The Administration of Distributed Computations in a Networked Environment
An Interim Report

Luis Felipe Cabrera [1], Stuart Sechrest, and Ramón Cáceres [2]

Computer Systems Research Group [3]
Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley. Berkeley, CA 94720

Abstract

Networks of computers running Berkeley UNIX™ allow users to program and run multiple-process applications that execute concurrently on several machines. We present solutions to the problems of process tracking, administration, and control in this networked computing environment. We have designed and implemented a *personal process manager* for an enhanced Berkeley UNIX system that provides the user with much needed process management and process control capabilities not found elsewhere.

The personal process manager is a distributed program based on a collection of user processes which make use of specialized system daemons. It provides on demand services, allows process control across machine boundaries, and may outlive the user login session in which it was created. When active, it becomes the process creation server for a user's remote processes, collects and preserves basic information about process activities, provides a notion of state of a distributed computation, and interfaces with several data analysis and data representation tools. The personal process manager also has crash recovery facilities.

1. Introduction

The advent of large local area networks of computers within institutions provides users with an unprecedented wealth of computing resources. The variety of processing alternatives now available was never present in traditional single-site timesharing systems. Although the aggregate computer power of these networks tops the fastest computers ever built, new operating system support and new user tools are required to make effective use of a network of collaborating hosts. In particular, process control across machine boundaries has to be dealt with in satisfactory ways. The management of multiple-process programs and multiple programs running simultaneously on several machines is a mostly unsolved problem in current computing environments. Many past and existing systems that have allowed multiple process collaboration have not included the user facilities necessary for administration of multiple-process computations whose components execute in several machines.

UNIX (22), in particular, provides for the management and control of a process within a host. This control is exerted indirectly by the user through facilities provided by (command interpreter) shells, or by calling a program to send a software interrupt to a process. The capabilities that this process control provides are well suited to the typical multiple-process program in UNIX, the pipeline of processes. Controlling a pipeline requires only the ability to control the shell's direct children, which is all that is provided in the UNIX C-shell (23). The UNIX paradigm of pipelined multiple-process programs is not, however, appropriate for general distributed computations. Here, arbitrary genealogical process structure relationships should be allowed to exist without sacrificing the user's ability to control the computation. Other UNIX based systems (1, 24) have not gone beyond the pipeline paradigm. In (21, 24) there are mechanisms for user controlled migration of processes. However, neither of these systems has user facilities for locating the execution sites of a distributed computation and broadcasting, say, a software interrupt to stop execution. Although 4.2BSD Berkeley UNIX has networking capabilities (13, 23), it does not have adequate user facilities for process control across machine boundaries.

Another deficiency of existing systems has been the lack within process management facilities of historical information about the processing behavior of computations. Those event tracing tools that have existed in different computing environments (15-16) have not been incorporated into process management facilities in the corresponding systems. Multiple-process computations that span processors require for their appropriate management not only powerful and flexible mechanisms for process control but also historical processing information. In this way history dependent events can be set by users to trigger process state changes. Systems that were designed to support multiple process computations, like task forces (6, 9, 17) and

---

1 Author's current address: Computer Science Department, IBM Almaden Research Center, 650 Harry Road, San Jose, California 95120-6099.

2 Author's current address: Pyramid Technology Corporation, 1295 Charleston Road, Mountain View, California 94039-7295.

3 This work was sponsored by the Defense Advanced Research Projects Agency (DoD), monitored by the Naval Electronics Systems Command under contract No. N00039-84-C-0089. The views and conclusions contained in this document

teams (4-5), do not have these capabilities. Configuration languages (6, 11-12) have not included such history dependent events in their process model.

As a solution to some of the problems of distributed computation administration and process control we have implemented the personal process manager, PPM, for networks of computers running an enhanced Berkeley UNIX 4.3BSD. In Berkeley UNIX 4.3BSD interprocess communication can be accomplished using different addressing families and styles of communication. Two processes wishing to communicate need not have a common ancestor nor reside in the same host. The PPM can determine in which state (running, stopped, or dead) each of the component processes of a multiple-process program is, find what resources have been used, locate in which processor each process is executing, and find the genealogical relationships between them. Moreover, process management facilities should also allow a user, in event dependent ways, to change the state of each of its processes and possibly the site of execution. Our process management mechanism also allows the delivery and handling of software interrupts with no interprocess constraints based on creation dependencies. The PPM does not currently support a configuration language. It provides access to its facilities through subroutine calls.

The current implementation of the personal process manager is a distributed program based on a collection of user-level processes. It provides on demand services, allows process control across machine boundaries, and may outlive the user login session in which it was created. When active, it becomes the process creation server for a user's remote processes, collects and preserves basic information about process activities, and interfaces with several data analysis and data representation tools. The PPM provides a layer of intercommunication based on reliable stream connections, allowing tools to ignore all topological aspects of requesting and gathering distributed information. The current PPM does not have process migration facilities.

The PPM is an instance of an architecture for process administration in a distributed processing environment based on collaborating local agents. The local agents individually provide all administration functions for the processes of a user within one host, and collectively maintain the state of a user's computation in a distributed environment. In our case local agents also play an active role in process creation and serve as trusted authentication agents accross machines. All of the services provided by the process administration facility are on demand in order to minimize system overhead.

The rest of this paper is subdivided as follows. In Section 2 we present the basic functions of the personal process manager. Section 3 discusses its design and the security issues involved with this kind of facility. Section 4 contains a description of the manager's operation in a failure-free environment while in Section 5 we present its error and crash recovery facilities. Section 6 presents implementation and performance considerations. Section 7 points to future work, while Section 8 consists of our conclusions.

2. Functions of the Personal Process Manager

A primary consideration in our effort has been that the implementation should be layered on top of existing mechanisms, with kernel changes kept to a minimum. The six additional design considerations behind our implementation are (1) that users need to be able to track, manage, and control their processes in flexible ways, (2) that mechanisms for user services should be on demand, (3) that the mechanism's overhead should be proportional to the amount of service provided, (4) that extensive event tracing facilities should be selectively available for user management of processing activities, (5) that the mechanism should scale well, and (6) that it should be robust. Our solution applies to networks of computers that have explicit machine boundaries and that share administrative authority.

The *personal process manager*, PPM, is a distributed program implemented as a collection of user-level processes called *local process managers*, LPMs. LPMs are created on demand, and are the basis of our management and control mechanism. A computation is considered to be a group of processes that have a common logical ancestor. Under the PPM the processes form a (logical) tree that may span a number of machines. Under some failure modes this tree may become a forest. It could also become a forest upon *exit* (23) of a process. However, we chose to retain exit information while there are children alive, and for the display of a genealogical distributed computation snapshot we mark the process as exited. Process identities can be made globally known to the network of machines. Thus the user may track and control each process, and simultaneously manage a number of distributed computations consisting of multiple processes. The implementation of the PPM includes the utilization of system daemons, and has required some modification of some key system calls provided by the 4.3BSD kernel.[4]

The decision to create a separate LPM for each user of a machine was motivated by three considerations. First, we wanted to minimize overhead and localize its cost and effects. We preferred to create process managers on demand so as to minimize the effect on users not using PPM. Second, we wanted to avoid unnecessary modifications to the operating system. We preferred communication between user processes to communication at the kernel level, because this is more consistent with current UNIX and more easily instituted between machines running different operating systems. Third, appropriate cooperation among processes belonging to a single user is easier to to define than cooperation among processes serving many users. We have chosen the perspective that process management is a problem of administering the processes of a particular user without regard to machine rather than the processes of a particular machine, without regard to user.

The design of the PPM extends a layer of the shell across several machines, rather than having the shell make use of kernel support offered remotely. The creation of a number of LPMs establishes a *session* in which the LPMs participate until the normal termination of each. The LPMs are able to perform authentication when channels are created, rather than upon every request. While the session continues, new LPMs may be created and come to participate. When LPMs lose contact with one another, they must seek to reestablish this contact. Process managers that acted in service of several users upon a machine could not be easily charged with discovering whether the owner of some process is logged into a different machine (at least not without significant kernel changes).

The PPM may outlive the user login session in which it was created. When present, the component LPMs become process creation servers of remote user processes. The LPMs gather and preserve local infor-

are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Defense Research Projects Agency or of the US Government.

4 UNIX daemons are system owned processes that are always available for use.

mation about user process activities, accept parameters that determine the amount of process events recorded, communicate with sibling LPMs, have crash recovery facilities, are message based, provide a notion of state for a distributed computation, interface with several data reduction and data display tools, and allow users to administer their processes in very flexible ways. We expect PPM performance to degrade gracefully when user processes span large number of nodes in an internetwork of computers.

3. Design of The Personal Process Manager

PPM is a distributed program implemented as a collection of LPMs. Per-host LPMs allow us to use existing mechanisms for process control within each host, and to minimize interhost messages and operations. It is not possible to require a site to be omniscient and still expect such a mechanism to scale well. The UNIX reality of many short lived processes highlights the desirability of on demand information and control requests. Decentralizing the implementation of the PPM was also motivated by the fact that most processes are never subject to control and information about them need not be propagated between machines.

PPM provides a variety of services to the user through tools that connect to the local LPM. Requests to the local LPM are forwarded to remote LPMs when appropriate. The management of interconnections between LPM's, however, is not the responsibility of the user. The local LPM will create a remote LPM when one is required, and maintain communication with the remote LPM when this is possible. Placing these responsibilities in the PPM greatly simplifies the work required of tools and applications using the PPMs services. Figure 1 displays the process genealogy a PPM may present the user when computations exist in three hosts.

All actions are symmetrical for all the LPMs participating in the management of a user's processes. The only exception to this rule is crash recovery. The crash coordinator site, CCS, is a specific LPM that acts as the leader in the presence of failures, as will be described in Section 5.

LPM creation is somewhat expensive in terms of message exchanges and in local processing. Therefore LPMs have a time-to-live period during which they are still present in a host even though that host may no longer contain processes belonging to their user. The creation of an LPM is accomplished with the help of two system daemons. First, the creation request is directed to the *inet* daemon (14),[5] *inetd*, which then passes the request to the process manager daemon, *pmd*, creating it if necessary. This daemon proceeds then to create the LPM, and returns the accept address (Figure 2) after verifying that there is no LPM for that user in that host. If an appropriate LPM is found in the host, its accept address is returned. These four steps are labelled (1), (2), (3), and (4) in Figure 2. The process manager daemon is present in an installation as long as there is any LPM present. It serves as a trusted name server for the creation of LPMs. Authenticated connections are then achieved by establishing a private reliable stream communication channel between sibling LPMs (Figure 3).

Our current authentication scheme can only prevent user-level masquerade. Host-level masquerade is not well addressed in most distributed systems. Verifying hosts' identities generally requires the sharing of secret information. Mechanisms for storing and using such information are not built into Berkeley UNIX. We have, therefore, not yet addressed the problem of host masquerade.

The 4.3BSD IPC mechanism provides no direct support for preventing user-level masquerade. User id's and process id's are not valid across machine boundaries and are not passed when connections between machines are created (18). We use the process manager daemons as trusted name servers, and communication between sibling LPMs is done by reliable virtual circuits provided by TCP connections. This allows us to avoid the use of system-wide unforgeable tickets for authentication. Virtual circuits, however, limit extensibility. A datagram based scheme would scale much better, but would require individual authentication for each message. In our computing environment TCP connections are also needed to assure message delivery.

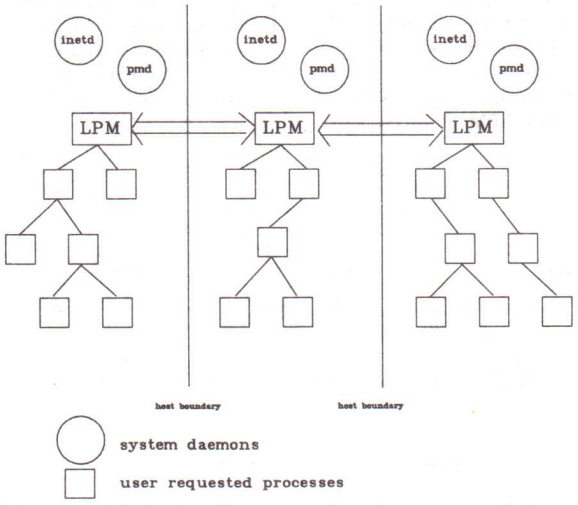

Figure 1: Possible State of a PPM Spanning Three Hosts

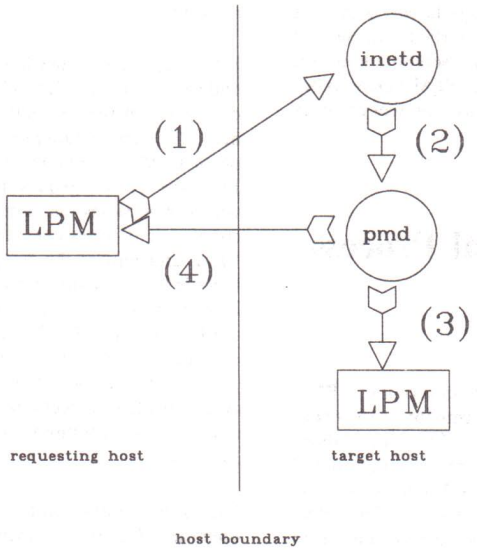

Figure 2: LPM Creation Steps *Ab Initio*

A reliable datagram protocol and a scheme based on remote procedure calls, would be promissing alternatives for scalability.

4. Failure-Free Operation

The services of LPMs must be explicitly requested. The PPM mechanism is not integrated with any command interpreter, and thus its services must be obtained by one of a series of tools (which may include command interpreters). Our present tools include snapshots, with basic process control functionalities (stop a process, execute it in the foreground, execute it in the background, kill it), and exited process resource consumption statistics. They are implemented within the PPM (just as UNIX shells have some built in commands). There are interfaces for other tools. Tools invoked by a user establish reliable stream connections with the appropriate LPM. User requests that require the collaboration of more than one LPM are handled through the communication infrastructure of the PPM maintained by the LPMs, which is transparent to the user.

A user's LPM has connections with local tools and with sibling LPMs on remote machines. LPMs also receive messages from the local kernel. All data pertaining to the local user's processes are obtained in this way. Kernel modifications were necessary to generate messages within the kernel. In Berkeley UNIX process communication end points are called *sockets*. LPM sockets are subdivided into three groups (Figure 4): one socket where the kernel deposits its messages (called *kernel*), another socket (called *accept*) whose address is distributed by the process manager daemon, and possibly multiple sockets for communication with sibling LPMs and local tools. LPMs use primarily 4.3BSD mechanisms for intramachine process control. The LPM gains write access to the process control block of its adopted processes by an extended form of the UNIX debugging access call *ptrace* (3). A similar but more ambitious form of expanded access has been described in (10).

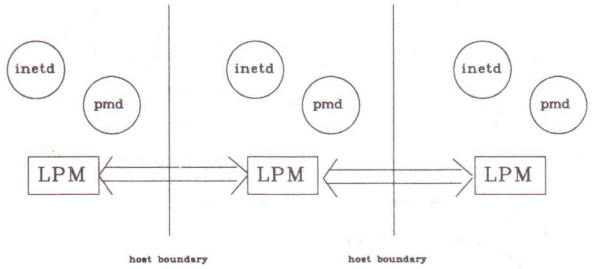

Figure 3: All LPMs of a PPM Maintain a Secure Reliable Communication Channel

5 This is an alternative to having a well known communications port.

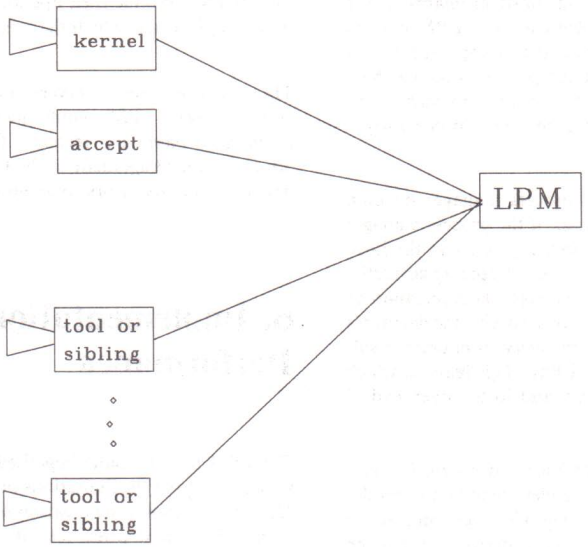

Figure 4: LPM Types Of Communication End Points

Requests can be made to LPMs to adopt particular processes. Adoption allows the LPM to keep track of a process and its descendants. Adoption may be necessary if the user did not invoke the process management services at login time. Moreover, adoption also provides the flexibility desired for the possible use of the mechanism by a debugger. The adoption operations fail if the process and the PPM belong to different users. As a result of adoption, user processes are modified to contain specific tracing flags used thereafter by the kernel for event detection. This mechanism is much like that described in (16).

The interconnection of the sibling LPMs is not a concern of the user. Interconnections are established when needed and maintained as long as they are likely to be used. Having logical children, i.e., a process creation in a remote host, is one such reason. Having a connection with the CCS during the recovery from failures is another. The interconnection topology is therefore quite flexible. In most operational scenarios we expect to have only very few of all the potential connections between sibling LPMs in place. The topology of the interconnection graph is dependent on the process creation patterns exhibited by the user processes, their intercommunications, and the failure modes in the network computing system.

The PPM will outlive a user login session if processes created by that user remain active in the networked computing environment, or if the time-to-live intervals of some LPMs have yet to expire. In these cases, a user's request for a LPM following a new login will yield an existing one. This simple scheme allows users to regain knowledge and control of all of the processes that have been created under the PPM mechanism in the past and are still alive. At least potentially, extensive historical information about the processing that took place while the user was logged off should also be accessible.

Because our on-demand communication topology is designed to produce low-connectivity graphs, we have to pay a price for broadcast requests. The PPM uses a graph covering algorithm. A scheme for not retransmitting old broadcast requests has been implemented using a signed timestamp in which the name of the originating host appears. The appropriate time window for retaining old broadcast requests is a configuration parameter whose optimum value will be derived from experience. All data returned to the originator of a broadcast request includes the message's source-destination route. This allows quick routing of messages affecting processes in topologically distant hosts. No attention is currently devoted to finding minimum hop routes to nodes. Information about process activities and events is not sent across hosts unless there is an explicit request.

The desicion whether or not to propagate connection information between sibling LPMs in order to increase the connectivity of the communication graph is a function of the cost of maintaining connections and of the additional benefit of the connections. For availability purposes, for example, multiple interconnections within one ethernet do not increase the probability of the services being operational. The frequency of use of a connection, on the other hand, depends on the speed and load that the underlying medium has. These two aspects need to be considered in the policies adopted regarding the communication topologies maintained by the LPMs and the routing of messages between LPMs.

It is the responsibility of network system administrators to have consistent password files across machines that trust each other. Authentication at the user level is done using the existing 4.3BSD facilities, including the use of .rhosts files that increases the flexibility of remote access to machines. No attempt is made to hide machine boundaries.

5. Robustness and Recovery

A PPM should continue to operate in the presence of LPM, host, and network failures. The PPM should inform the user about the nature of the failure and reestablish its internal consistency quickly. Host crashes affect our management scheme only in that communication paths may be lost. All process activities in that host, obviously, cease. Were we managing resilient computations, control would have to be carefully transferred to another host. This can be achieved with

robust protocols implemented on top of our basic mechanism. We have chosen not to do so in our first implementation. LPM crashes are handled just as host crashes. However, the disappearance of a LPM does mean that information about the processes in that host will be lost. This may make the actions of certain tools more complicated. For example, the snapshot of the genealogical process structure may now become a forest.

We expect few crashes to be due to the process manager daemon itself as its structure is very simple. However, if the process manager daemon loses information about a LPM currently active in the host, then the process management mechanism does not operate correctly. The state information kept by the process manager daemon could be stored in secondary (even stable) storage so as to survive the daemon's possible failure modes. This would allow recovery from crashes suffered only by the daemon but not by any LPM. This feature, which has not been implemented, would certainly add to the overhead of creating LPMs.

At all times in normal operation, one LPM has the distinguished role of being the *crash coordinator site*, CCS. Under some failure modes we may have no CCS or multiple CCS. The CCS becomes active only when a failure is detected. CCS selection is performed through user information, or established by default by the system when the user first invokes the mechanism. The .recovery file in the user's home directory has a list of hosts in decreasing order of priority in which their CCS should reside. This recovery list is the basis of the driving search strategy for recovery. We assume that the recovery list will be short, will exist in all hosts where a user normally executes processes, and will contain the names of those hosts where the user logs in most often. Upon creation of a sibling LPM, the network address of the CCS is passed along. In case of need each of the sibling LPMs can establish a connection with the CCS. For the CCS, the time-to-live interval has a different meaning: as long as there is any sibling LPM in the networked system, time-to-live is not decremented.

Our crash recovery mechanism is based on the premise that, in a network of computers, users tend to use only a few hosts as home machines. These home machines serve as recovery orchestrators. The crash of a host (or a LPM) in the network results in LPMs trying to establish connections with the (known) CCS. If the CCS were found to be down, or inaccessible, or the process manager daemon in that host were not responding, or it provided information regarding the foreign manager that is different from the information used in creating the connection, then the LPM that has detected the failure would try to connect in descending order of priority with the hosts listed in the user's .recovery file. If none of these hosts is available, a time-to-die interval exists that tells the LPM when to exit after having terminated all of the user's processes in that host. However, a LPM not in contact with a CCS resumes the normal mode of operation if it manages to connect to the CCS at any future retry, or gets a communication request from a LPM in contact with a valid CCS. The rationale for this policy is that, if things go wrong for a long period at those sites where the user most customarily logs in, and if the user has made no manual attempt to connect to the dispersed processes in his computation, the appropriate action is to close down all the activities. A different approach would have to be taken in a distributed computing environment with fully transparent processor boundaries, as there the notion of home machine would probably be meaningless.

The case of network partition requires some additional care, as separated subnetworks may exist each of which contains a host in the .recovery file. What happens in this case is that those new CCSs that are not at the top of the list keep probing, at a low frequency, the hosts higher on the list. Whenever such host comes up, they connect to it. Our current implementation allows connected components of this kind to continue their operations with no bounds in time because they include a host which the user is presumed to log into frequently.

The existence of name servers in the network could be used to aid in crash recovery. LPMs would query the name server for a CCS. The mechanism based on .recovery files would not be needed. In this approach the assignment of the CCS could be better coordinated by network administrators to avoid possible bottlenecks.

6. Implementation and Performance

The PPM has especially benefited from three previous efforts. First, we learned from the limitations of the *rexec* facility present in 4.2BSD (23). *Rexec* allows the creation of remote processes and the delivery of signals to these processes. By itself, however, it is insufficient for starting distributed computations since no provision is made for flexibly configuring the communication links and open files of the remote process, or for separately signalling any children of the remote process. Moreover, since the *rexec* call is made directly from a user process to a remote daemon, the shell's process control facilities do not affect the remote processes. Remote processes must therefore be explicitly hunted for and signalled. Second, in the Summer of 1984, a process control mechanism had been designed and implemented for 4.2BSD Berkeley UNIX (3). That mechanism dealt exclusively with the problem of software interrupt delivery across machine boundaries. It required all processes to have a control socket, and there was a centralized system wide process control facility. That experience led us to formulate several of our design decisions. Third, we had at our disposal a system similar to Xerox's METRIC system (15) implemented for 4.2BSD Berkeley UNIX (16). The synthesis of these three experiences and our current functional requirements inspired our present PPM.

The LPMs in the current implementation execute code written in C. They accept messages arriving from tools, the kernel, and other LPMs. The LPM is, itself, a multi-process program. It consists of a main dispatcher process, and some number of handler processes. Many of the arriving messages contain requests that can be acted upon immediately. Others contain requests that require communication with other processes. To avoid unnecessarily complex code and unnecessary delays, these latter requests are handled by processes created by the dispatcher. These handler processes may block while waiting for a response from a remote process without interrupting the service of the LPM. Since process creation in UNIX is relatively expensive, processes that have handled a request may be given further requests, rather than simply creating new processes. If responses are never received by a handler, they inform the dispatcher of the failure, which returns a failure message to the originator of the request. Otherwise, a positive response, together with any associated information, is forwarded to the originator.

At present, our implementation includes two tools: snapshots with process control, and exited process resource consumption statistics. Work is beginning on graphics interfaces for these tools and on various additional tools. A library of subroutines handles most interactions with the PPM, so that user-written programs may easily make use of PPM's capabilities.

A software interrupt delivery mechanism based on the processes as files approach presented in (10) is a very elegant alternative to our message based approach. Through the incorporation in the file system of the /proc directory, one is able to access any process in the system. With the advent of a network file system (25), that mechanism extends to multiple hosts. Had we had such code, we would have used it for message delivery, replacing several functions of the LPMs. However, those aspects of process management that incorporate event detection cannot be handled by that approach without the additional capabilities present in our LPMs. Nor does the /proc mechanism easily generalize to provide the creation and configuration of remote processes.

Load	Host Type		
	VAX 11/780	VAX 11/750	SUN II
$0 < la \leq 1$	7.2	7.2	8.31
$1 < la \leq 2$	9.8	9.6	14.13
$2 < la \leq 3$	13.6	12.8	22.0
$3 < la \leq 4$		18.9	42.7

Table 1: Estimated 112-byte Kernel-LPM Message Delivery Time in Milliseconds. Load estimator: *la*.

We only have preliminary assessments of both the overhead and performance of some modified system calls, and of the overall efficiency of the built-in capabilities. The code added to the system calls typically amounts to a 40 line message delivery function. The runtime overhead for the users not requiring the PPM is negligible, as it only involves comparing to zero the value of a variable. The estimated cost of transmitting a 112 byte message between the kernel and a LPM is displayed in Table 1. The load estimator used, *la*, is a time-averaged cpu run queue length. Table 2 shows the time taken to create a remote process on a machine where an LPM is running and the time to stop or terminate the process. The process creation time does not include the time to create the LPM or to form a connection with it.

action on processes	Topological Distance		
	within host	one hop	two hops
create	77	N/A	N/A
stop	30	199	210
terminate	30	199	210

Table 2: Elapsed Time of Process Creation and Termination Events in Milliseconds (N/A: Not Applicable.)

We have also performed a limited set of measurements of the elapsed time required to gather snapshot information about distributed computations. For this purpose we transmitted between the appropriate LPMs information about six user processes in each of the remote machines. The resulting times are shown in Table 3. Figure 5 displays the four PPM topologies used. Processes are identified in the network by <host name, pid>. Our timing results compare well with the general performance results found in (2) for the networking subsystem of 4.2BSD.

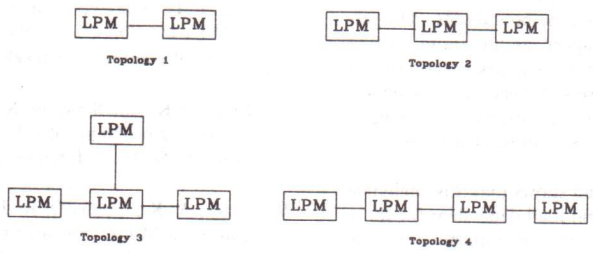

Figure 5: Snapshot Configuration for Four PPM Topologies

	topology 1	topology 2	topology 3	topology 4
Time	205	225	461	507

Table 3: Elapsed Time in Milliseconds To Transmit Snapshot Information in Four Topologies

7. Future Work

The current implementation and design of the PPM allows extensions in several directions. Providing more user tools which use the interfaces of our mechanism has high priority. User utilization of the system depends on the availability of more tools. If we look towards distributed systems with no host boundaries some of our design decisions need to be reevaluated and the implementation certainly be changed. One area of our implementation that deserves a second look is the establishment and maintenance of the PPM communication topology. Different administration goals may dictate different requirements for the underlying communication topology. Our current set of criteria may not suffice. A PPM could cater for tailored management of different types of computations. Mechanisms could be different for short-lived distributed computations than for long-lived ones. Mechanisms must certainly be different from the current ones for management of resilient computations. One could also envision enhancing the PPM to manage operations on special objects such as replicated files.

8. Conclusions

A resilient mechanism for on demand flexible process management in a network of computers has been presented. We call it the personal process manager, PPM. Its prototype implementation is viewed as a tool for experimentation in networked environments, as well as a testbed for ideas to be implemented in a host transparent distributed system. The PPM provides the user with capabilities not found elsewhere. In particular, it allows for process control across machine boundaries, as well as for event driven user defined actions. It is resilient to software, host, and network failures.

The PPM may gather and use historic data for process management. The PPM overhead is proportional to the services requested. Most actions are performed on demand. There is also the flexibility to have many different tools for data reduction and display. The PPM's algorithms were designed to scale well into the tens of nodes, but we have yet to stress test our implementation. The current design does not apply to systems without explicit machine boundaries. The services of the PPM can be used by a debugger, as the granularity of event tracing is user-settable. Moreover, in networking computing environments with only one or a few users per each host, scenario that we expect to be the dominant one in the future in a number of environments, our mechanism should operate most effectively.

An initial assessment of the PPM overhead shows that it is negligible for users not requiring the mechanism, and load dependent for those using it. The intrahost kernel-manager message transmission time varies between 7.2 and 42.7 milliseconds, depending on the cpu power of the host and its load. Remote process creation, once a connection between sibling managers exist, takes 177 milliseconds under lightly loaded conditions. Transmitting the required information about six processes, to display the snapshot of genealogical process dependencies, for a small set of remote hosts took between 205 and 507 milliseconds.

Our plans are to develop more tools that will use the interfaces present in the PPM. In particular a display tool, a historical data gathering tool, a tool for displaying the open and closed files of processes, a tool for displaying file descriptors, and one for IPC activity tracing and analysis. We also need to work on better human interfaces, and to assess our algorithms, especially in the area of message routing.

9. Bibliography

1. Brownbridge, D. R., Marshall, L. F., and Randell, B., The Newcastle Connection or UNIXes of the World Unite. Software -- Practice and Experience, Vol. 12, 1982, pp. 1147-1162.

2. Cabrera, L. F., Karels, M. J., and Mosher, D., The Impact of Buffer Management on Network Performance in Berkeley UNIX 4.2BSD: A Case Study. Proceedings of the 1985 USENIX Summer Conference, Portland, Oregon, June 1985, pp. 507-518.

3. Cáceres, R., Process Control in a Distributed Berkeley UNIX Environment. Report No. UCB/CSD 84/211, December 1984, Computer Science Division, University of California, Berkeley.

4. Cheriton, D. R., Malcolm, M. A., Melen, S., and Sager, G. R., Thoth, a portable real-time Operating System. Communications of the ACM, Volume 22, No. 2, pp. 105-115, February, 1979.

5. Cheriton, D. R., The V Kernel: A Software Base for Distributed Systems. IEEE Software, Vol. 1, April 1894, pp. 19-42.

6. Ericson, L. W., DPL-82: a language for distributed processing. Proceedings of the 3rd Int. Conf. on Distributed Computing Systems, 1982.

7. Ferrari, D., The Evolution of Berkeley UNIX. Report No. UCB/CSD 83/155, December 1983, Computer Science Division, University of California, Berkeley.

8. Jones, A. K., and Schwans, K., TASK forces: distributed software for solving problems of substantial size. Proceedings of 4th Int. Conf. on Software Engineering, 1979.

9. Jones, A. K., Chansler, R. J., Durham, I., and Vegdahl, S. R., StarOS, a Multiprocessor Operating System for the Support of Task Forces. Proceedings of the 7th SOSP, Operating Systems Review, Voll. 13, December 1979, pp. 117-127.

10. Killian, T. J., Processes as Files. Proceedings of the Summer 1981 USENIX Conference, Salt Lake City, Utah, June 1984.

11. Kramer, J., and Magee, J., Dynamic configuration for distributed systems. IEEE Trans. on Software Engineering, Vol SE-11, No 4, April 1985.

12. LeBlanc, R. J., and Maccabe, A. B., The design of a programming language based on connectivity networks. Proceedings of the 3rd Int. Conf. on Distributed Computing Systems, 1982.

13. Leffler, S. J., Fabry, R. S., and Joy, W. N., A 4.2bsd Interprocess Communication Primer. Report No. UCB/CSD 83/145, July 1983, Computer Science Division, University of California, Berkeley.

14. Leffler, S. J., Karels, M., and McKusick, M. K., Measuring and Improving the Performance of 4.2BSD. Proceedings of the Summer 1984 USENIX Conference, Salt Lake City, June 1984, pp. 237-252.

15. McDaniel, G., Metric: A Kernel Instrumentation System for Distributed Environments. Proceedings of the 6th SOSP, Operating Systems Review, Vol. 11, No. 5, November 1977, pp. 93-99.

16. Miller, B. P., Sechrest, S., and Macrander,C., A Distributed Program Monitor for Berkeley Unix. Software - Practice & Experience, to appear, 1985.

17. Ousterhout, J. K., Scelza, D. A., and Sindhu, P. S., Medusa: An Experiment in Distributed Operating System Structure. Communications of the ACM, Vol. 23, No. 2, February, 1980, pp. 92-105.

18. Postel, J., User Datagram Protocol. RFC 768, USC Information Sciences Institute, August 1980.

19. Postel, J., Internet Protocol - DARPA Internet Program Protocol Specification. RFC 791, USC Information Sciences Institute, September 1981.

20. Postel, J., Transmission Control Protocol. RFC 793, USC Information Sciences Institute, September 1981.

21. Powell, M. L., and Miller, B. P., Process Migration in DEMOS/MP. Proceedings of the 9th SOSP, Operating Systems Review, Vol. 11, No. 5, November 1977, pp.23-31.

22. Thompson, K., UNIX Implementation. The Bell System Technical Journal, Vol. 57, No. 6, July-August 1978, pp. 1931-1946.

23. UNIX Programmers Manual. 4.2 Berkeley Software Distribution, Virtual VAX-11 Version, August 1983, Computer Science Division, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, Berkeley CA 94720.

24. Walker, B., Popek, G., English, E., Kline, C., and Thiel, G., The LOCUS Distributed Operating System. Proceedings of the 9th SOSP, Operating Systems Review, Vol. 17, No. 5, November 1983, pp. 49-70.

25. Weinberger, P. J., The Version 8 Network File System (Abstract). Proceedings of the Summer 1984 USENIX Conference, Salt Lake City, June 1984.