sequential while that was highly parallelized (it can be shown that the number of *time units* used by the previous algorithm is $O(N \log d)$ while that for this algorithm is $O(Nd)$). Several variants of these algorithms can be easily constructed by noting that we have given algorithms at two extremes of the spectrum.

Fault tolerance is an aspect we have not addressed here. Observe that it is not trivial to define the sorting problem when some sites could fail at any time. In fact, since the topology considered is a straight line, the network can get partitioned due to a single edge/site failure making it impossible to communicate a message from a site to all other sites. Our algorithms do not work in such cases. In fact, we know of no work on fault-tolerant sorting on any topology.

## References

[1] F. Chin and H. F. Ting, "A near-optimal algorithm for finding the median distributively," in *Proc. 5th Int. Conf. Distributed Comput. Syst.*, May 1985, pp. 459–465.

[2] G. N. Frederickson, "Tradeoffs for selection in distributed networks," in *Proc. 2nd ACM PODC*, Aug. 1983, pp. 154–160.

[3] M. C. Loui, "The complexity of sorting on distributed systems," *Inform. Contr.*, vol. 60, no. 1-3, pp. 70–85.

[4] M. Rodeh, "Finding the median distributively," *JCSS*, pp. 162–166, 1982.

[5] R. Rom, "Ordering subscribers on cable networks," *ACM TOCS*, vol. 2, pp. 322–334, Nov. 1984.

[6] D. Rotem, N. Santoro, and J. B. Sydney, "Distributed sorting," *IEEE Trans. Comput.*, vol. C-34, pp. 372–376, Apr. 1985.

[7] ——, "A shout-echo selection in distributed files," *Networks*, to be published.

[8] S. Zaks, "Optimal distributed algorithms for sorting and ranking," *IEEE Trans. Comput.*, vol. C-34, pp. 376–379, Apr. 1985.

# DPM: A Measurement System for Distributed Programs

BARTON P. MILLER

*Abstract*—**DPM is a system for monitoring the execution and performance of distributed programs. An important characteristic of its design is the simplicity of each part of the design. This simplicity has resulted in a system of tools that has a wide range of applications and that was relatively easy to construct. We start with a simple framework for distributed computation based on message interactions. We use this framework to develop a structure for a measurement tool for distributed programs, implemented for both the DEMOS/MP and Berkeley UNIX operating systems.**

**DPM can measure communication statistics, dynamic program structure, and parallelism. It can be used for post mortem analysis of a program's performance, real-time performance monitoring, and generating data to be used by the operating system for such things as a scheduler for load balancing.**

*Index Terms*—**DEMOS, distributed program, monitor, performance evaluation.**

Manuscript received October 15, 1985. This work was supported by National Science Foundation, Grant MCS-8010686, the State of California MICRO program, and the Defense Advance Research Projects Agency (DoD) Arpa Order 4031 monitored by Naval Electronic System Command under Contract N00039-82-C-0235.

The author is with the Department of Computer Sciences, University of Wisconsin-Madison, Madison, WI 53706.

IEEE Log Number 8715440.

## I. Introduction

This paper presents a framework for measuring the performance of distributed programs. This framework includes a model of distributed programs, a description of the measurement principles and methods, and a guideline for implementing these ideas. We have constructed a measurement system (called the Distributed Programs Monitor, or DPM) based on these concepts. DPM has been implemented and used for measurement studies on two different operatings systems (DEMOS/MP [1], [2] and Berkeley UNIX [3]).

Collecting data about a program's performance is not enough; we must supply some form of interpretation or analysis of the data. We include, as part of DPM, several analysis techniques that can provide information about the structure, the amount of parallelism, and the communications patterns of a distributed program. DPM is more than a particular implementation of a measurement facility. It provides a framework for other activities that are based on the monitoring of a distributed program. Some of these activities include real-time monitoring and display of the activities of a program, and use of the measurement data for feedback scheduling activities such as load balancing.

### A. Overview

The driving principle in the design of DPM is simplicity. The model of distributed computation is simple in the sense that it is general enough to make it applicable to a wide range of systems. Our methods of measurement are simple to ensure easy implementation. The implementation of our tools is simply structured to provide confidence in their correctness.

The goal of simplicity has produced a subordinate goal, *transparency*. The goal of transparency enforces simplicity of use for the programmer. To measure a program we should not have to recompile, relink, or write in a special style or language. We should not have to supply special information to the measurement system to have it function correctly.[1] Transparency also means that the performance of the program being monitored is not significantly disturbed. A monitor built in software will always have some affect on a program's performance, but our design goal is to minimize this effect. This goal will influence both the design and the implementation of the measurement system.

Our measurements are done passively, as opposed to systems that interact with the computations—such as happens with interactive debuggers. By this, we mean that actions such as redirection of messages, breakpoints, and modifications of the message streams are not allowed. DPM is an observer of the computation, and not a participant.

### B. What is a Distributed Program? And Other Definitions

Our model of distributed programs provides the guidelines for the design of DPM. It is not a formal model in that we do not use it as the basis for mathematical analysis; rather, the model can be considered as a reference point for the design and implementation of the measurement system.

We define a *distributed program* to be a collection of processes cooperating to perform some computation. The component processes are not constrained to run on the same machine. No assumptions are made about the locations of the processes. A distributed program (more simply called a *computation*) is made up of *processes* that are the basic building blocks of a computation. A process consists of an address space containing code and data, and an execution stream. Each process has access only to its own address space. Processes do two things: compute and communicate. Computing is the normal execution of instructions and does not affect the state of other processes. These instructions are referred to as *internal events*. Communication is the means by which a process will interact with

[1] Note that we say "have to." The option is still available to augment the measurement system with, e.g., compiler supplied information.

other processes and the operating system. Interactions are referred to as *external events*. The complexities of the distributed environment become apparent when a process in a computation interacts with another part of the computation.

Communication is based on messages. A message allows the copying of part of one process's address space into that of another process. A message is an interaction involving exactly two processes: the process originating the data (the *sender*) and the process consuming the data (the *receiver*). We make no restrictions on the structure of the message delivery. The communications path may be unidirectional or bidirectional. The message passing operations may be synchronous or asynchronous. Message delivery may or may not be guaranteed or required to preserve message order. Message paths may be dynamically or statically created and destroyed, and the processes in the computation may be dynamically created and destroyed. We make no assumptions about the network or facility underlying the communications mechanism. Our model of computation applies to a wide range of systems because of its simplicity.

Our model of computation does not include systems that have processes with shared address spaces. Conceptually, a shared memory system can be modeled as a message-based system (and vice versa) [4]. In practice, the interactions in a message-based system are generally easier to detect than in a shared memory system, and therefore easier to monitor.

Processes execute on *machines* that do not have direct access to each other's memories. Each machine has a portion of the operating system running on it to support process execution, communications, memory management, and device management. The communication functions supplied by the operating system provide for interprocess communications both within and between machines.

## II. The Measurement Design

Our measurement design follows the basic philosophy of "look, but don't touch" with respect to the program being studied. The goal is minimal disturbance of the execution of the program. This means that the computation being measured should not execute more slowly or achieve different results because it is being measured. If the cost of measurement is high, then the act of measuring a computation could substantially change its execution behavior. These guidelines have determined the design of our measurement system. This section provides an overview of our design.

Fig. 1 gives an overview of event detection and our measurement system. Internal events are not visible from outside a process and are therefore not detected. The detection of external events is referred to as *metering*. A trace is produced for each event that is detected. After the trace is produced, a decision is made whether or not to keep the trace. The selection decision is called *filtering*. If the trace is kept, it is stored until it is processed to provide results that may be used to understand the behavior of the process (and the overall computation). We call the processing of the traces *analysis*.

The metering stage of measurement lies within the kernel of the operating system because of the desire not to change the program itself. The facility should be simple, to make the necessary modifications as simple as possible. Changes to an operating system kernal are typically much more difficult than those to parts of the system outside the kernel.[2] Alternatively, the event detection could be placed in the language run time library, compiler-generated code, or could be inserted directly by the programmer. While these methods may be simpler to implement, they provide for less generality and less transparency. For example, these alternatives might require the programmer to use a particular language or might allocate an available file descriptor.

The filtering stage provides for a flexible set of rules to perform

[2] In our experience, the effort necessary (including design, coding, and testing) to put a given function in the operating system kernel is about 10 times greater than implementing the same function outside the kernel (in a process). A similar statement can be made when moving kernel functions into microcode.

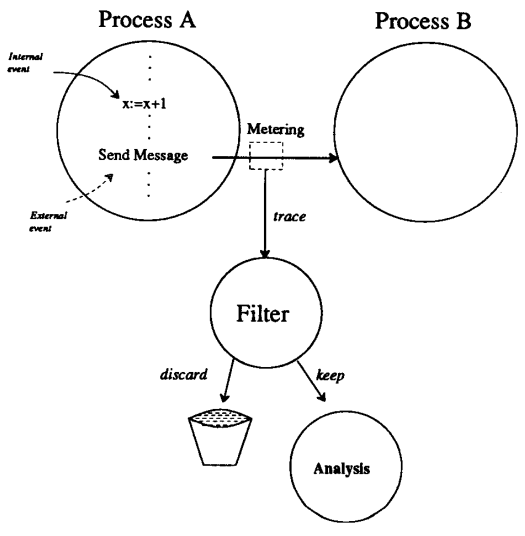

Fig. 1. Events and the measurement model.

data reduction. This facility allows easy change to the selection criteria and easy adaptation to new or changed trace types.

The analysis of the data provides a summary of execution of the computation. It is at the analysis stage that useful information is provided about the computation. The goal of the measurements dictate the type of analysis being performed and the overall structure of the measurement system (see Section IV).

## III. The Measurement Facility

The measurement facility is described by the events that we measure and the structure of the measurement tools. The measurement tools consist of the previously mentioned components (meter, filter, analysis) and user interface. We describe the events, meter, filter, and user interface in this section. The analyses are described in Section IV.

### A. Events and Trace Records

There is a set of meter events that reflect the basic operations as seen by the programmer of a distributed computation. The structure of the metering stage is very simple due to the small set of meter events (currently 10). These event types are, for the most part, the same across the different operating systems supporting the measurement facility. These events consist primarily of activities (such as the sending and receiving of a message) that reflect interactions between processes. Other events related to communications are also recorded. This group of events consists of actions that effect the creation, modification, and destruction of communications paths. The last group of recorded events pertains to the state of the processes in the computation. The basic events are the creation of a process, the starting and stopping of its execution, and the destruction (termination) of the process. Depending on the system from which the measurements are being extracted, there may be slight variations in the details of the data collected with each event type.

METRIC [5] allowed the users to specify their own event trace types. It would be easy to add such a mechanism to DPM. Only the meter would need to be changed and these changes would be minor. We chose to not provide this facility for reasons of transparency. User-defined traces would require explicit use of the trace facility within the processes being measured.

Included with each event trace is a standard header describing the trace. The header of an event trace contains the following fields: MACHINEID (machine from which the trace came), PROCTIME (amount of CPU time used by this process up to the time this trace was generated), TIME (wall clock time as known by this machine), TRACETYPE (type of event described by this trace), PC (program counter indicating the location in the process causing the event), and LOADAVERAGE (current number of runable processes on this machine).

The event trace types are: SEND, RECEIVECALL, RECEIVE, MESSAGEQUEUED, CREATEPATH, DESTROYPATH, CREATEPROCESS, STARTPROCESS, STOPPROCESS, and DESTROYPROCESS.

The meter traces do not include the contents of the messages sent by the users. The meter traces record only the occurrence of an event and information about which processes were involved. This results in less trace data to communicate and store.

Even though we do not include the message contents in the traces, we can still derive information about specific activities within the processes that are being measured. By using the PC information in the trace header we can identify the specific procedure within the process that caused the event. These are the same techniques used by high-level language debugger to allow symbolic reference to program procedures and variables.

### B. Metering

Metering is the activity that takes place within the operating system kernel to extract the events of interest. This portion of the measurement facility is the only change required to the supporting operating system.

The point in the operating system kernel where the necessary information is available must be located to meter the specified events. At these points, we insert *meter probes* into the code of the kernel. These probes are procedure calls to a software module (*meter module*) that is responsible for passing the traces to the filter. The parameters for the specific trace being generated are passed to the meter module with the procedure call. These values, along with the standard format header, are passed to the filter. A communications path, called the *meter path*, is used by the metering routines for sending traces to the filter stage.

We try to minimize the performance overhead of generating the trace messages. Two mechanisms help in this effort. The first is the buffering of the trace messages. The major cost in generating the traces is that of sending the message over the meter connection. We typically buffer up to 50 traces (for a given process) before sending a trace message. The second mechanism that contributes to the performance is that the meter module can access the communications routines with substantially less overhead than could a process.

### C. Filtering

Filtering is the data selection and reduction stage in the measurement system. It reduces both the size and number of traces as they are produced. Data are received from the metering stage, filtered, and then passed on to the analysis stage or stored for later analysis. The scheme currently used in the measurement system is based on a general, one-level, pattern matching algorithm. More sophisticated schemes (such as presented in [6]) may be used in subsequent versions of DPM. This filter also allows for the specification of trace record formats, so that the filter can process traces coming from different systems. A complete description of the current filter design is presented in [7].

Typically, the processes forming a computation send traces to a single filter, which selects and stores the data for later analysis. We would expect this structure to be useful to a programmer evaluating a new program. Different configurations provide for the ability to apply the measurement system to different problems. For example, we could have a filter collect data on all communications activities within a single machine. This type of configuration allows the measurement of message quantity and frequency, queue lengths, and process scheduling. DPM can perform the tasks that would have traditionally required specialized tools to be built. It takes no extra work to extend this type of measurement to a collection of machines, or to the entire system. If network (communication) load is critical, then we can have a filter on each machine and merge the trace records when the computation has terminated. If CPU load is critical, then the filter process(es) can be placed on its (their) own machine(s).

### D. User Interface

The user interface to DPM is a command interpreter that allows the programmer to specify the 1) program (processes) to run, 2) events to monitor, 3) name of the filter (with descriptions and templates, if the standard filter is used), and 4) the analyses to be run on the trace data after they are collected. A complete description of the command language and structure of the user interface is given in [8] and [7].

## IV. Analysis Techniques

A collection of data needs some form of interpretation to have some meaning. A basic tenet of this paper is that the measurement model and techniques, and the associated tools, can provide useful data. To demonstrate this, we describe several approaches for the analysis of the trace data generated by our measurement system. These analyses are implemented and working in DPM.

### A. Basic Communications Statistics

We have defined a computation to be a collection of cooperating processes. The processes cooperate, and the cooperation is based on some communications mechanism. It is reasonable then to want to know the nature of the communications between processes. Several basic questions come to mind. Who is talking to whom? (Which processes are talking to which other processes?) What is the volume (total message traffic) of the communications? How frequent (time density) are the communications? How large are messages?

In addition to these basic questions, a few more interesting queries come to mind. Given information about the arrival and consumption of messages, we can derive information about the message queues. It is possible to gather statistics such as the maximum and average queue lengths for each process. With the same information we can obtain the minimum, maximum, and average time that a message waits in the incoming message queue before it is consumed by the process. We can also record the distribution of the various measurements.

### B. Detecting Paths of Causality

When we write a computation consisting of several processes, we specify the order and frequency of the communications in the computation. We specify this information for each interaction between processes. We establish rules and protocols to provide for the correct execution of the program. But when the entire computation is executing, the overall interactions are more complex than suggested by this static picture of the computation. The increased complexity comes from parallel execution within the computation and from the fact that several partially completed activities maybe be simultaneously active within the computation.

The model of computation in which we are interested for this analysis is that of a *server*. A server is a computation that receives a request from processes outside the computation, computes a result (involving one or more of the processes within the computation), and then returns the result to the requesting process. There are several questions to be answered about the behavior of a server. One question is: given a request message received by the server, what message paths within the server are most commonly traveled? This question can be translated to: what sequences of interprocess communications occur most frequently? For sequences of length two, we can derive this information from the basic message statistics (see Section IV-A). The message statistics cannot provide information about longer sequences of interactions. Related to the longer sequences of interactions is a second question: given a process that has just received a message, where will that process next send a message? This question can also be viewed as determining a branching probability, given a specific input to a process.

The basic strategy for causality analysis is to identify each request to the server, and follow the sequence of interactions within the server caused by that request. To do this, we first collect SEND and RECEIVE traces and construct a program history graph of the events

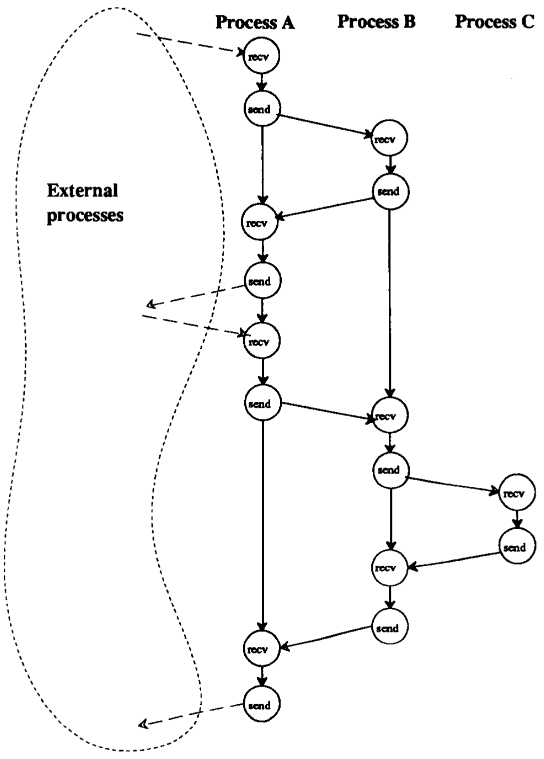

Fig. 2. Sample computation graph for causality analysis.

in the processes, with arcs in the graph between each corresponding send and receive (see Fig. 2). A SEND trace is associated with its corresponding RECEIVE trace by using two pieces of information. First, we can identify the connection or link over which a message was sent. Second, messages sent over a given connection contain sequence numbers, and the SEND and RECEIVE traces record these sequence numbers. The program history graph represents the complete collection of interactions between processes during the life of the computation.

To reduce the complexity of analyzing such a large quantity of data, we convert the problem to one of manipulating character strings. Each process in the computation is assigned a single letter designator. The two events corresponding to a message being sent and received are designated by the letter for the sending process, followed by the letter for the receiving process. For example, the first send and receive pair in Fig. 2 would be the string

$$AB.$$

We create a list of strings, where each string represents one request to the server and the subsequent activity within the server. These strings are called *causality strings*.

There are three types of processes that are visible during a causality analysis. The first type of process is the *server* process. This process is contained within the computation that is providing service. The second type of process is the *requestor* process. Requestor processes are the customers of the server. They make requests and receive results. The last type of process is the *system* process. System processes are those processes to which the server makes requests. The system processes may be other servers, or perhaps a host kernel. Messages received from a requestor indicate a request for service from the server computation. Messages to system processes from the server are ignored (as are the responses), as we are interested in tracing the flow of control through the server, and not through external processes.

The algorithm for building the causality strings traverses the entire computation graph. The list of causality strings is built by

1) searching for each message receive event from a requestor process;

2) for each such receive, the message sends immediately following the receive are identified;

3) the message receives corresponding to sends in 2) are identified (i.e., the message arcs are followed), and steps 2) and 3) are repeated for each receive.

A causality string is initially a single character, which is the process performing a message receive that was detected in step 1). Each time a send is followed (i.e., a message arc is traversed) to its corresponding receive [step 2)], an additional letter (identifying the receiving process) is added to the causality string. Events associated with system processes are ignored in this algorithm. For example, the causality strings for Fig. 2 are

$$ABA$$

$$ABCBA.$$

Once we have the causality strings, there are several results that we can obtain from them. The first result is identifying the most commonly traveled paths through the server. We store these strings in lexicographical order with a value indicating the number of times that the string has occurred in the computation. This list of strings identifies the most commonly occurring message sequences.

Given three processes, $A$, $B$, and $C$, we use the causality strings to compute the probability that process $A$, having just received a message from process $B$, will next send a message to process $C$. This information is obtained by generating all of the substrings of length three, and then tabulating them. The probabilities cannot be calculated from the simple message statistics since these statistics do not correlate message receives with the corresponding message sends.

This analysis technique was used in a study of the DEMOS/MP file system [9], [2]. The DEMOS/MP file system consists of four processes (request interpreter, directory manager, buffer manager, and disk interface) which function together to provide a file service to user processes.

The file system was run under heavy loads (many user processes) and its execution was monitored by DPM. User processes were distributed among most of the machines. The trace data that were collected were used to build the graphs and strings described previously. Substrings of length three were used to compute the probability of messages flowing between the file system processes and these probabilities were used to construct the diagram in Fig. 3. This figure shows the flow of data and control within the file system.

The interesting result is that by using a general performance tool, such as DPM, and knowing nothing about the internal structure of the file system, we can obtain valuable information about its internal operations.

For example, from Fig. 3, we can conclude the following.

1) Messages from users go to the directory manager only 4 percent of the time (corresponding to file opens). This provides us with the ratio of file opens to reads/writes.

2) The request interpreter asks the buffer manager for data, and 73 percent of the time a result is immediately returned. 27 percent of the time the buffer manager must ask the disk interface for the data. This represents the buffer hit rate (cache efficiency).

3) The buffer manager will ask the disk interface for additional blocks of data 17 percent of the time, representing the frequency of following indirect references on the disk.

The above results give important structural information when provided to the programmer/analyst of the file system. These results were provided without the need for specialized measurement tools.

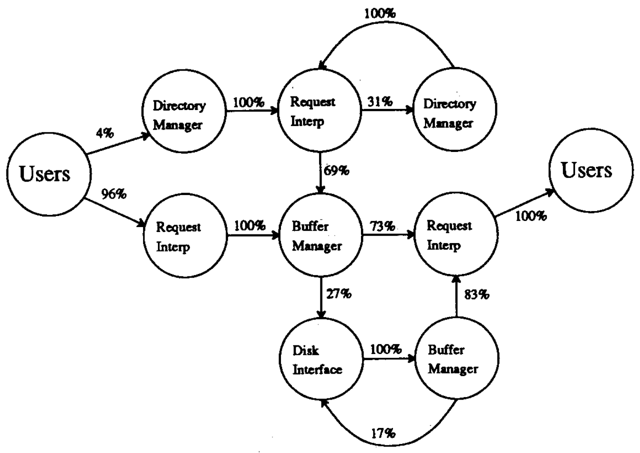

Fig. 3. File system state diagram.

### C. Measuring Parallelism in a Computation

One motivation for writing a distributing computation is to achieve an increase in its speed of execution. This performance increase is obtained by means of parallel execution of the processes within the computation. Once we construct a distributed program, the problem becomes: how do we measure the degree of parallelism in the execution of our program? In addition, it is useful to be able to project the amount of parallelism that can be achieved as we vary the location of processes among machines.

The algorithm for the analysis of parallelism uses traces obtained from the execution of a program and constructs a program history graph similar to the one used in the causality analysis (see Fig. 2). An arc between two event nodes in the same process is labeled with the amount of CPU time consumed by that process between those two events (calculated from the PROCTIME field of the event traces). An arc between a SEND event and its corresponding RECEIVE event is labeled with the delivery time (or an estimate of the delivery time) for the message. Once the graph is labeled, we calculate the length of the longest path through the graph, $t_{\max}$. We also calculate the total amount of CPU time used by all processes in the program for this execution, $T$. The parallelism (or speedup) is equal to $T/t_{\max}$.

This method of measuring parallelism can be used even on systems that are currently running other users (since it is based on process time, and not on real time). The techniques used for this analysis are described in detail in [10] and a study using this analysis is described in [11].

## V. Variations on a Theme

### A. Real-Time Monitoring and Display

In Section IV we assumed that the data analyses were performed after the program had completed. The structure of DPM provides the flexibility to use analysis routines that process trace data as they are generated. The trace log files can become large (we have had examples of 10-50 Mbyte files). and real-time monitoring does not require the traces to be stored. We can store the results of the analysis or these results as the basis for graphic displays. The graphic displays could also be driven from stored trace logs.

The analysis of communication statistics presented in Section IV-A is easily adapted to execute in real time. The more complex causality and parallelism analyses would be much more difficult to perform in real time.

We are currently using several types of graphic displays for the real-time communication statistics analysis. These are the "hot spots," matrix, and history displays. The hot spots display represents a distributed program as a collection of nodes (processes) connected by directed arcs (message channels). Communication traffic levels are displayed by coloring the arcs between process nodes. As traffic levels increased, the color of an arc progresses from violet to red, thus identifying the active communications paths in the program. The numeric values associated with the traffic level are used to label each arc. The metrics that can be displayed on the message arcs are messages/second, bytes/second, (cumulative) message counts, (cumulative) byte counts, average message sizes, and input message queue lengths. The processes are also colored—representing the amount (percentage). The hot spots display allows a programmer to quickly identify the program's execution patterns of time that the processes execute.

In addition to the basic display of metrics, the hot spots display can show the first derivative of the metrics. The derivative values provide information about the changes (phase behavior) in a program's execution.

The hot spot display is useful for small to medium size programs. When the number of processes increases to more than 10 or 15, the display becomes difficult to comprehend. We use the matrix display for larger programs. The matrix display represents a distributed program as a square matrix with each process labeling the same row and column. The values in the matrix element $[i, j]$ represent the message traffic from process $P_i$ to process $P_j$. The matrix elements are colored to indicate the traffic levels as was done in the hot spots display. The labels of the matrix (process names) are colored as were the nodes in the hot spots display. The order of the processes in the matrix can be changed to group together the most active processes or communication paths.

The history graphs display a single value from either the hot spots or matrix display. This value is displayed over a time period extending from the present time backwards. The history graph is used to provide a reference for the hot spot or matrix displays (which only present current data).

### B. Feedback Scheduling (Load Balancing and Other Uses)

The analyses discussed thus far are intended to provide the programmer or system manager with information. This information may cause the programmer to restructure the program being studied or cause the manager to change the environment in which the program executes. In both cases, a human being is part of the feedback loop.

It is possible to return the results of some of the data analyses directly to the host system. After the meter traces have been analyzed and reduced to some reasonable statistic, this information can be passed directly to the host system to be used in scheduling decisions. The trace data become direct feedback to the host operating system.

We can use this structure to collect communications load data for

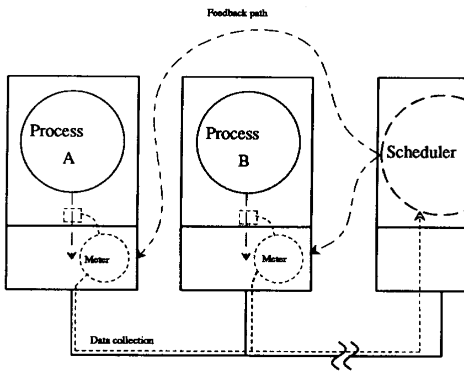

Fig. 4. Measurement system for feedback scheduling.

process load balancing and file migration decisions. CPU load information can be gathered in most operating systems, but communications load data are more difficult. The difficulty increases if we are interested in data about a specific process, rather than the system as a whole. DPM allows us to measure communication levels between any two processes.

The data collection for feedback scheduling is similar to that described in the previous section on system communications measurements. The same type of organization for metering and filtering could be used. The filters would not store the selected data. After filtering, the selected data would be passed on to the host system to provide data for scheduling. This is illustrated in Fig. 4.

The type of resource scheduling that can make use of the meter data is limited by the frequency with which the data need to be collected. It is reasonable to measure activities that occur with about the same frequency as interprocess communications, or with lower frequencies, since the meter traces themselves are messages. Attempting to measure activities more frequent than these would overly degrade system performance. The measurement system can provide information that is gathered from other sources. These other sources could be more traditional performance tools gathering data on machine loading, memory usage, or paging activity. The meter message would be the medium that would carry periodic summaries of these other activities.

## VI. Conclusion

DPM is a simple tool. Each piece was constructed to provide the needed functions by the most straightforward means. This simplicity provided for its ease of construction (in two operating systems) and for the flexibility of its design. Transparency to the programmer extended the range of applications. We can measure any program (including existing system services) and programs written in any language. We minimized the restrictions on the use of the tool.

Two design decisions were made in DPM that have inspired some controversy. The first is the lack of user-defined event trace types. While it would be easy to add this to DPM, we have resisted the temptation so as to maintain transparency. The second decision was the lack of message contents in the meter traces. An argument for the inclusion of message contents comes from monitoring the synchronization protocols in a distributed database system. For example, the last traces we obtained may have come during a commit/abort decision followed immediately by a system crash or deadlock. We would like to be able to know whether a commit or abort was occurring and this might be difficult without knowing the contents of the synchronization. However, the inclusion of the PC field in the traces provides us with information on which procedure generated the trace events and this could provide the additional information needed to discriminate between the commit and abort in our example.

DPM is a running system. It was originally developed on the DEMOS/MP operating system and is now running under the Berkeley UNIX 4.3BSD operating system. Research on DPM continues. The current analyses (communication statistics, paths of causality, parallelism) have provided useful tools for program development. Work is ongoing in the areas of real-time monitoring facilities, feedback scheduling, and graphic display techniques.

## References

[1] M. L. Powell and B. P. Miller, "Process migration in DEMOS/MP," in *Proc. Ninth Symp. Oper. Syst. Principles*, Bretton Woods, NH, Oct. 1983, pp. 110–119.
[2] B. P. Miller, D. L. Presotto, and M. L. Powell, "DEMOS/MP: A distributed operating system," *Software—Practice & Experience*, to be published.
[3] W. Joy, E. Cooper, R. Fabry, S. Leffler, K. McKusick, and D. Mosher, "4.2BSD system manual," Comput. Syst. Res. Group Tech. Rep., Univ. California, Berkeley, July 1983.
[4] H. C. Lauer and R. M. Needham, "On the duality of operating system structures," *Oper. Syst. Rev.*, vol. 13, pp. 3–19, Apr. 1979.
[5] G. McDaniel, "METRIC: A kernel instrumentation system for distributed environments," in *Proc. Sixth Symp. Oper. Syst. Principles*, Purdue Univ., Nov. 1975, pp. 93–99.
[6] P. Bates and J. C. Wileden, "An approach to high-level debugging of distributed systems," in *Proc. ACM SIGSOFT/SIGPLAN Symp. High-Level Debugging*, Asilomar, CA, Mar. 1983, pp. 23–32.
[7] B. P. Miller, S. Sechrest, and C. Macrander, "A distributed program monitor for Berkeley Unix," *Software—Practice & Experience*, vol. 16, Feb. 1986, also appears in short form in *Proc. 5th Int. Conf. Distributed Comput. Syst.*, Denver, CO, May 1985.
[8] C. M. Macrander, "Development of a control process for the Berkeley UNIX distributed programs monitor," M.S. thesis, Comput. Sci. Tech. Rep. UCB/CSD 84/216, Univ. California, Berkeley, Dec. 1984.
[9] M. L. Powell, "The DEMOS file system," in *Proc. Sixth Symp. Oper. Syst. Principles*, Purdue Univ., Nov. 1977, pp. 33–42.
[10] B. P. Miller, "Parallelism in distributed programs: Measurement and prediction," Comput. Sci. Tech. Rep. 574, Univ. of Wisconsin—Madison, 1985, to be published.
[11] N. Lai and B. P. Miller, "The traveling salesman problem: The development of a distributed computation," in *Proc. 1986 Int. Conf. Parallel Processing*, St. Charles, IL, Aug. 1986.

# On Self-Fault Diagnosis of the Distributed Systems

S. H. HOSSEINI, J. G. KUHL, AND S. M. REDDY

*Abstract*—**The problem of achieving fault-diagnosis in a network of interconnected processing elements called nodes, in which there is no central facility to control, coordinate, or mediate among the processing elements, is considered. Every node can eventually determine the status of nodes and communication paths between them. A diagnostic algorithm for homogeneous systems (systems with only testing nodes) is given. The self-fault-diagnosis of nonhomogeneous systems (systems with nodes of varying degrees of testing capability) is studied and diagnostic algorithms are proposed.**

*Index Terms*—**Distributed fault-diagnosis, homogeneous and nonhomogeneous systems**

Manuscript received March 26, 1985; revised April 14, 1987.

S. H. Hosseini is with the Department of Electrical Engineering and Computer Science, University of Wisconsin, Milwaukee, WI 53201.

J. G. Kuhl and S. M. Reddy are with the Department of Electrical and Computer Engineering, University of Iowa, Iowa City, IA.

IEEE Log Number 8715444.