2.2. Communications

Client requests and process output are handled via a *select()* on the sockets where the meterdaemon may receive messages. A stream socket is used to listen for connection requests from a client controller. A datagram socket is used to receive the redirected standard output of the meterdaemon's children.

Changes in a process's execution state are detected via the *signal()* system call. When it creates a process, a meterdaemon uses *signal()* to enable the *CHILDSIG* signal so that an interrupt will be generated when a child of the meterdaemon changes state. This interrupt will force a transfer of control to a special signal handling routine. This is an asynchronous change in control. The signal handler is then responsible for determining which process caused the interrupt. If the interrupt was due to a process's termination, a termination report is created and sent to the controller that was responsible for the process's creation.

6.2.3. Data Structures

In order to report process terminations and redirect process output, the meterdaemon must maintain information about the processes it creates. It must know which controller requested each process's creation and the address of that controller's communication socket so that it can establish a connection to that controller. The meterdaemon must also know the address of the

datagram socket its child is using for its standard output. To do this, the meterdaemon keeps a linked list of process entries (a process log) that contains one entry per process. The process record maintained by a meterdaemon is pictured in Figure 6.6.

Process Record
pid
control host
control port
i/o port
next process

Figure 6.6 Process Log Record

Filter processes are included in the list of processes maintained by the meterdaemons.

6.2.4. Modules

The meterdaemon is written in the C programming language since the meterdaemon is a system server in the UNIX environment, and since the meterdaemon must interact heavily with the UNIX IPC. IPC uses many complex data structures that would be difficult to redefine in another language.

The main modules used to implement the meterdaemon are:

- meterdaemon
- daemonsubrs
- sighandler
- processlog

The *meterdaemon* module sets up the meterdaemon connections so that the meterdaemon listens on the predefined port (obtained from the services database via the *getservent()* system call for the *meter* service) and the meterdaemon main loop containing the function switch.

Daemonsubrs contains the function routines which take control upon the function switch. This is where the actual work requested by the user (e.g., process creation) is done.

The *sighandler* module is the routine which receives control upon an interrupt generated by a change in a child process's state. If necessary, the sighandler initiates a connection to a controller and sends a message.

Processlog contains all of the routines used to manipulate the linked list of process records.

6.3. Client/Server Interface

In this section we describe the message protocol used for the interactions between a controller and a meterdaemon. As stated in section 5.2.1, stream connections are used for each interaction. This is to provide us with a reliable path for communications. The connections are established as they are needed. Each interaction begins by setting up a communication path (*connect()* and *accept()*), and completes by closing the path.

The client's request resembles a procedure call: a *send()* followed by a blocking receive() . The server resembles a procedure being called: a blocking *receive()* followed by a nonblocking

send() of a reply to the client controller. This is a synchronous, two-way exchange of messages.

An example of the format of these requests and replies is shown in Figure 6.7.

length	type	body
bytes	1: create request	filename
		parameter count
		parameter list
		filter port
		filter host
		meter flags
		control port
		control host

a.) format of a request from a controller to a meterdaemon

bytes	8: create reply	pid
		status

b.) format of a reply from a meterdaemon to a controller

Figure 6.7 Example Formats of a Controller/Meterdaemon Message

This format includes the message *length*, the message *type* and the message *body*. The *length* field is needed so that, if two messages are waiting in the same queue, the message boundary can be detected. The *type* field identifies the purpose of the message. This may either be the type of request issued by the user or the type of reply returned by the meterdaemon. The remainder of the message, the *body*, is variable format, and its interpretation depends on the message type. All of the information that is required for the execution of the operation (in the case of a request) or for reporting the operation's result (in the case of a reply) is contained in the message body.

7. Experiment

7.1. Environment

Experiments with DPM were run on the Berkeley network, which includes several DEC VAX computers (VAX 11/780's and VAX 11/750's) connected by a 3 megabit Ethernet[Metcalfe & Boggs 76]. Each machine runs Berkeley UNIX 4.2BSD. The basic system supports process management, interprocess communication, memory management, and file management. Each machine has varying amounts of memory and peripherals.

One of the problems of the development environment in which we tested DPM is the lack of an adequate workload. To date, distributed programs are scarce creatures. In this section, we will present a brief overview of a distributed computation we were able to analyze. This is a parallel algorithm for solving the classic Traveling Salesman Problem (TSP) [Christofides 79].

7.2. A Study of The Traveling Salesman

7.2.1. Overview of the Problem and the Algorithm

Briefly stated, TSP is the problem of finding the Hamiltonian circuit of minimum cost in a complete graph $G = (V, E)$, where V are the vertices and E are the edges of the graph. Each edge in E is associated with a cost. Its name, the Traveling Salesman Problem, comes from the following problem context. If each vertex in V represents a city, and the cost of each edge represents the cost incurred if one travels between the two cities adjacent to the corresponding edge, the objective of the TSP is equivalent to that of a salesman who wants to find a tour of minimum cost such that he will visit each city exactly once.

Finding a solution to the Traveling Salesman Problem is a computationally intensive task. Yet, it is also a problem that can easily be decomposed into concurrently executing parts. The TSP solution we tested is similar to that of [Mohan 82]. The implementation was done by Nick Lai and is described in detail in [Lai & Miller 84]. The basic idea behind the algorithm is to divide the set of possible paths into a smaller set of paths and choose from this smaller set the path that appears to be part of the optimal path. If our choice is wrong, we backtrack and try again. For a given set of paths, the choice of the optimal path is independent of the choice for another set of paths. Therefore, we can have a number of processes concurrently computing the optimal choices for various sets of paths.

Organizationally, the computation involves a central control process and a variable number of server processes p, where $p \leq N$. The control process is the coordinator. It divides the computation up into pieces and gives a piece to each server. The main data structure of the implementation is an N X N cost matrix that is distributed to each server. A piece of the problem is represented by a subset of the cost matrix (or a subset of paths). The control process sends messages to its servers to tell them what piece of the matrix to work on. When they have finished their current assignment, the servers will send a message to the control process. The computation continues until the path of minimal cost is found.

7.2.2. Overview of the Analyses

The analyses of the TSP solution were performed by Bart Miller and Nick Lai and are discussed in detail in[Lai & Miller 84]. Their goal was to evaluate how well the computation performs at various levels of concurrency. The sizes of the problems analyzed were those corresponding to 8, 12, 16, and 20 vertices. For each solution the levels of concurrency analyzed were 4, 8, 12, and 16 server processes. The events monitored were message sends and message receives.

The analysis performed for each execution was the analysis that indicates the amount of parallelism, or speed up. [Miller 84] Briefly, what this analysis provides is an index **P** that expresses the amount of parallel execution that our computation achieved. **P** is computed as:

$$\mathbf{P} = \frac{T}{t_{max}}$$

Here T is the sum of process times for all of the processes in the computation, or *total time*, and t_{max} is the maximum value, taken over all of the processes in the computation, of the maximum possible execution time for a process. The maximum execution time for each process is determined from a directed graph constructed from the computation's trace. The way to interpret the value of **P** is as follows. Suppose we divide a computation into 5 pieces. The maximum value of **P** we could hope to achieve is **5**. This would mean that we succeeded in dividing our computation into five equal and independent, or perfectly synchronized, pieces. A **P** of **1** would tell us that our distributed program did no better than a program that executes sequentially; in other words, that we gained no parallelism. If **P** is less than **1**, we did poorly, as sequential execution would have been more efficient. This situation occurs if our communication costs outweight our gains from parallel execution.

The analysis of parallelism has parameters. These include the number of processes to assign to each machine, and an estimate of the network delay or communication cost. Luckily, from a given trace, we can obtain values of **P** for different assignments of processes to processors and for various values of network delay simply by changing the values of these parameters and rerunning the analysis. For example, **P** can be determined assuming that each process in the computation has its own processor (no contention) and that the network communications delay is zero (instantaneous communication). This analysis would give us the theoretical maximum for the amount of parallelism achievable by the program. For a more realistic value of **P**, we could specify two processes per processor and a positive value for network delay. The analysis routine will then take into account the delays due to processor sharing and to communications.

7.2.3. Results

The resulting values of **P** for various problem sizes and levels of concurrency are summarized in Figure 7.1.[†]

# of Processes		Matrix Size				
		4	**8**	**12**	**16**	**20**
4	max	1.4	2.1	2.4	2.5	2.6
	1/machine	0.3	0.8	1.3	1.6	1.9
	2/machine	0.3	0.7	1.0	1.2	1.3
8	max	———	2.7	3.6	3.8	3.9
	1/machine		1.1	2.0	2.6	3.0
	2/machine		1.0	1.6	1.9	2.1
12	max	———	———	3.7	4.1	4.5
	1/machine			2.2	3.0	3.6
	2/machine			1.8	2.3	2.7
16	max	———	———	———	4.4	5.0
	1/machine				3.2	4.1
	2/machine				2.5	3.1

Figure 7.1 Values of P for the TSP Problem

By changing the parameters for the analyses, three different values of **P** were computed for each run of the TSP program. These are: (1) the theoretical maximum value, assuming no CPU contention and communication costs equal to zero, (2) the value for one server process on each machine (N machines), still assuming no CPU contention but accounting for communication delay, and (3) the value for two server processes on each machine ($N/2$ machines). These results are summarized in graphical form in Figures 7.2, 7.3, and 7.4.

† Figures 7.1 through 7.4 are taken from [Miller 84], Chapter 5, pages 83-85.

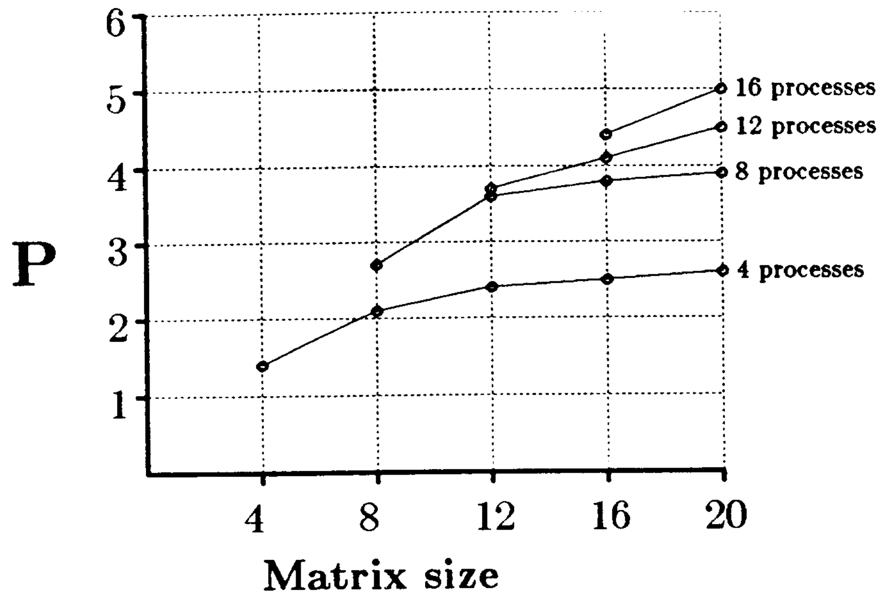

Figure 7.2 P for Maximum Parallelism

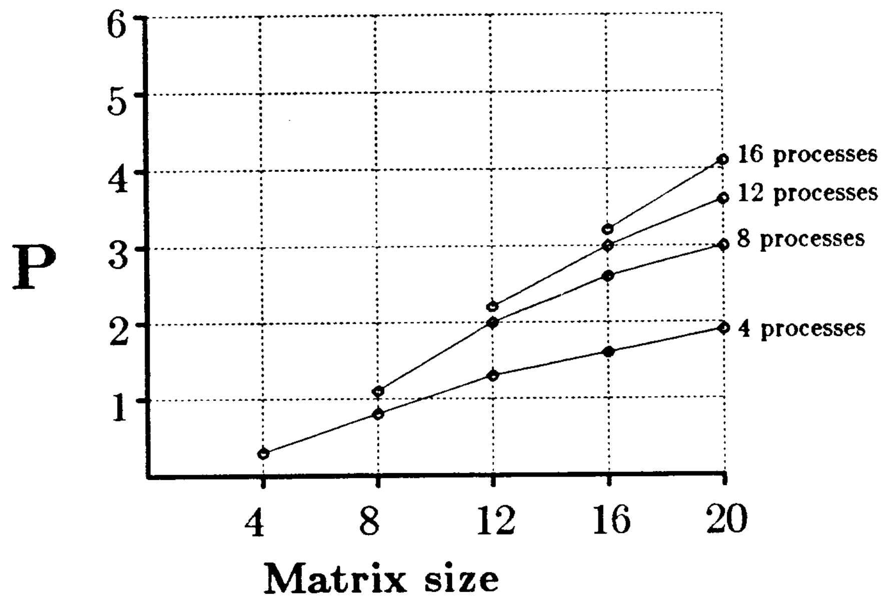

Figure 7.3 P for 1 process per machine

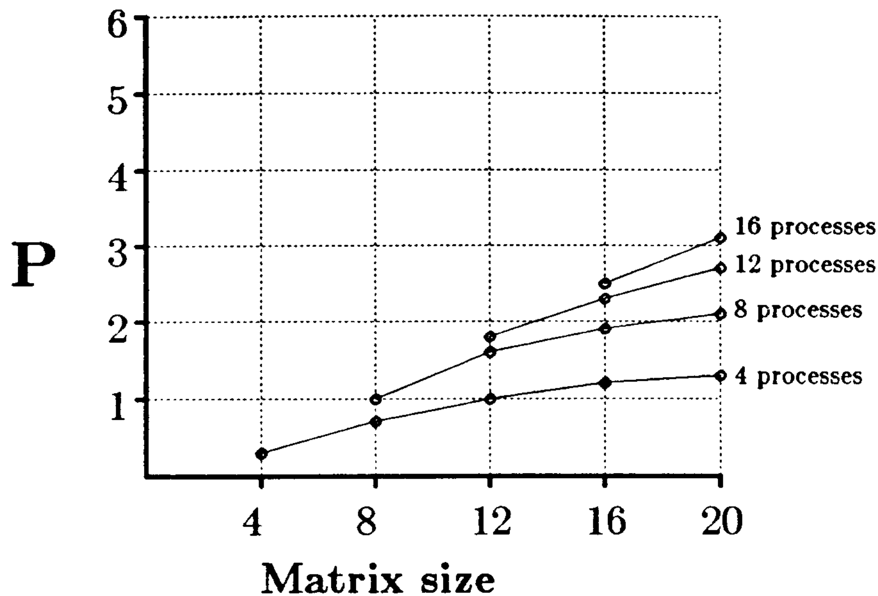

Figure 7.4 P for 2 processes per machine

We would first like to mention an interesting debugging result. Although DPM is explicitly not meant to be a debugger, debugging information is contained in the trace that DPM produces. For the initial runs, the TSP program had some subtle bugs. First, the program was creating only one server process even when it was intended to create many. This problem was immediately evident from the trace produced when program was monitored. When we fixed this bug, the next trace revealed a second bug. This time we discovered that even though the correct number of servers were being created, only one was being loaded by the control process. The remainder of the servers were always idle. With these bugs fixed, the performance analyses could proceed.

From the above graphs, we can infer the following about the behavior of our program. From Figure 7.2 we can deduce that, given no contention for CPUs and zero communications costs, for a given problem size, we always gain parallelism by increasing the number of server processes. For a problem of size 16, with 4 servers a **P** of 2.4 was obtained. Increasing the number of servers to 16, we observe and increase in **P** of 2 units, making it 4.4. This larger value of **P** indicates success in speeding up the program. It does not, however, include consideration of the cost of the additional servers.

We also deduce from Figure 7.2 that, as the size of our problem increases, we benefit more from distributing our program. Looking at the **P** curve with the number of servers kept constant at 4, we observe a value of **P** of only 1.3 for a problem of size 4, but of 2.6 for a problem of size 20.

Moving to Figure 7.3 where we have introduced communication costs, we see that we do not always increase the performance of a program by distributing it. In fact, with 4 server processes, we actually do worse than sequential execution for small problems. The reason for this is that, in a small problem, there is not enough computation to be done to make distributing the computation convenient, and communication delays dominate the time the servers spend computing. As the problems become larger, the amount of CPU time needed by the servers becomes larger with respect to the amount of time lost due to communication delays. Hence, the performance gained through parallelism warrants program distribution.

When we also consider CPU contention, as in Figure 7.4, the amount of parallelism achieved decreases further. This is the behavior we would expect in practice, since a process no longer has immediate access to a processor.

The above data represents analyses run on the initial implementation of the algorithm. From interpreting this data, insight into the programs behavior helped to find areas where the program could be optimized. These algorithm optimizations and their analyses are described in [Lai & Miller 84].

8. Conclusions

8.1. Analysis of the Monitor's Implementation

How useful is DPM? The experiment of the previous section, in addition to others performed by Miller [Miller 84], has shown that DPM is a useful tool. It has been productive as a performance measurement instrument and even as a primitive debugger.

Another important question is: how well did we do with respect to attaining our goals of consistency and transparency? A proof that consistency was achieved is the situation that occurred in the TSP experiment. Recall that simply by looking at the trace, certain program bugs became evident. If the trace contained data that was not consistent with the user's view of a computation, bugs would not be as apparent. A user would first have to decipher how the trace messages related to the primitive functions that he had coded. Transparency of the mechanism was also achieved. By putting the event detection mechanism in the kernel, a program does not have to be recompiled in order to be monitored. Any process can be monitored, even one that already exists. Of course, perfect performance transparency was not achieved. Every time someone makes a system call that may have to be monitored, it must be checked for monitoring, even if no monitoring is being done. This introduces an extra subroutine call and bit check into the execution of these system calls. Furthermore, we pay a price in terms of space. First, we pay for the three extra fields that were added to the process table entry, and second for additional code in the kernel. One should realize that, no matter how the monitoring is implemented, some price will have to be paid for the privilege of monitoring.

8.2. More on Evolution

Research and development on Berkeley UNIX continues. Efforts towards a remote file system are in progress. The eventual realization of this facility will simplify the task of the control process. With a remote file system, the need to copy files to their intended locations will disappear. A remote file will be accessible in the same way as a local one.

Moving closer toward a distributed system and moving further into the future, we can imagine distributed process control in Berkeley UNIX. If this should happen, the structure of the control process could be greatly simplified. The need for meterdaemons would disappear. All process management could be handled be a single process.

In addition to continuing work on the Berkeley UNIX DPM, the idea for a third implementation of the monitor design has been conceived. This implementation is for the Charlotte operating system [Finkel 84]. Charlotte is a distributed operating system developed at the University of Wisconsin in Madison. It was designed for Crystal, a loosely coupled multicomputer network[DeWitt 84]. Charlotte was developed with the goal to provide an environment in which computationally intensive distributed programs could be solved. It was intended to be a test bed for the design and implementation of multi-process applications and distributed algorithms.

Charlotte provides another new and interesting environment for DPM.

8.3. Future Research

The past history and the current status of the Berkeley UNIX monitor are summarized in the following table:

DEMOS/MP implementation completed	September 1983
Kernel changes to 4.2BSD working	April 1984
Control process working	May 1984
Debugging, refixing	ongoing
Analysis routines (parallelism, communication statistics, control flow)	May 1984
New analysis techniques and automation of analysis procedures	ongoing

As of May 1984, the Berkeley UNIX DPM was being used for the TSP study. Polishing and improving DPM continues. User experience is important for such improvements but is something we have a shortage of. When more user experience is gained, suggestions for a better user interface will be used to polish the tool. An obvious improvement is the automation of the measurement-to-analysis step. Instead of having to retrieve the program trace and subsequently run analyses, a user should be able to have numbers and graphs displayed by issuing a simple command.

More interesting and promising is the potential for new and different uses of DPM. Real time monitoring is one idea. The analysis could be performed and presented to a user as the computation proceeds, thus presenting a dynamic view of a computation. Reaching further, we could have the information from real time analysis fed back into the system. Such information might be used as an input to the system's scheduler to help with scheduling decisions, or as an input to the process manager to help implement load balancing policies.

Another promising area is the area of distributed debugging. Process interactions must be accounted for by a debugger for distributed programs. A monitor such as DPM could be used in conjunction with conventional debuggers, the latter for debugging the individual processes and the former for debugging the process interactions. Last but not least, there is always a need for new analyses.

In summary, DPM appears to be a useful and powerful tool with a promising future.

9. References

[Finkel 84]
Y. Artsy, H. Chang, and R. A. Finkel, "Charlotte: Design and Implementation of a Distributed Kernel," Technical Report TR-554, University of Wisconsin (August 1984).

[Baskett *et al* 77]
F. Baskett, J. H. Howard, and J. T. Montague, "Task Communcations in DEMOS," *Proc. of the Sixth Symp. on Operating Sys. Principles*, Purdue, (November 1977).

[Christofides 79]
N. Christofides, "The Traveling Salesman Problem: The Development of a Distributed Computation," pp. 131-149 in *Combinatorial Optimization*, ed. Christofides, Miugozzi, Topf, and Saudi, Wiley (1979).

[DeWitt 84]
D. J. DeWitt, R. Finkel, and M. Solomon, "The CRYSTAL Multicomputer: Design and Implementation Experience," Technical Report TR-553, University of Wisconsin (September 1984).

[Ferrari 78]
D. Ferrari, *Computer Systems Performance Evaluation*, Prentice-Hall, Englewood Cliffs, NJ (1978).

[Gusella & Zatti 83]
Riccardo Gusella and Stefano Zatti, "TEMPO: A Network Time Controller for a Distributed Berkeley UNIX System," Technical Report UCB/CSD 83/163, University of California, Berkeley (December 1983).

[Lai & Miller 84]
N. Lai and B. P. Miller, "The Traveling Salesman Problem:," Technical Report UCB/CSD, University of California, Berkeley (June 1984).

[Leffler 83]
S. Leffler, "A 4.2BSD Interprocess Communications Primer," Computer Systems Research Group Technical Report, University of California, Berkeley (1983).

[McDaniel 75]
G. McDaniel, "METRIC: a kernel instrumentation system for distributed environments," *Proc. of the Sixth Symp. on Operating Sys. Principles,* pp. 93-99 Purdue University, (November 1975).

[Metcalfe & Boggs 76]
R. M. Metcalfe and D. R. Boggs, "Ethernet: distributed packet switching for local computer networks," *Comm. of the ACM* **19**(7) pp. 395-404 (July 1976).

[Miller 84]
B. P. Miller, "Performance Characterization of Distributed Programs," Ph.D. Dissertation, University of California, Berkeley (May 1984).

[Miller *et al* 84]
B. P. Miller, S. Sechrest, and C. Macrander, "A Distributed Programs Monitor for Berkeley UNIX," Technical Report UCB/CSD, University of California, Berkeley (September 1984).

[Mohan 82]
J. Mohan, "A Study in Parallel Computation – the Traveling Salesman Problem," Technical Report CMU-CS-82-136, Carnegie-Mellon University (August 1982).

[Morris 80]
J. B. Morris, *A Manual for the Model Programming Language,* Los Alamos Scientific Laboratory, Los Alamos, New Mexico (February 1980).

[Nelson 81]
B. J. Nelson, "Remote Procedure Call," Technical Report CSL-81-9, Xerox Palo Alto Research Center (1981).

[Powell 77]
M. L. Powell, "The DEMOS File System," *Proc. of the Sixth Symp. on Operating Sys. Principles,* pp. 33-42 Purdue University, (November 1977).

[Powell & Miller 83]
M. L. Powell and B. P. Miller, "Process Migration in DEMOS/MP," *Proc. of the Ninth Symp. on Operating Sys. Principles,* pp. 110-119 Bretton Woods, N.H., (October 1983).

[Sechrest 84]
S. Sechrest, "Examples of Interprocess Communication in UNIX 4.2BSD," Technical Report UCB/CSD, University of California, Berkeley (June 1984).

Appendix A: User's Manual

This appendix describes the details of the user's commands to the control program. The commands are: **help, filter, newjob, addprocess, acquire, setflags, startjob, stopjob, removejob, removeprocess, jobs, getlog, source, sink**, and **die**. In the command descriptions to follow, all parameters will be surrounded by the symbols '<' and '>'. Optional parameters will be enclosed by '[' and ']'. Command parameters must be literals formed from the digits 0 through 9, the upper and lower case letters, and the characters '/' and '.'. Delimiting characters are blanks, tabs, new lines, and carriage returns. An example of a session with the controller is presented in Section 4.4.

help

This command is a source of information. It displays the menu of the control commands with a brief statement of their function. The required and optional parameters to the command are indicated. At the end of the menu, the available metering flags are listed.

filter [<filtername>[<machine>[<filterfile>[<descriptions>[<templates>]]]]]

If parameters have been specified, a filter process will be created from the program in "filterfile" and will be given the name "filtername" for identification within the control program. The filterfile must be an existing, executable file. If no filterfile has been specified, the default file "filter" is used. If no machine has been specified (only the filtername has been specified on the command line) then the default is to create the filter locally. A standard filter requires two additional files: the trace descriptions file and the templates selection rules. If these files are not specified on the command line, standard filenames ("templates" and "descriptions") are used. Read access is required to these files.

If no parameters have been specified for the filter command, information about existing filter processes is displayed. The information is a list of the available filters indicating their process identifiers, their names, and the machines on which they are executing.

newjob <jobname> [<filtername>]

This command is used to create and initialize a job. The job is given the name "jobname". The user can optionally specify a filter for the job by including the name of an existing filter process. If no filter is indicated, the control program uses the default filter process. A job cannot be created if a filter has not been created. This implies that prior to the execution of **newjob**, a **filter** command must have been executed.

No restriction is placed on the number of jobs or on the number of filters the user can create. Many computations could be executing simultaneously, having traces collected by different filters.

addprocess <jobname> <machine> <processfile> [<parm1 parm2 ...>]

Addprocess, or **add**, is used to create a process as part of a job. The process is added to the job "jobname" and is created on the host "machine", by executing the file "processfile" with the parameters "parm1, parm2, ...". A process does not begin executing at this time, and its process state is new. The process is connected to jobname's filter and inherits the flags of job "jobname". A process can be added to any job at any time, even after processes in the job have started execution.

acquire <jobname> <machine> <process identifier>

The **acquire** command provides the user with the ability to meter a process that is already executing. In this case, the process identifier must be known and specified as "process identifier". To meter this process, the user must have the appropriate access rights. "Jobname" indicates the job to which the acquired process will be assigned. This

assignment determines the filter to which the event traces will be directed. "Machine" is the host on which the process is executing.

The motivation for this command is that situations may arise in which a process such as a system server is an important component of a computation. Thus, it should be metered along with the other processes involved in the computation. Even more simply, a user may be interested only in monitoring a system server to better understand its behavior.

setflags <jobname> <flag1 flag2 ...>

This command is used to set, or reset, the metering flags on a job. **Setflags** is the user's interface to the 4.2BSD UNIX system call *setmeter()* described in section 4.1. The event flags are:

fork	process creation
send	sending a message
receivecall	ready to receive a message
receive	receipt of a message
socket	creation of a communication socket
accept	request for a connection (readyness to accept)
connect	establishment of a connection

The effect of **setflags** is to record the flag set "flag1 flag2 ..." with "jobname" and then set the flags for each process which is part of jobname. Flags can be set at any point in the execution of a job. A process does not have to be stopped in order to set its flags.

Flags are set by listing the desired flags on the command line. They can be reset by indicating a '–' in front of the flag to be reset. For instance, "–send" will turn off the metering of the *send* event for the job indicated. A shorthand notation can be used. The flag 'all' will set all of the metering flags, and '–all' will reset all of the flags.

If two **setflags** commands are executed, the set of active flags is the union of the two groups of flags. A **setflags** command does not implicitly negate the previous flag settings; all resetting must be explicit.

jobs [<jobname1 jobname2 ...>]

Jobs provides the user with information about the state of a computation. It does not change the state of the system in any way. If no parameters are specified, a list of the current jobs is displayed. This list indicates the number, the name, and the filter for each job.

If a list of jobnames is specified, then, for each job in the list, information about the job's processes will be displayed. For each process, such information includes the process identifier, the current control state, the process name, the machine on which the process is executing, and the meter flags that are set.

startjob <jobname>

This command initiates the execution of a job. "Jobname" specifies the job to be started. All processes in the new or stopped state are signaled to begin or resume execution. Processes that are running, killed, or acquired cannot be started. The user is informed as to the status of each process.

stopjob <jobname>

This command is used to halt the execution of a job. "Jobname" specifies the job to be stopped. All processes in the specified job that are in the new or running state are signaled to halt execution. Processes that are killed or acquired are ignored.

removejob <jobname>

This command is used to remove the record of a job from the controller. Jobname specifies the job to be removed. A job can only be removed if all of its processes are in one of the states killed, stopped, or acquired. If any of the processes are still running or new, the job will not be removed. When an acquired process is removed, the control program insures that the filter connection of that process is taken down so that the process will not continue to be metered after its job has been removed, but the process continues to execute.

getlog <filtername> <destination filename>

The log file for the filter process identified by "filtername" is retrieved. When retrieved, the log file is copied to the file "destination filename".

source <filename>

This command (and the **sink** command) allows the user to run scripts of control commands. **Source** instructs the control program to read command lines from the command script "filename" instead of from the terminal. Any of the control commands may be specified in this script file. Source commands may be nested within scripts to a maximum depth of sixteen. When the entire script has been processed, input is again expected from the terminal.

sink [<filename>]

Sink provides a way for the output of commands to be written to a file instead of to the terminal. Thus, the output of a sourced script can be redirected to the file "filename" by having a "sink <filename>" command as the first line of the script. When the script is completed, the output file should be changed back to the terminal by including a **sink** command without parameters as the last command in the script; output is directed back to the terminal when a destination filename is not specified.

die

This command terminates the control program. Upon exit, all executing filter processes are removed. Aliases of this command are **exit, bye,** and control D (**^D**). If there are still active processes (new, stopped, running, or acquired), the user is warned, and the controller does not exit. If the user immediately repeats the **die** command in this situation, the controller will assume the user is aware of the situation and exits with the processes active.