A Client-Server Shell Architecture for Distributed Programming

by

Stuart Sechrest

Abstract

Networks of computers provide the opportunity for a user to run distributed applications and to distribute his workload among many available machines. Unfortunately, the design of current command interpreters, and in many cases of the underlying operating systems, often makes this hard to do. The design of remote execution and program management facilities is often based upon the assumption that programs run at a single site, and that they run as a single process, or have a restricted class of configurations. These problems can be corrected by providing a layer of distributed services between client programs and the underlying operating system. PPM, the Personal Program Manager, provides a service layer that can be used by both command interpreters and distributed applications. The PPM Services provide remote execution facilities that allow distributed programs to be run. They support a login session abstraction that provides location transparent bookkeeping and a common pool of name-value bindings in a session that can encompass many machines. They support a job abstraction that provides job control actions, such as termination, a means of expressing intrajob failure dependencies, and a pool of job-specific name-value bindings. These abstractions make the writing and running of distributed programs easier, while also making it easier to redistribute the work load.

The PPM Creation Service allows distributed jobs to be created and their communications configured. It allows these jobs to be suspended and terminated as units. The PPM Bookkeeping Service allows clients to monitor the work within a login session that may span many hosts. These services simplify the writing of programs that behave predictably in the face of machine failures. The PPM Environment Service allows programs to share name-value bindings with one another. This is useful as a simple means of interprocess communication, as a repository of information affecting future programs, and as a way of communicating to programs user-specific settings for values used by programs. The Environment Service provides a rich name space that allows bindings to affect processes on the basis of where they run, what program they run, from what command session they run, and so forth.

The PPM Services provide an interface that hides many of the details of the underlying system from their clients. They can be implemented on a variety of systems and their facilities used to support login sessions spanning machines running heterogeneous operating systems. Distributed jobs can thus take advantage of the facilities of different systems, without losing the ability to interact through a single interface. A prototype of the PPM Services for UNIX 4.3BSD is described, as well as the applicability of the PPM Services to other operating systems. Several issues related to the design of command interpreters for distributed login sessions are discussed, as well as a description of a prototype command interpreter.

TABLE OF CONTENTS

ACKNOWLEDGMENTS

I am grateful to many people who have supported, encouraged, and taught me in the time that I have spent in Berkeley.

I would like, first, to thank the members of my committee. Domenico Ferrari supported my work with great patience, providing guidance and time and energy. Felipe Cabrera and David Anderson provided valuable criticism of my ideas and of my writing. Stuart Dreyfus provided a helping hand at the end, for which I am grateful.

My thinking about distributed systems has been strongly impacted by working with the QuickSilver operating system at IBM Almaden Research Center. I would like to thank the members of the QuickSilver group, particularly Felipe Cabrera, who was instrumental in arranging for me to work with the group, Roger Haskin, and Marvin Theimer, who provided challenges and insights.

Of course, in graduate school one learns more from other students than from professors, and, so, I would like to thank those who taught me most of what I know. Although I can only begin the list, I would like to thank Doug Terry, Bart Miller, Songnian Zhou, Venkat Rangan, and Joe Pasquale. I would also like to thank Ramon Caceres, who helped in the preliminary development of PPM. I should not pass up the opportunity to thank Stefano Zatti, Wayne Citrin, and Mark Hill, who helped in many ways.

This work was sponsored by the Defense Advanced Research Projects Agency (DoD), ARPA Order No. 4871, monitored by the Naval Electronics Systems Command under contract No. N00039-84-C-0089. Support was also provided by IBM Corporation.

CHAPTER 1

INTRODUCTION

1.1. PPM -- the Personal Program Manager

Networks of computers offer an abundance of computing power and access to a wide range of resources, while at the same time presenting one with a number of unexpected, tricky, frustrating, but surmountable problems. One may want to run distributed programs, or to use the available resources more effectively by running programs on lightly loaded processors in the network. To make effective use of network resources, however, one needs support at a variety of levels. The ability to create and control processes throughout the network must, of course, be provided. It is important, in addition, to have provisions for cooperative distributed bookkeeping that is resilient to machine and communication failures. Running distributed programs need help in establishing communication and in detecting and handling failures. Support must also extend into the command interpreter; one should find it easy to run distributed jobs written in languages without tasking support, such as C. This dissertation explores an architecture for providing this support for distributed computation through a suite of distributed services and client programs. The support is offered, for the most part, by agents belonging to particular users, and is in that sense "personal." Together, these services and clients form the *Personal Program Manager*, or *PPM*.

PPM accomplishes the goals outlined above by hiding the boundaries between machines and by supporting new abstractions for multiprocess and multimachine programming. A user's work has an intuitive structure that we would like to support, but which is usually not directly supported by the operating system. It makes sense to think of a user's work as a series of *jobs* in a *login session*, rather than as a set of unrelated processes scattered among a set of machines. PPM builds upon the intuitive characteristics of jobs and login sessions to provide useful abstractions for distributed computing. A *job* is

- a process or set of interdependent processes engaged in a common task;
- an administrative unit that is suspended, resumed, and terminated as a whole;
- a bookkeeping unit for the user monitoring (or debugging) his work;
- a scoping unit for information passed into or exchanged within the set of processes.

A login session is defined not simply by the set of current jobs, but also by resources and state information preserved between jobs. A *login session*, then, is

- a bookkeeping unit representing all of a user's current work in a system;
- a collection of machines on which the user's processes can run without further authentication; and
- a collection of state information, which could include everything from working directories to default type fonts, used to set up new jobs.

1.1.1. The PPM Services

Supporting these notions of jobs and login sessions in a network of computers requires both new functionality and new mechanisms for old functions. Support for a user's work in previous systems has been supplied partly by the operating system, partly by command interpreters, or *shells*, and partly by the applications themselves. PPM takes the approach of providing a layer of *distributed shell services* between applications and the underlying operating system. This reassignment of responsibility has two motivations. First, the PPM Services are offered through a client-server interface to allow any application to take advantage of their support for distributed computing. Both command interpreters running distributed applications and the applications themselves can be clients of a standard set of services. This makes both types of program simpler to write and to maintain. Second, placing the services outside the operating system allows the specification of a system-independent service interface, while making the services easier to implement on a variety of systems including future systems. It is important for the services and the abstractions they support to avoid unnecessary system dependencies, since this allows login sessions and jobs to span the boundaries between heterogeneous operating systems. Spanning system boundaries is particularly important in a research setting, where jobs may take advantage of a wide variety of specialized machines and systems.

The principal PPM Services are

- *The PPM Creation Service*, which allows the creation, configuration, and control of user jobs;
- *The PPM Bookkeeping Service*, which provides clients with information about running and recently terminated jobs; and
- *The PPM Environment Service*, which offers clients access to a shared pool of name-value bindings.

Client programs access these services through a remote procedure call mechanism. The PPM Services simplify the task of programming distributed applications, offering facilities that would otherwise have to be written anew for each new application. They allow command interpreters to run distributed applications and to share load between hosts without limiting control over these jobs and without losing track of surviving jobs in the event of a host failure. They provide a framework for sharing information that simplifies communication within jobs, interactions between jobs, and the passing of information from command interpreters to the jobs they run.

Distributed programs have special needs. They can, for example, suffer partial failures. The PPM Services assist applications in detecting and handling such failures. They need to communicate. The PPM Services can hide many of the details of channel establishment and addressing, as well as allowing communication through the PPM Environment. They have special debugging needs. Developing distributed programs can require information about processes that might not be gathered or preserved by the underlying operating system. The PPM Bookkeeping Service collects and holds information that would not otherwise be available.

The PPM Services are useful for nondistributed programs as well. The PPM Environment Service, for example, allows programs to be more adaptable to the user's preferences. Information, such as the font for printing text in a window, has either to be hardwired into a program, passed into the program explicitly, or looked up by the program. The first choice is inflexible. The second bothersome to the user who must type the information (perhaps over and over) and to the programmer who must parse the type information. The third is supported by the PPM environment, which provides a simple and flexible way of passing information. With larger

networks, a greater variety of resources, and more flexible displays, adaptability to the needs of particular users is of growing importance, and *ad hoc* mechanisms for passing information should be replaced by a single, consistent approach.

Besides simplifying the programming of applications, the PPM Services support tools needed for login sessions spanning a network of machines, such as command interpreters, debuggers, and program monitors for distributed programming. These tools require the ability to start and control processes on both local and remote machines and access to accurate information about the user's work on all the machines covered by a login session. They also need ways of sharing information among themselves, so that more cooperative interfaces can evolve, using multiple bitmapped windows, menus, icons, and other techniques for interaction. Without such tools, load sharing will remain inconvenient, and distributed programming will remain the province of specialists (and will still be hard, even for *them*).

1.1.2. Providing the PPM Services

The PPM Services are provided by sets of agents (called *Local Program Manager agents*, or *LPM agents*) belonging to a particular user. There are several reasons for taking this "personal" approach.

- The most important goals of the services is to allow a particular user to write and run distributed computations. Providing mechanisms for distributing information about one user's jobs to other users was not considered important. Likewise, providing a job abstraction that would encompass jobs with components belonging to different users was not considered important.
- The underlying systems are presumed to be loosely coupled and not reliant upon the PPM Services. Therefore, the services can be provided on demand, and need only provide services to those users who wish to use PPM client programs. Separating the agents of different users also simplifies the programming of the agents.
- Providing separate agents for different users decreases the ability for one user to interfere with another. This independence allows the services to rely upon the cooperation of their clients. For example, agents can rely upon their clients to say when information will no longer be needed. This trust allows the servers to provide more assistance to client programs.

The management of the LPM agents serving a particular user is not the concern of client programs (although clients must supply the necessary information to authenticate the user). An initial set of agents is created when the user logs into a system. The set of agents for the user can expand and contract as the activity of its owner changes. PPM Service requests are sent to the local agent and, if the request requires action on other machines, they may be forwarded to agents on those machines. LPM agents for a particular user are created on additional machines as they are needed. The expansion of the session is thus not explicitly requested by the client, but is a side effect of a request for service on a new machine. When a user stops using a particular machine, the agents on that machine may terminate and the machine dropped from the session. Again, this contraction of the session is a side effect of the client's behavior, rather than the result of a request by a client. This is an important and useful type of transparency.

The PPM Services provide abstractions for jobs that are independent of those provided by the underlying system, and thus can be implemented on a variety of systems. (In the same way, the job abstraction for UNIX shells has proven implementable on a variety of systems.) Since the abstractions defined by the services are bound to have differences from those provided by any underlying system, we have therefore not felt constrained to match any particular system's

interfaces, identifiers, or other features. Some of the services described may require modifications to existing systems for full efficient implementation. For example, doing bookkeeping outside the kernel requires that a mechanism be available for the kernel to report events to the bookkeeping agent. Nonetheless, the sorts of modifications required are minor, and requiring them does not undercut the benefits of the PPM services.

An earlier version of PPM was described in [CSC86]. That version provided only job creation and bookkeeping. Experience with that implementation led to a broader set of responsibilities for PPM and more useful abstractions. A new prototype of PPM is described in this dissertation.

1.2. Related Ideas

PPM builds on ideas for login sessions, distributed programming, and job administration from many different sources. Some services from other systems must be extended to networks of computers; some must be made more general; some services should be offered in restricted versions because they are difficult to implement or are complicated to use in their full generality. The resulting design offers simple and effective support for distributed computing that improves upon that offered by other current systems.

The PPM Services are used to implement a command interpreter for use on a network of computers. This interpreter follows in broad outline the command interpreters, or shells, developed for the UNIX system [Bou86][Joy86][Kor83]. These shells have proven quite popular in the research community, and have been adopted on a variety of systems besides UNIX. The structure of the UNIX shells, however, makes them difficult to use for distributed computing. They are single process programs that use the local system to create local processes. Remote execution facilities can be used to run remote processes, but the mechanisms used to keep track of and control local processes do not easily extend to remote machines. PPM, therefore, changes the structure used by UNIX shells, making the interpreter a client of the PPM Services, rather than a direct caller to the underlying system. Services offered by the UNIX shells are generalized and relocated in the PPM Services.

One of the goals of the PPM Services' design is to hide the difference between local and remote operations. This has come to be called *location transparency*. A number of other systems have sought to hide remote operations behind location transparent kernel interfaces. The Newcastle Connection [BMR82], LOCUS [PoW85], and Sprite [OCD88] have all chosen this approach, to a greater or lesser degree. Calls to the local kernel in these systems may be forwarded to a cooperating remote kernel and acted upon at the remote site.

PPM does not require location transparency from the underlying operating system. The services can be implemented through a network of local agents using only local operations. This approach was taken because the PPM Services are intended to allow cooperation between dissimilar systems. Providing location transparency only at the level of system calls is inherently limiting, since it only allows homogeneous systems to cooperate.

Either approach to transparency will allow a system to use the UNIX shells to run programs remotely, but transparency alone does not address all of these shells' deficiencies in running multiprocess programs. The UNIX shells make assumptions about the form of multiprocess computation, supporting only a particular pattern of byte stream interconnections for communication. While the model has many useful applications on a single machine, other models are equally important for multimachine computations. PPM provides support for nonpipeline communication configurations, including the use of message-based communication. It follows a more flexible approach to job membership, allowing the implementation of command

interpreters offering command languages more suitable for running arbitrary distributed computations.

The job abstraction supported by PPM has antecedents in the process groups of both UNIX [RiT74] and the V system [ChZ85]. Both mechanisms allow operation such as termination and suspension of groups of processes. The UNIX mechanism assumes that all processes in a group have a common ancestor. The V mechanism is part of a general purpose multicast message system, and hence can be used to deliver arbitrary messages to group members. The V mechanism is more flexible, but is currently used to implement the UNIX shells' job control mechanisms on the V system.

Jobs in PPM, however, are more than a grouping for control operations. They define scopes for communication and information sharing through the PPM Environment Service. Job membership in the UNIX shells is tied to the pipeline job model. In PPM other more flexible job models are supported. Another aspect of the job abstraction supported in PPM is the interdependencies of the processes within multiprocess applications. The failure of one process may require action to be taken to inform or terminate another. The handling of failures has not been addressed by shells, but has been addressed by transaction management systems.

Transaction systems can be helpful in implementing jobs. Transaction management, however, is not identical with job management, and is not included at the proper level in many systems. PPM therefore uses some ideas suggested by transactions, but does not require them of the underlying system.

Transactions are an abstraction for activities that are well-behaved in their use of resources. This means, among other things, that the resources used in the activity are always left in an acceptable state in the case of failure. The QuickSilver system [HMS88] uses transactions to manage the resources associated with particular processes. These transactions could be nested withing a job transaction, with the processes viewed as resources of the job. The QuickSilver system, however, does not support nested transactions explicitly. Even with support for nested transactions, it is necessary to manage the resources, that is the processes, involved in a transaction. The interdependencies of processes can be complex and changeable. Therefore the management of these processes may not be amenable to fixed policies. One alternative is to include code in the application to handle possible failures. PPM takes an alternative approach, allowing the application to specify rules for managing process failure, such as requesting that the job be terminated if a particular process exits abnormally, but leaving the implementation of these rules to the PPM Services.

The PPM Environment Service provides a shared pool of name-value bindings. The UNIX shells use environment bindings to pass information from a command interpreter to programs started by that command interpreter. The simple inheritance provided by these shells is of limited use, however, particularly for distributed programming. For example, when the command interpreter may run on a different machine than the program it starts, and when the program may, in fact, be distributed across many machines, machine dependent information cannot be passed. PPM has taken the approach of providing a richer name space for the environment. The rules for setting and looking up bindings allow a process to see an appropriate set of bindings, even when they are specific to a particular machine or application. PPM also allows bindings to be shared between processes, providing a simple means of interprocess communication.

1.3. Plan of the Dissertation

The remainder of this dissertation discusses in detail the PPM Services and a prototype implementation for UNIX 4.3BSD. Chapter 2 discusses the support provided for multiprocess

programming and job administration by current operating systems, configuration languages, and command interpreters. Chapter 3 describes the architecture of PPM. Chapter 4 specifies the interfaces of the PPM Services. Chapter 5 is concerned with the management of a network login session, in particular discussing identities and authentication in a distributed system. Chapter 6 describes in detail the PPM Environment, and shows how it can be used to support distributed computations. Chapter 7 describes how user interfaces can act as clients of the PPM Services, and presents features of command languages that can be used to support distributed computations. Chapter 8 presents a prototype PPM built for UNIX 4.3BSD and assesses its performance. Chapter 9 describes the applicability of the PPM architecture to operating systems other than UNIX, in particular, to the QuickSilver, Mach, and V operating systems, and discusses the role of PPM in a heterogeneous network. Finally, Chapter 10 provides concluding remarks on PPM's design.

CHAPTER 2

SUPPORT FOR MULTIPROCESS PROGRAMS

2.1. Means of Support

There are a number of types of support that a user might wish for multiprocess and distributed programming. These include

- support for process creation and control
- support for configuring communication
- support for job control
- support for information sharing
- support for debugging

Support can be offered in a number of different ways, with a number of tradeoffs between generality, simplicity, ease of use, and the portability of application code making use of the support. Support can come from an operating system kernel or system server processes, from a programming language runtime system, particularly those of configuration languages, and from interactive command interpreters. PPM offers its support through library routines and services contacted through remote procedure calls.

In this chapter, we examine the support given to multiprocess and distributed programming by some existing operating systems, configuration languages, and command interpreters. This review provides a context for understanding PPM's facilities. We describe the UNIX, MACH, V, and QuickSilver systems here. In Chapters 8 and 9 we describe how the PPM facilities augment these systems. We briefly review some other projects involving job configuration, to provide a setting the design decisions made in PPM. We do not discuss language constructs in general purpose programming languages for supporting distributed computing. A good review of these constructs may be found in [Sco84]. We describe some of the strengths and limitations of command interpreters originally designed for UNIX and now adopted on many other systems. This is useful for understanding many of the issues discussed in Chapters 5 and 7.

2.2. Threads, Processes, and Jobs

We must now explain what is meant be several terms that we shall be using. These include the notions of *thread*, *process*, and *job*. These are abstractions offered by operating systems. In attempting to explain them, we must realize that we are attempting to find commonality among the abstractions of different systems, rather than necessary and sufficient conditions for each term. The following definitions, therefore, appeal to intuition, but cannot capture all the subtleties and variations that arise in different operating systems.

Most modern operating systems provide improved performance by interspersing the executions of a number of different programs. This lowers the expected time to completion, as well as allowing the machine to avoid sitting idly when waiting for external I/O. The computer itself may well be playing similar tricks by interspersing microinstructions from different instructions at the architectural level. Just as the machine architecture provides the compiler with an idealized model of its units of execution, to which the hardware conforms, the operating

system provides an idealized model of its units to the programmer. We will call these units *threads of control*, or simply *threads*.

As just described, threads in execution are interspersed by the scheduler of the operating system. It is also possible, however, for a runtime system operating within a system-provided thread to multiplex that thread among a second set of threads. Just as the operating system provides a particular idealized model to this runtime system, the runtime system would offer its own, user-level, idealized model. The user-level thread model might differ in some ways from the system-level model, but they are fundamentally similar.

Besides providing efficient performance, threads, particularly user-level threads, serve an additional purpose. Programs that perform a number of different tasks can often be made simpler by dividing tasks between a number of separate threads. We will refer to this as *multithread programming*. It is especially important in distributed systems, where, because tasks must be performed at different sites, this mode of operation is forced upon the programmer.

Just as the execution sequence of a machine is multiplexed between threads, the resources of the machine are multiplexed between a number of different *virtual machines* or *processes*. The term virtual machine is usually reserved for systems that attempt to provide a highly detailed model closely matching the hardware of a real machine, while the term process is used for systems providing a more abstract model. Typically, a process has an abstraction for memory, called an *address space*, and abstractions for I/O devices, which we will call *communication descriptors*. Some systems associate a single user-level thread with each process, while others allow a variable number of threads. Allowing a process to have more than one thread allows the threads to cooperate with one another with a simple means of communication through memory. A process's address space and its communication descriptors may be private, or they may overlap with those of another process.

A *job* is a term for a collection of processes. Usually these processes are cooperating in some endeavor. The rules for the composition of a job can vary considerably between systems, if the systems support a job abstraction.

A thread, process, or job can have a state, such as *not yet run*, *runnable*, *suspended* or *terminated*. by *runnable*, we mean that the underlying scheduler is free to cause it to continue executing, even if it is not actually executing at some given moment. By *suspended*, we mean that the thread is not eligible for execution. We will say that changing the state of a thread is a *thread control* action. A state change operation affecting a whole process is a *process control* action, and changing the state of a job is a *job control* action. It is up to an individual system to specify what the hierarchical relationships will be between actions at these levels.

2.3. Operating System Support

2.3.1. UNIX Processes and Process Groups

The UNIX system [RiT74] was originally developed for operation on a single machine, but has been extended for use in a network of machines. These extensions include

- facilities for network communication [LFJ86],
- network services, including remote execution facilities [BMR82] [AgE85], and
- location transparent operation [PoW85].

Moreover, the UNIX system interface has heavily influenced or been adopted by many more recent systems oriented towards network computing (Sprite [OCD88], for example). UNIX offers support for particular forms of multiprocess computations. This support, however, may be difficult to extend to a network, and the model itself may be insufficient or inappropriate for

distributed jobs. An alternative approach, taken by PPM, is to provide network services with abstractions and semantics somewhat different from those provided by UNIX.

UNIX processes are single-threaded and do not share writable portions of their address spaces. Multithreaded computations are therefore executed by a set of processes interacting through interprocess communication. UNIX I/O operations are performed on a process's descriptors, which may represent a file, a device, or a communications channel. Multiprocess computations are supported by I/O configuration operations that allow byte stream communication channels to be created between processes before the computation starts. This allows I/O to be performed on a preconfigured descriptor without requiring that the program know the ultimate source or destination of the I/O.

The way that UNIX supports configuration is to split the creation of a new process running a new program into two steps. A parent process creates a child by calling *fork()*. The new child shares its parent's I/O descriptors, and its address space duplicates that of its parent. Since the same code is loaded into the new process, the child continues to run the same program as its parent. The child *process* therefore performs configuration operations under the direction of the parent *program*. To run a different program, the child must call *exec()* specifying the program to be loaded. Following this call, the process resumes execution at the new programs main entry point with its I/O configuration arranged.

This way of running new programs is one area of difficulty in extending UNIX semantics across a network. There are three points that should be noted here:

- Taking two steps to run a new program can be quite wasteful if the parent's address space must be copied for the child, only to be almost immediately overwritten. On a single machine, it may be possible to avoid most of the copying, but copying is much harder to avoid if a child is created on a remote machine.
- Child processes inherit access to I/O descriptors from their parent process. This is hard to implement if the child runs on a different machine than does its parent (it has been implemented for LOCUS, a distributed version of UNIX [PoW85]).
- Under UNIX, a process's I/O descriptors can only be manipulated by that process. This is acceptable because, by splitting program creation into two steps, UNIX allows the parent program to configure the child process. This functionality can be provided by much more restrictive mechanisms that are simpler to implement across a network.

These semantics are difficult or wasteful to reproduce when the parent and child run on different machines. It is not surprising, therefore, that systems like Berkeley UNIX choose to support distinct remote execution facilities. (The remote execution facilities offered by Berkeley UNIX are described later in this chapter.)

UNIX processes can be suspended, resumed, and terminated by the *kill()* system call. This call "sends a signal" to a process or to a *process group*. A process group is a collection of processes sharing a common tag. Typically they are all descendants of a process that was marked with the tag by the command interpreter. A process group is used as a job abstraction for the purposes of job control.

There are two difficulties with using the process group abstraction in a network. First, the identifiers used for processes and process groups are not well suited to identifying objects in a network. Identifiers that are valid throughout a network can be provided in a UNIX-like system by using *<Machine-Id, Process-Id>* pairs for processes, and a corresponding pair for process groups [PoW85]. Some systems have used a network-wide file system naming scheme, to provided an alternative means of naming [CuW84] [Kil84]. Second, process groups, as such, are

simply tags on processes, and are too weak an abstraction for job control in a network. Mere tags provide no way of keeping track of where in a network processes belonging to a process group might be running. Machine failure, coupled with inadequate bookkeeping, can leave portions of multiprocess computations running without mechanisms for finding out that they exist.

There are problems with UNIX support of multiprocess programming even if one leaves aside the question of distribution. There are, for example, many ways in which UNIX does not address very well the problems of debugging multiprocess programs. The state of a process, and notification about changes in the state are made available only to the process's parent in most versions of UNIX. (This has changed in some newer versions, for example AT&T Bell Laboratories' 8th Edition UNIX [Kil84].) When debugging a distributed program, information about a process that has already terminated, such as it's communication configuration, running time, or exit status, may be relevant. This information is not kept by the UNIX kernel, nor is it necessarily the proper role of the kernel to do so.

Another area where the support provided by UNIX has proven too weak is in the configuration of I/O. UNIX I/O was originally entirely byte-stream oriented. This common abstraction for I/O allows a process to perform many operations, such as reads and writes, on descriptors without regard to the object represented by the descriptor. Thus, the same code can be used to process input read from a file, from a terminal, or sent by another process through a communication object. The objects present a byte stream interface, which allows programs to read and write arbitrary numbers of bytes, rather than enforcing a record or message structure on data accesses.

The usefulness of a common abstraction for communication descriptors is somewhat dependent upon the uniformity of semantics for the various objects represented by the descriptors. Berkeley UNIX [LFJ86] has introduced communication objects with non-byte stream semantics. In particular, it provides sockets that send and receive messages. Depending upon the protocol used, the delivery of messages sent may be reliable or unreliable. Because of the different semantics for these objects, even though the system calls used on their descriptors are identical, the code that uses the descriptors must take into account the semantics, and so cannot be used with byte stream descriptors. Multiprocess jobs that communicate using messages need different configuration support than jobs using byte streams. This support requires new mechanisms.

2.3.2. Threads, Tasks, and Ports in MACH

The MACH system [BRS86] [Ras86] is a variant of UNIX that has advanced a number of innovations in process structure and in interprocess communication. MACH, however, has not innovated in the area of *job* abstractions. It continues to rely upon UNIX process groups and upon the UNIX shell facilities (see Section 2.5.1). MACH provides a more complex process model than UNIX by allowing multithreaded processes, and the sharing of address space segments. MACH uses the term *thread* in the sense that it is used here, but uses the term *task* to describe the machinery of a process. We shall use the term *process* to refer to a MACH task.

MACH separates the creation of threads from the creation of processes [BBB88]. Processes are created without any threads. Later threads can be added to processes. Newly created processes derive their address space from that of their parent as in UNIX, with the exception that memory segments can be shared with the parent, rather than simply copied. Processes are also able to restructure their address spaces by mapping in files or new segments.

MACH processes communicate through message *ports*. These ports have local addresses that can be mapped to addresses valid across networks [SJR86]. Access to these ports are

controlled through capabilities. One process holds the right to receive messages at a particular port. Other processes can have the right to send messages to this port. Unlike UNIX descriptors, MACH ports can be created by one process for another. This makes it possible to configure the I/O of suspended processes.

2.3.3. Processes, Teams, and Process Groups in V

The V system [Che88] is a message-based distributed operating system for workstations connected by local networks. It consists of a kernel and a distributed collection of servers. The kernel provides threads and processes, with support for multithreaded computations within nonoverlapping address spaces. The threads sharing an address space are sometimes referred to as a *team*. The kernel also supports both single-recipient and multicast interprecess communication (the latter is described in [ChZ85]).

Program initiation in V does not follow the two-step *fork/exec* model of UNIX. Instead, processes can be created and their I/O configuration manipulated by the parent process before the computation is set running. It is much easier to hide system boundaries when following this model. To initiate a program execution in V, a client sends a request to a program manager (also called the *team server*) to create a new process, loaded with a specified executable image. The client can contact either a local or a remote program manager. The program manager uses kernel routines to create the process and load the image, and to create a thread within the process. The client can load the new process's address space with arguments, environment variables, and default I/O information. The client then releases the new thread, and independent execution begins.

Interprocess communication in V is based upon synchronous message passing. Message recipients are addressed directly. There is no logical abstraction, such as a Berkeley UNIX socket or a MACH port used for communication. Since threads are not assigned identifiers until runtime, the V kernel provides a mechanism for registering and looking up services by name. When a client wishes to find the address of a service, it makes a call to the kernel, specifying a logical identifier, and returning a thread identifier [ChM84b]. Byte-stream I/O can be performed by mapping read and write operations onto a set of message exchanges [Che87]. Information about open files, pipes, devices, and so on, is held within an address space. Thus, the address space, while not having system supported file descriptors as in UNIX and MACH, contains similar configuration information. This information can be provided by the parent process as part of the initialization described above.

Job control is done through messages to the program managers. An operation such as *DestroyProcess()* can be performed on either a thread or a *process group*. In V, a process group is a set of threads that can be addressed through multicast communication. The command interpreter can see to it that all threads in a job are part of such a process group. Process control uses the "co-residence" feature of group communication to send to the members of the program manager group that are co-resident with (on the same machine, and hence in control of) either a specific process or members of some process group.

In contrast to single-recipient communication, multicast communication provides only best-effort delivery [ChZ85]. The membership of a group is not centrally managed. Instead, each kernel is responsible for maintaining a list of groups to which each local threads belongs. Delivery of multicast messages is guaranteed only through explicit management by the sender. Group membership may be obtained by multicasting a query, to which the various kernels respond. Any message losses, however, will result in only a partial membership list being returned. Without central management of group membership, process control is built on an optimistic base that cannot guarantee that orphan processes will not be left running.

The program manager acts as an *exception server* for local threads. The manager receives notifications of exceptions and terminations. The manager also collects information on programs and their resource consumption. Other servers do not generally receive notifications of thread termination. This is somewhat a philosophical decision. Servers are made responsible for checking for the continued existence of clients. Cheriton [Che88] states that "in our experience, the garbage collection code [that is code to clean up internal state associated with failed correspondents] of the servers is not significant and the garbage collection overhead is minimal." Thus no provisions are made by the system for assisting distributed jobs in detecting and handling failures.

2.3.4. Processes and Transactions in QuickSilver

The QuickSilver system [HMS88] is a message-based operating system developed at IBM Almaden Research Center. The processes offered by QuickSilver are not unusual; they are single-threaded and have private address spaces. Multithreaded programs can be constructed using a coroutine package. QuickSilver's largest innovation with respect to program management is the way that it has integrated program execution with a transaction-based recovery manager.

Embedded in the QuickSilver system is the notion of a *transaction*. Transactions are derived from the database notions of atomicity, serializability, and permanence. IN QuickSilver, transactions are logical entities managed by a Transaction Manager that detects failures and supervises a voting protocol for transaction termination. A transaction can terminate with either a commitment or an abortion. The voting protocol ensures that all participants see a consistent result for the transaction. Transactions are used to coordinate the recovery of resources held by different managers on behalf of some operations.

Among the operations represented by a transaction is the execution of a process. Each process has a *default transaction*, used in managing all resources held on a process's behalf. These resources include the address space segments, which are managed by a virtual memory manager, open file connections, which are managed by a file system, open windows, managed by a window manager, and the record of the process's window, managed by a process manager. The termination of the process can be regarded as being equivalent to the termination of its default transaction. Resource managers are informed of the termination os the transaction using the protocols described in [HMS88]. The process's default transaction may be either committed or aborted. For managers of permanent resources, such as the file systems, the distinction may be important. For managers of volatile resources, such as window managers, the distinction is irrelevant.

QuickSilver transactions make the management of resources by servers easier. By associating a default transaction with every process, they guarantee that all resources held in for a computation can be associated with some transaction. Multiprocess computations, however, require multiple transactions. Moreover, individual processes in a multiprocess computation can be regarded as resources of the overall job. These resources require a manager that takes account of their interdependencies and handles partial failures appropriately. PPM can act as such a manager, as will be discussed in Chapter 9.

2.3.5. Some Other Systems

Switchboards and Name Servers

To facilitate communication between unrelated processes, many systems employ some form of *switchboard* or *name server*. A switchboard can be used in systems that allow the endpoints of communications channels to be passed from one process to another. A switchboard

allows a program to register a channel endpoint under some name. A second program can then request that endpoint by name and get a communication channel with the first. In systems that do not allow endpoints to be passed, a program can register a contact address with a name server. A second program can then look up the address by name, and use it to it to communicate with the first. Name servers can hold other information as well, allowing the exchange of a wide variety of information.

DEMOS [BHM77] was an early system employing a switchboard process. Each process in the system has a connection, called a *link* in DEMOS, with the switchboard. The DEMOS link abstraction allows a link endpoint to be passed across another link. The recipient of he endpoint can forward it across yet another link. The switchboard is, thus, in a position to compose communication channels between any two processes. Processes in DEMOS register links with the switchboard using simple, unstructured names.

Switchboards and name servers can be made more powerful in two ways. First, they can offer their services throughout a network, either through remote access or through distribution. Second, the naming scheme offered can be made more powerful. In implementing a remote procedure call mechanism for a distributed environment, Birrell and Nelson [BiN84] used the Grapevine name server [BLN82]. Grapevine serves a potentially large network of machines, providing access to a replicated database of name-value bindings. The name space is organized as a two-level hierarchy, with simple names placed within particular registries. The registries correspond to organizational or geographical divisions. The remote procedure call stubs use Grapevine in two ways. They bind the address of the procedure call provider to an instance name and a a list of instance names under a generic name. A client stub can find the call provider by looking up an instance name, or if that name is unknown, by looking up the generic name and then the instance name. The registry names do not play a significant role in this strategy, although through them searches for providers can be limited to local machines.

Resource Sharing

Dannenberg [Dan82] [DaH85] introduced the notion of a *butler* process on a machine that would act as an agent offering local resources to remote clients, and as a manager of remote processes acting on behalf of local clients. The butler is an entity owned and trusted by a machine's owner, which limits the activities of *guests* on the local machine. This is different from allowing the guests to login on the local machine, in which case there is no control over the resources that guests might consume. The butler belongs to a particular owner, and, given a resource specification by a local client belonging to the owner, will locate resources and invoke services on the client's behalf. An owner's butler, then, allows him to create and control remote or distributed jobs through interactions with a local entity. Butler processes of this type were used by Theimer to implement process migration and remote execution facilities in V [The86].

2.4. Configuration Language Support

Configuration and communication are two aspects of multiprocess programming that in some programming languages are made quite distinct and in others are blurred together. In object-based programming systems, such as Emerald [JLH88], the process structure underlying the object instantiations can be, for the most part, hidden from the user. Emerald provides language primitives to fix the location of an object at a particular processor, or to force the location of two objects on the same processor. Outside of this, however, configuration need not be a concern of the user. Other languages, however, have a model much closer to an operating system's process abstraction. In these languages, the management of a program configuration, that is the number of servers, their locations, and their configuration for communication, is a

direct concern of the user. Older languages, such as C [KeR78], have no constructs for configuration and communication. Therefore, programs either make use of operating system services, or rely upon a distinct configuration system that provides their initial configuration. Configurations are expressed in languages designed for interactive use, or in noninteractive configuration languages.

PCL [LSB79] is a configuration language allowing the specification of a job through a hierarchy of parameterized templates. The templates include shared memory segments and communication links with a variety of semantics. The communication links include simple, multiplexing, and demultiplexing links, as well as links providing synchronization semantics, for example a demultiplexing link that will block the recipient until messages arrive from all members of a set of senders.

The TASK language [JoS79] is a configuration language for describing distributed programs to be run on the Cm* multiprocessor. The language allows the description of programs through parameterized templates. An important concern of the language, which derives from the asymmetries of the Cm* architecture, is the constraints on the placement of the component threads that will ensure more efficient execution.

DPL-82 [Eri82] is a configuration language used to describe a dataflow network, where each node is a process executing a (single threaded) Pascal program. The dataflow network can be distributed across many machines. The programmer describes a set of nodes and the arcs between each node, and provides the initialization code and a sequence of activation conditions. The runtime environment supplies protocols for process instantiation, interprocess communication, and certain forms of job control. The activation conditions provide the overall sequencing of the program execution. This includes the termination of the entire job, testing for process failure, and a simple form of restart, where a failed process will be replaced by a new process, perhaps on a different machine.

PRONET [LeM82] is a language system that decouples the programming of individual processes from the programming of the program configuration. The configuration language allows the specification of a static initial configuration and a (static) list of actions to be taken when certain events occur, or are "announced" by a process. These actions can include reconfiguration and termination. The network events detected by the runtime system do not include process or processor failures, although these could be announced by a process.

CONIC [KrM85] is a configuration language oriented to supporting the configuration, and the dynamic reconfiguration, of long running distributed programs. The language allows the specification of an initial configuration and the later addition of modifications to the configuration. Program instantiation and modification is done by a distributed set of agents. The language system has been used to support fault tolerant systems by adding modifications to the system specification when failures occur.

2.5. Command Language and Command Interpreter Support

2.5.1. The UNIX Shells

A number of shells have been created for various versions of UNIX, including three popular shells commonly referred to as the Bourne Shell [Bou86], the C Shell [Joy86], and the Korn Shell [Kor83]. Although these shells differ slightly in their syntax, their built in functions, and in other details, they are fundamentally similar, and will be referred to collectively as the *UNIX shells*.

2.5.1.1. The Pipeline Job Model

The UNIX shells are command interpreters that execute a series of commands. The command language allows these commands to be issued successively or to be embedded in *shell programs* using includes variables, expressions, and flow-of-control constructs, such as loops and conditional statements. Shell programs saved in files are called *shell scripts*. The user can type commands or shell programs to an interactive shell, or submit a script to a noninteractive shell.

The basic command for the UNIX shells is called a *pipeline*. This term can refer to the command typed to the shell, the logical structure of the job it invokes, and to the set of processes that realize this job (see Figure 2.1). Textually, a pipeline is a set of program invocations connected by vertical bars. A program invocation is a program name and a set of arguments. For example, the command "ls | more" is a pipeline invoking the program *ls* and the program *more*. Logically, the programs invoked by a pipeline command are a set of *pipe segments* connected by *pipes*. Intuitively, the output of one pipesegment flows unidirectionally through a pipe, serving as input to the next segment. Thus, in the pipeline invoked by the command "ls | more," the output of *ls*, which lists a directory, flows to the program *more*, which prints it on a terminal or window, one screenful at a time. This pipeline is realized, on the Berkeley UNIX system, for example, as two processes running the code found in the files */bin/ls* and */usr/ucb/more*. The first process is connected to the second by a byte-stream connection. The endpoints of the connection are descriptors attached to the two processes.

Logically, each pipe segment has one input (the *standard input*, or *stdin*) and a principle output (the *standard output*, or *stdout*). The segment also has a secondary output (the *standard error*, or *stderr*), intended, generally, for diagnostic or error messages that should be sent immediately to the user's terminal screen, rather than to some other process. Typically the standard input of the first segment of a pipeline, and the standard output of the last segment, is connected to the user's terminal, as shown in Figure 2.1. The standard outputs, not shown in the figure, generally will all go to the terminal.

The UNIX process that instantiates a pipe segment can explicitly specify a file descriptor to use as an input or an output when performing I/O. The C I/O library, however, provides routines that support the model of a read-only standard input, a write-only standard output, and a write-only standard error. The library uses particular descriptors consistently for the standard input, standard output, and standard error and defines constants for these descriptors. The UNIX shells ensure that each process created has these three standard descriptors, following the C I/O library's conventions for their use. By default, the standard input for a pipeline (that is, *stdin* for the first segment of a pipeline) is read from the shell's terminal. The standard output of the last pipe segment is written to the terminal.

The pipeline's standard descriptors can be explicitly manipulated by the UNIX shells. This is called *I/O redirection*. There is syntax agreed upon among the UNIX shells that allows a pipeline's standard input to be read from a file (including special device files, such as terminals and tape drives) and its standard output to be written to a file. The handling of other descriptors is not agreed upon. The Bourne shell and the Korn shell allow the manipulation of descriptors by number. This allows the manipulation of the standard error, as well as of other descriptors. The descriptors can be connected to a file (as either a source of input or a destination of output), or can be made to refer to the same file or pipe as some other descriptor. The latter can be used to allow the standard error of a pipe segment to be combined with the standard output as input to a file or to the next segment. The C shell has syntax for this specific purpose, but does not support manipulation of descriptors other than the standard error.

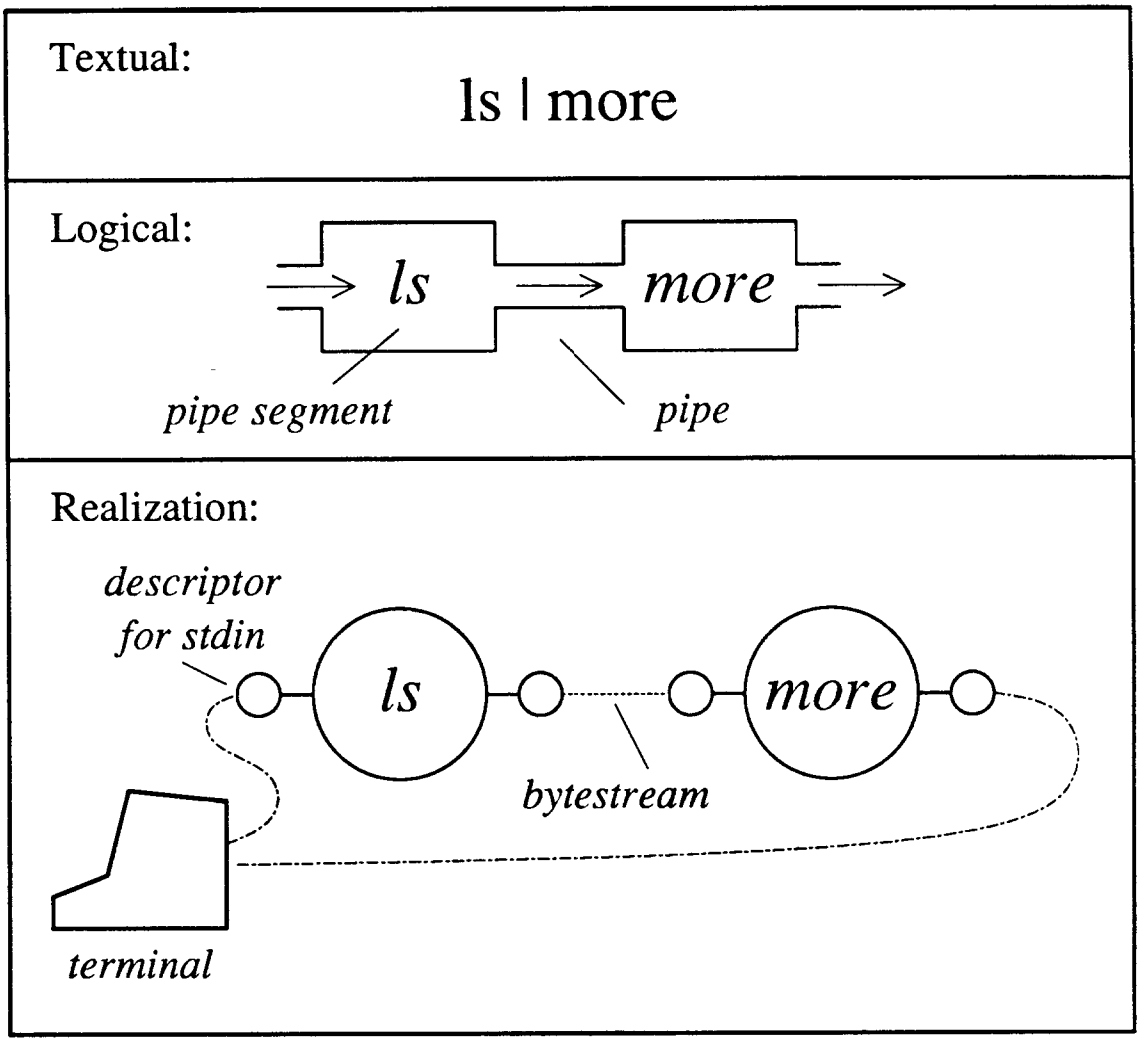


Figure 2.1 A pipeline for the command "ls | more"

The pipeline model has shown itself to be a very useful model in UNIX. Most commands are pipelines of only a single segment, and are instantiated as a single process. Many programs in the UNIX system, however, operate as *filters*, reading from *stdin*, performing some action (for example, a string search), and writing output to *stdout*. In some cases several filters are written to work together as a complete application. The passes of a compiler, for example, operate on the output of the previous pass. The passes may be easier to write as separate programs, which are connected in a pipeline configuration. Filters can also be used in pipelines to select optional processing steps in some applications. For example, the **troff** text processing system allows the user to select options such as adding diagrams or equations to a paper by inserting pipe segments. Pipelines can also be used to construct new applications for some special purpose from "off-the-shelf" parts. One might, for example, use a string manipulation program in one segment to fashion from its input arithmetic expressions to be sent to a calculator program.

2.5.1.2. Support for Nonpipeline Jobs

The pipeline model has been used for a wide range of applications, but it is not a flexible job model. Multiprocess computations can have organizations other than pipelines: data may not have a unidirectional flow, segments may have numerous inputs or outputs, or the communication may not fit a byte stream pattern. The UNIX shells, however, do not support job models other than the pipeline.

On systems supporting named pipes, such as LOCUS [PoW85], it is possible for the UNIX shells to produce combinations of processes more complex than pipelines. I/O redirection through a named pipe can be used to connect an output from one pipeline to an input from another. The resulting complex of pipelines, however, is not a single job, since it is not treated as such for the purposes of job control.

The Bourne shell and the Korn shell support the opening of input and output files for descriptors in addition to those for standard I/O, as well as supporting the creation of pipelines with closed standard inputs or standard outputs. These extra inputs and outputs could include connections to named pipes, if named pipes are available. These descriptors are identified by number, with no naming scheme or other logical indirection.

The shell does not convey to the program executing in each pipeline segment information about the I/O configuration of the segment, even though this configuration may be nonstandard, including, for example, extra inputs or outputs. Most programs are written to read from and write to the standard I/O descriptors with the assumption that each of these descriptors will be attached to an I/O object with byte stream semantics such as a file, a pipe, or a byte stream socket. If the program uses additional descriptors, which the shell creates, there must be agreement between the program and the shell about which descriptors are to be used for which purpose.

Given the availability of I/O objects with message-based semantics, it is possible that shells might provide support for opening descriptors with different types of semantics. This is not provided in current UNIX shells. However, if support were to be extended to non-byte stream objects, the choice of semantics for each descriptor would be another piece of information that a program might obtain from the shell.

2.5.1.3. Job Control

Job control refers to actions that the shell can take affecting a job, including suspending, resuming, or terminating them. In the UNIX shells, a pipeline is the unit of job control. Job control is exerted through software interrupts, or *signals*. All processes have a process identifier and a process group identifier. Signals can be sent either to a particular process or to a process group. The processes of a pipeline are all placed in the same process group. This allows them to be sent signals as a group. Through signals, a running job can be suspended or terminated and a suspended job can be resumed.

In UNIX, a child process, by default, belongs to the same process group as its parent process. This has the fortunate effect of allowing a pipe segment to create child processes subject to the same process control signals as the parent. Thus, the shell need not and, indeed, does not know that a new process has been added to the job. Instead, it relies upon the kernel to propagate signals to all the members of a process group.

Reliance upon process groups for job control poses problems in distributed systems. In LOCUS [BuM85], processes can be created at remote sites or migrated to them. To allow the delivery of remote signals, each process group identifier includes an identifier for a synchronization site. This site keeps tack of all sites where members of the process group exist. This site

must, therefore, be informed of all remote process creations and process migrations. Signals to process groups are sent first to the synchronization site, and from there to all member sites. If the synchronization site is for some reason unavailable, a ''surrogate'' synchronization site is used, which can use broadcasts to figure out where members of the process group lie. In NEST [AgE85] remote processes are represented at the home machine by an agent process. Each process's process and process group identifiers encode the home machine of the process. On a machine other than the home machine, the signals are forwarded by the kernel to the home machine. If the target of a signal is an agent for a remote process, the kernel forwards the signal to a server process on the remote machine, where it is delivered to the remote process. In SPRITE [DoO87] processes can be created locally and then migrated to a new machine or a new process can be created remotely. In either case, however, the home machine retains a representation of the remote process. Signals to the process are processed at the home machine, and forwarded to the remote machine. These approaches allow the UNIX shells to create remote jobs, or jobs that span machine boundaries, without substantial changes to the shells. These approaches, however, require the cooperation of the kernels on the different machines to propagate signals between them. It is, therefore, not a general strategy for handling distributed jobs.

A shell runs a succession of jobs, either ''in the foreground'' or ''in the background.'' A foreground job is run synchronously, that is no further commands are executed by the shell until the job is suspended, terminated, or exits. Thus, there can only be one foreground job. This job is allowed to read input from the terminal. Background jobs are started without waiting for their completion. These jobs are not allowed to read from the terminal. The C shell and Korn shell allow a suspended job to be resumed in either the foreground or the background and a background job to be switched to the foreground.

The notion of foreground and background has two components: (1) access (or lack of access) to the terminal input and (2) synchronous (or asynchronous) execution. When all interactive input and output is through a single terminal, these two components are closely connected. With window interfaces, however, the two are somewhat more distinct, since new terminal windows can be easily created. For example, a new terminal window could be created whenever an asynchronous job is run. This job would have exclusive access to the new window, thus, it would have a source of terminal input, and its output would be kept separate from that of synchronous processes.

2.5.1.4. The Environment

The *environment* is a list of name-value bindings that is passed to an executed program in the same way as a normal argument list. When the UNIX shells are started, they inherit an environment. The shell makes copies of the environment bindings, these copies are called *parameters*. This separates the bindings known to the shell into two compartments, bindings passed on to the programs the shell starts, and bindings kept only within the shell. The latter can be used used by the shell as variables in shell scripts. Changes to their values will have no effect on the environment inherited by executed jobs, unless a command is issued to copy a value from a parameter to the environment.

The environment is not extensively used by the UNIX shells. For example, a typical environment for the C shell contains seven entries:

HOME -- the user's home directory
MACH -- the user's home machine
PATH -- the user's execution path
SHELL -- the file name of the shell

TERM -- the terminal type
TERMCAP -- a list of the terminal control strings
USER -- the user's login name

Note that this information includes static information about the user and information about the login session. Other environment bindings are used to hold information specific to a particular program; for example, EXINIT holds a list of startup commands to the editor. Bindings are also used to indicate defaults to programs based on user preferences; for example, PRINTER contains the name of the default printer for the commands to print files, or to query the printer queue. Most applications do not use bindings for these purposes; thus, the examples are more representative of how the environment might be useful than of how it is actually used.

2.5.2. Remote Execution Facilities

Berkeley UNIX provides a number of network services, including a *remote login server* and a *remote execution server*. The remote login server allows the user to create a new shell that performs its I/O through a network connection, rather than through a terminal. The remote execution server allows the creation of a process in response to a request that arrives across the network.

The remote login server allows the user to log into several machines simultaneously. For each login, a separate shell is created. From an original, or *root*, shell, the user can login

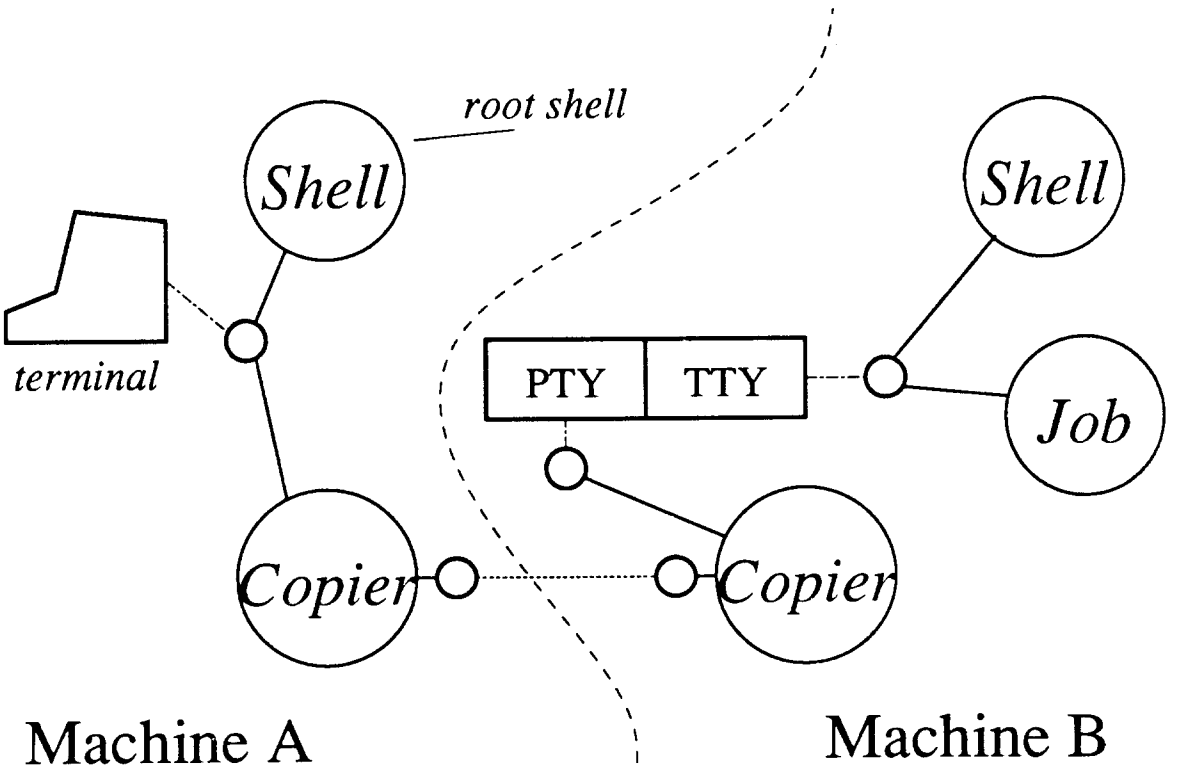


Figure 2.2 A remote login session created through *rlogin*.

remotely (''*rlogin*'') to other machines (see Figure 2.2, where processes are represented by large circles, and I/O descriptors by small circles). On the local machine, Machine A, the shell runs a program that first sends a request to the remote login server on Machine B, and then copies terminal I/O back and forth across the machine boundary. On the remote machine, the remote login server creates a shell process and a copier process. The copier is connected to the other copier through a two-way byte stream channel. The shell is connected to the copier process not directly, but through a pseudodevice (a teletype/pseudoteletype pair, or TTY/PTY pair) that allows the shell to treat its input source exactly as it would a real terminal. (In other versions of UNIX, theses pseudodevices have been dispensed with. Instead, a more general communications facility allows the shell and copier to be connected by a channel that provides the appropriate terminal protocol processing [Rit84] .)

It is possible to login to still more machines, either be running the *rlogin* program on the remote machine, or by suspending the local rlogin job, that is the local copier, and starting another. The resulting shells form a tree. If one runs remote logins from other remote logins, the I/O must be forwarded through several copiers. To have access to a remote shell, all the copiers connecting it with the actual terminal (or terminal window) must be running. The user

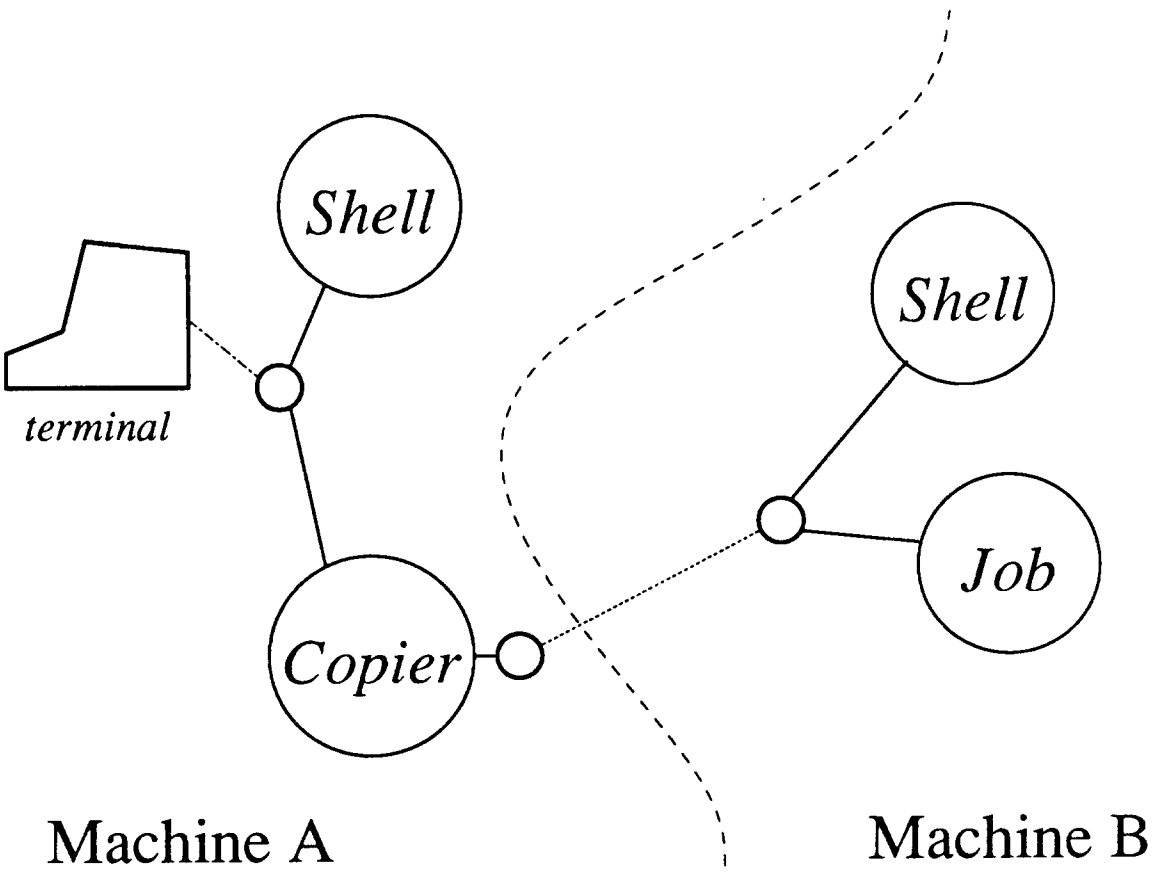


Figure 2.3 A remote shell created through *rexec()* or *rcmd()*

can only interact with one shell of the tree at a time. It is possible to change the active shell to a child of the current active shell (that is, downward in the tree) by resuming a suspended copier. It is possible to change upward, by suspending an active copier. One ordinarily suspends a job by typing a combination of keys that cause a signal to be sent by the kernel device driver to the target job. When one runs a remote login, the key combination is passed by the copiers to the destination machine, where the local PTY/TTY generates the signal. Thus, to suspend the copiers, certain input must be recognized by the copiers as a command to suspend, and not simply passed along to the remote machine.

A program that wishes to create a remote process contacts a different server on the remote machine. The program calls the library routine *rexec()*. Authentication for this routine is discussed in Chapter 4. If the calling process is executing with its identity set to *superuser*, who has highest privileges, a different authentication scheme can be used. In this case, the program calls the routine *rcmd()*, instead of *rexec()*. Both of these routines contact the remote execution server on the remote machine. The user specifies a command string that is to be passed to a shell created on the destination machine. The remote execution server creates a shell on the remote machine (see Figure 2.3). The remote shell sessions created by *rexec()* differ somewhat from those for *rlogin*. There is no pseudodevice on the remote machine (and hence no need for a copier); thus, interactive programs cannot generally be run in this way. Since there is no pseudodevice on the other machine, there is no way of generating signals to suspend the remote process.

If the user wishes to run a single remote command, he may use the command *rsh* rather than *rlogin*. This command takes a single command line as an argument to be submitted to a shell to be created on the remote machine. The *rsh* program uses the routine *rcmd()* to execute the command on the remote machine.

2.5.3. Shells for Multiprocess and Distributed Programming

Lantz [Lan80] describes a command interpreter structure for a distributed heterogeneous system. The concern is to provide the user with access to a number of dissimilar systems, but not to provide distributed jobs. Access to a number of machines is provided by a network of agents. Information about the user's identity, including passwords, is kept in files, to allow transparent agent startup on remote machines.

View3 [Kra83] is a shell that runs on top of an early version of Berkeley UNIX. It allows the creation and control of multiprocess programs on a single site. The shell allows the instantiation of jobs described by templates. It allows the termination of these jobs, or of their component processes. A server layer provides communication abstractions, and synchronization primitives. The service layer allows clients to wait for particular events, including the failure of a program, or the creation of a new process, and allows clients to inquire about the status of jobs, processes, and communication ports. The command interpreter will provide a user with a list of running jobs and the status and configuration of a job's components.

The UNIX shells offer a textual command interface. This limits the expressiveness of the command language, because of the syntactic complexity necessary for describing jobs with arbitrary interconnections. A command interface can also be offered graphically. Graphical command interfaces have been offered in several recent systems. The Upconn tool [BCM88] for example, allows the user to create a description of a distributed program using a graphical editor. Upconn then uses the remote execution facilities of Berkeley UNIX, described above, to instantiate and run this job.

CHAPTER 3

PPM ARCHITECTURE

3.1. The Logical Structure of a User's Work

A user's work is not simply a set of threads scattered across a set of machines, but rather has a natural structure of communication and cooperation. We would like to make achieving this communication and cooperation simpler by providing services and libraries for multiprocess programming that reflect this logical structure and by supporting this structure administratively. We approach the problem outside the operating system, because we would like to use these services to tie together work running under heterogeneous operating systems. It is therefore necessary to define our own logical structure for a login session.

PPM places a fixed hierarchical structure (see Figure 3.1) on a user's work. A user's *threads* belong to *processes*; this much is supported by the underlying operating system. PPM makes processes components of *jobs*, and jobs, in turn, components of *command sessions*. These command sessions compose a login session or, as we shall call it, a *PPM Session*. Threads within a single process, and hence with the same address space, must run on the same machine (at least this is required in current operating systems). Above the level of processes, however, the logical hierarchy is orthogonal to the location of the threads. Thus, jobs, command sessions, and the PPM Session may all span machine boundaries.

Each level of this hierarchy has a distinct purpose and requires somewhat different support:

- A *PPM Session* should encompass all of the work of a single user, spanning the machines on which the user is actively running programs. (We will call the user associated with a particular session the *owner* of the session.) In practice, it may be possible for an owner to have more than one PPM Session, if they run on disjoint sets of machines, but, ideally, a user will own only a single session. In Chapter 1 we mentioned intuitive characteristics of a login session that should be supported for PPM Sessions. It is a scope for bookkeeping, with information held for all the owner's work in the system. It represents a sphere in which the owner has been authenticated, that is a collection of machines on which new processes can be created without requiring the user to supply additional authentication information. It is also a collection of state information used to set up new jobs.
- Some of the state information held for a PPM Session is divided among a number of *command sessions*. It is quite common for a user to interact with a number of command interpreters, particularly when using a bitmapped display device. Each of these command interpreters may have separate defaults for I/O sources and destinations, for working directories, and so forth. Rather than maintaining an absolute distinction between information held for different command sessions, we would like to make distinctions only when necessary, and allow command sessions to share state information when possible. Similarly, while an interactive tool, such as a program monitor, can be made easier to use by displaying for the user information about only a single job or command session, we would like to give such tools access to the bookkeeping information for the entire PPM Session when necessary.

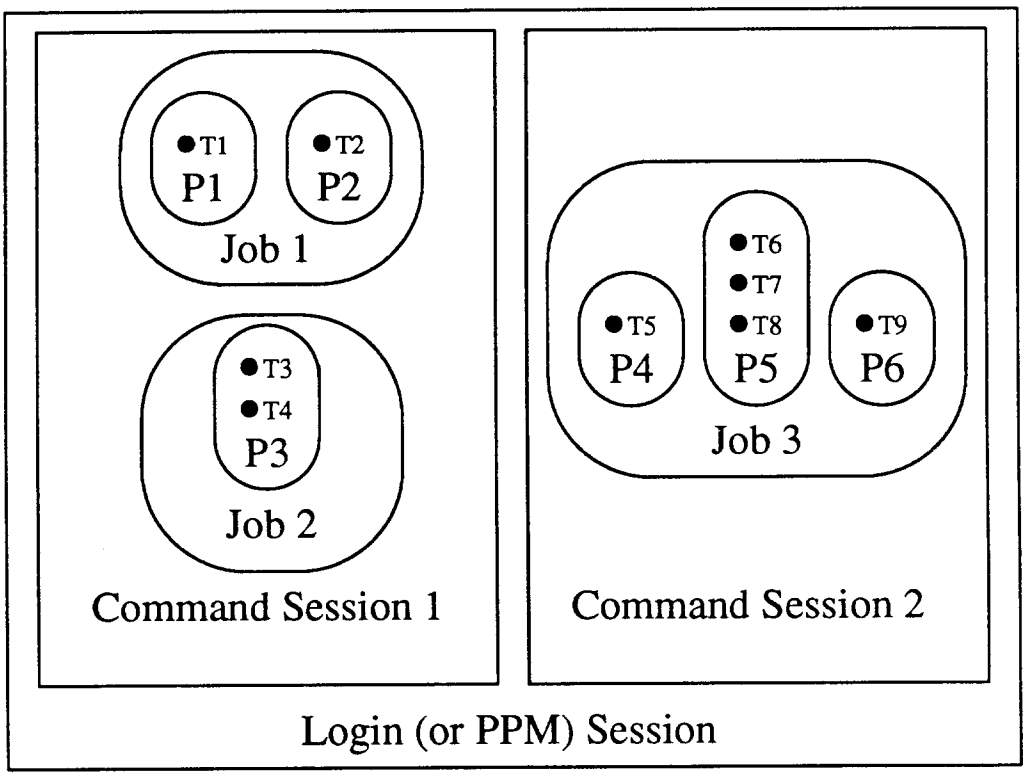


Figure 3.1 Logical divisions of a user's work in a PPM Session

- A *job* in PPM is intended to match a user's intuitive idea of a unit of the work that is being done on his behalf (some of this intuition was described in Chapter 1). Essentially, a job is no more than a grouping of related processes. By grouping processes together, we indicate that they should be subject to the same job control actions, that is it should be possible to suspend or terminate the group as a unit. A job, however, is more than just an administrative unit. The processes of a job are engaged in some common task. This can require them to share parametric information and to communicate, and can also make them interdependent, so that the creation of a new member process or the failure of a member may be of importance to all. Sharing information and handling intrajob dependencies can all be made easier with the assistance of runtime services from PPM.
- A *process* is an assemblage of threads of control within an address space. It is an administrative unit, which a user may wish to suspend, resume, or terminate. It is also a programming unit, with inputs and outputs that must be connected to sources and destinations. The process has to be made to run a particular program on a particular machine, and must be given some set of arguments.
- *Threads* are primarily an administrative unit, as far as PPM is concerned. They may be suspended, resumed, or terminated, and their status made available to their owner. They are not, however, a programming unit, as presented by PPM. Their creation is assumed to be handled by the underlying operating system, and they are assumed to be able to communicate and to share information by virtue of occupying the same address space.

It should be emphasized that these are PPM-provided abstractions. They rely upon the underlying operating system to provide the facilities necessary to actually execute programs. The programmer has access to the underlying system-provided abstractions as well as those of PPM when system-specific functionality is required, but many system dependencies can be hidden by PPM.

PPM provides distinct functionality at each of the five levels of the logical hierarchy. It would be possible to allow particular levels to be divided into arbitrary numbers of sublevels having similar functionality. The command session level could allow multiple levels of subcommand sessions, and the job level could allow multiple levels of subjobs. Adding subcommand sessions would make it easier to allow changes to be made to command session state information and then undone. Adding subcommands would make it easier, in some cases, to guarantee that two separately written programs can be combined in a single job without interfering with one another. It is not clear, however, that these benefits were worth the added complexity in logical organization, and so PPM supports a fixed hierarchical organization.

3.2. Administering a PPM Session

PPM allows a user to administer a login session spanning several machines. The owner of a PPM Session needs several facilities:

- *Job execution* -- PPM lets an owner, through command interpreters, create jobs and instantiate their component processes. Jobs can be distributed; PPM allows the creation of processes on any of the machines in the PPM Session and, when necessary, the expansion of the session to include new machines.
- *Bookkeeping* -- The PPM Session includes all of an owner's processes, wherever they lie. Bookkeeping facilities allow the user to obtain information about his active jobs and their processes (without having to cull this information from information about processes belonging to other owners). We will accept the loss of information about processes that had been running on a machine that fails, but running processes should not be "lost" due to such a failure.
- *Control* -- An owner of a session can suspend, resume, or terminate a job. PPM also allows control of a particular process or thread. This control is hierarchical, so that, if a process within a job is suspended, and then the entire job suspended and then resumed, the process remains suspended.
- *Information sharing* -- PPM provides a pool of name-value bindings, shared among all processes in the PPM Session, called the *PPM Environment*. The environment information to be passed to new jobs and processes indicating service addresses to use, the name of the host machine, default values for parameters, and so forth. It also allows a simple form of interprocess communication through shared bindings. The name space for the environment reflects the logical structure of the PPM Session providing scopes for sharing within the entire session, particular command sessions, particular jobs and particular processes. Scopes provide a means of disambiguating which of two values bound to the same name applies to a particular thread. The PPM Environment Service does not provide a separate scope for bindings applicable to a single thread, because threads within an address space can easily use different names.

3.3. Supporting Multiprocess Computation

A job run by PPM is a set of processes containing some number of threads. PPM provides services that allow clients to create processes within a job, configure their I/O, and start them

running. The PPM Services and associated library routines simplify some aspects of interprocess communication, and the detection and handling of process termination and failure. A process created by PPM is single-threaded. It has a set of byte-stream *inputs* and *outputs*. These can be connected within the job, to form one-way communications channels, or they can be connected to sources of input or destinations for output lying outside the job. A process can also have *ports*, through which it can receive messages without connecting to the message sender.

This is a restricted model of a multiprocess computation, but one that is realizable on a wide range of operating systems. The underlying operating system may allow programs to create additional threads or I/O devices with other forms of semantics. PPM attempts to describe such configurations in its bookkeeping operations, but it does not provide an interface through which to create them.

3.3.1. Job and Process Creation

A process is added to one job or another as it is created. By default, a child process is placed in the same job as its (logical) parent, but, by explicit request, it can be made a member of a new job, or can be added to some other existing job. Command interpreters can take advantage of this to create a new job for every command executed. Jobs in PPM belong to *command sessions*. As with the job membership of a new process, a new job is, by default, a member of the same command session as is the process that created it. It is also possible to explicitly make a new job a member of an existing command session, or a member of a new command session.

3.3.2. Detecting and Handling Termination and Failure

A multiprocess job will often have dependencies among its components. The abnormal termination of one component may require the termination of the job, or may call for some compensatory action. It can be difficult for applications programs to properly handle termination and failure because of the difficulty in detecting these events. PPM offers two forms of assistance: *notification* and *exit actions*.

- *Notification* -- A program can explicitly wait for a thread, process, or job to terminate (or simply change state). Notification will occur even if the machine on which the component is running fails.
- *Exit actions* -- When a process terminates (normally or abnormally) PPM checks for any specified exit actions. These are instructions that can be specified for any process. The actions can call for the termination of other components of the job, including the entire job, or for a notification message to be sent to some specified recipient.

3.3.3. Supporting Communication

Jobs must communicate both internally and with external services. PPM can assist communication in three ways.

- *Communication through the environment* -- The PPM environment provides a shared pool of bindings. Processes can communicate with one another by setting and evaluating bindings. Although this is somewhat slow, it is quite useful for limited communication and for setting up other communication channels.
- *Configuration* -- PPM allows the creation of communications channels between processes as part of job creation. This allows configuration to be programmed as command scripts or in some configuration language, rather than as part of the application itself. The

applications are then freer of dependencies on particular network addressing schemes and connection semantics.

- *Communication libraries* -- Applications should be able to hide dependencies upon particular network semantics. Configuring channels before starting a job can hide one class of dependencies. Message-based communication and the dynamic creation of channels require support as the program is running. Addressing and channel establishment semantics can be hidden at runtime by communication libraries. The PPM communications libraries use the PPM Environment to provide logical names for communications endpoints and to store and distribute their addresses.

PPM does not attempt to support all communication paradigms that are available in a system, although applications are free to use other paradigms without PPM's assistance. Instead, some simple paradigms of proven utility are supported. PPM is attempting not to make easy the programming of all types of interprocess communication, but to make certain generally useful types of IPC easier. In the current version of PPM, the supported paradigms are one-way byte stream communication and client-server message communication. Future versions may support other byte stream multiplexing and demultiplexing and reliable message multicast.

One-way byte streams, such as UNIX pipes, have proven popular in a variety of systems. Through configuration operations, they can be provided in such a way that two processes will be able to communicate, in one direction, from the moment the processes start. Library routines can assist in the establishment of new channels while a program is running.

Client-server message communication is supported through configuration operations to create a port for a server and to make its address available to potential clients. Library routines can be used to allow clients to communicate with servers while hiding details of the underlying addressing scheme and message semantics. Client-server communication can be used to implement other styles of communication, such as remote procedure calls.

3.3.4. Some Things Not Supported

The PPM logical hierarchy is not hierarchical in the sense of LeBlanc and Friedberg [LeF85], that is processes cannot be combined to form subjobs, with inputs and outputs, and these subjobs combined to form larger subjobs, and so forth. That approach provides the ability to hide information and to avoid certain name conflicts. As mentioned above, we preferred to use a simpler hierarchy. We do not provide typed interfaces for I/O objects, as do LeBlanc and Friedberg, preferring to leave this sort of support to applications, or libraries built on top of the PPM facilities.

Jobs are useful to the programmer of multiprocess applications in other ways. A process may wish to communicate with its peers engaged in some common activity. Forming the processes into a job can help solve this problem if there are mechanisms for communicating with all of the job members. One such mechanism is a multicast group [ChM84a]. If process control is based on IPC messages, a multicast mechanism can also be used for the job control functions described above [ChZ85]. The use of multicast mechanisms is compatible with PPM, but is not currently supported.

Some systems allow jobs to designate a particular process to handle a particular type of exception. This may be done explicitly [JCD79] [OSS80], or implicitly, as in TENEX [BBM72], where responsibility for exception handling can be deferred by a child to its parent. This sort of functionality requires kernel support not available in many systems, and is therefore not part of the PPM job abstraction.

It has been argued in the past that jobs are an important unit for scheduling. The Medusa system [OSS80] advanced the notion that a "team" of processes in a multiprocessor should be *coscheduled*, that is guaranteed to be executing concurrently, to prevent a form of thrashing that can develop when a process holding a lock is descheduled. The authors note that, if coscheduling is not provided by the operating system, it cannot be implemented using higher level processes. Thus, the notion of coscheduling is inappropriate for a network, where both system heterogeneity and variable network delays come into play. This idea of coscheduled "teams" is extremely system dependent and is not supported in the present work.

3.4. Supporting a PPM Session

Shells, such as the UNIX shells described in the previous chapter, provide support for computing on a single machine. These shells provide certain services internally, while relying on the underlying kernel (or possibly on local services) for others, as schematically shown by Figure 3.2. Two problems limit the usefulness of these shells in a distributed systems. First, the reliance on the kernel for some of the services limits the shell to operation on only a single machine. Second, the models used for the environment and for jobs may be too inflexible to accommodate distributed computations. The first of these problems can be attacked either by providing distributed operations underneath the kernel interface or by building a set of distributed services, offering a new location transparent interface, between the shell and the kernel. PPM takes the latter approach. The second problem cannot be attacked through location transparency. PPM attacks the problem through the abstractions supported by the PPM Services.

Because the layer of distributed services supports abstractions for jobs and the environment, abstractions previously supported by the shell, we can say that the PPM Services offer *shell services*. The remaining portions of the shell that use these services we will call *shell client programs*. PPM thus presents a division of the shell functionality into facilities provided by shell services and facilities provided by clients of these services (see Figure 3.3). The functionality provided by the PPM Services includes the creation and control of jobs and processes,

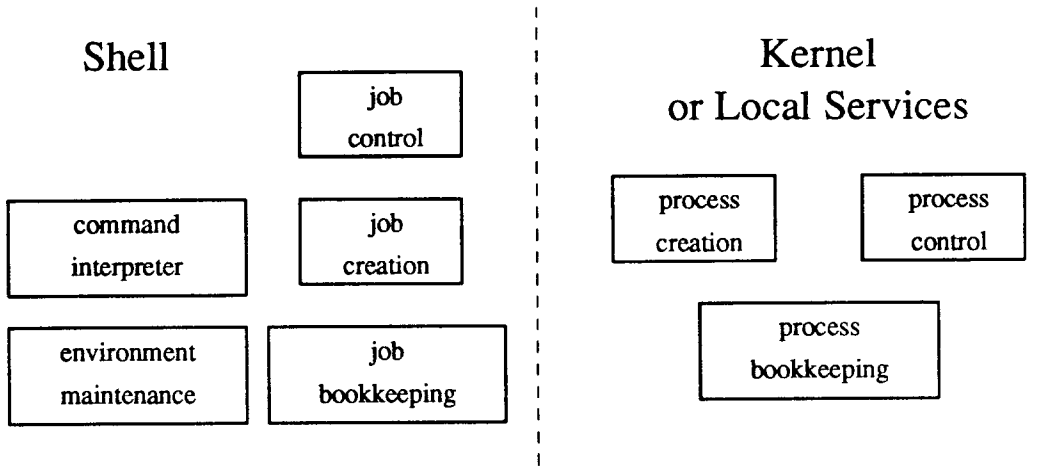


Figure 3.2 The UNIX shells provide certain services internally while relying on the underlying kernel for others.

keeping track of job membership, process relationships and other information, and exporting a shared environment. These services are provided by an infrastructure of cooperating agents that can be extended to cover a network of many sites without losing the essential unity of the interface offered to the clients. The agents use the services of the underlying kernels, but provide distinct abstractions for jobs, processes, and environments. Client programs can act through the services to create, track, and control jobs operating in a network of computers.

Unlike the services of a UNIX shell shown in Figure 3.2, the PPM Services draw no distinction between the command interpreter and any other client. Figure 3.3 shows only a single command interpreter as a client, but the services are available to any user program. This not only makes the services available to distributed applications, but also makes it easy to program new forms of command interfaces, and for these interfaces to cooperate. One might, for example, start a job through a command interpreter, and terminate it from a menu offered by a completely different program.

3.5. The PPM Services

This section briefly describes the PPM Services. Detailed discussion of their interfaces is left to Chapter 4.

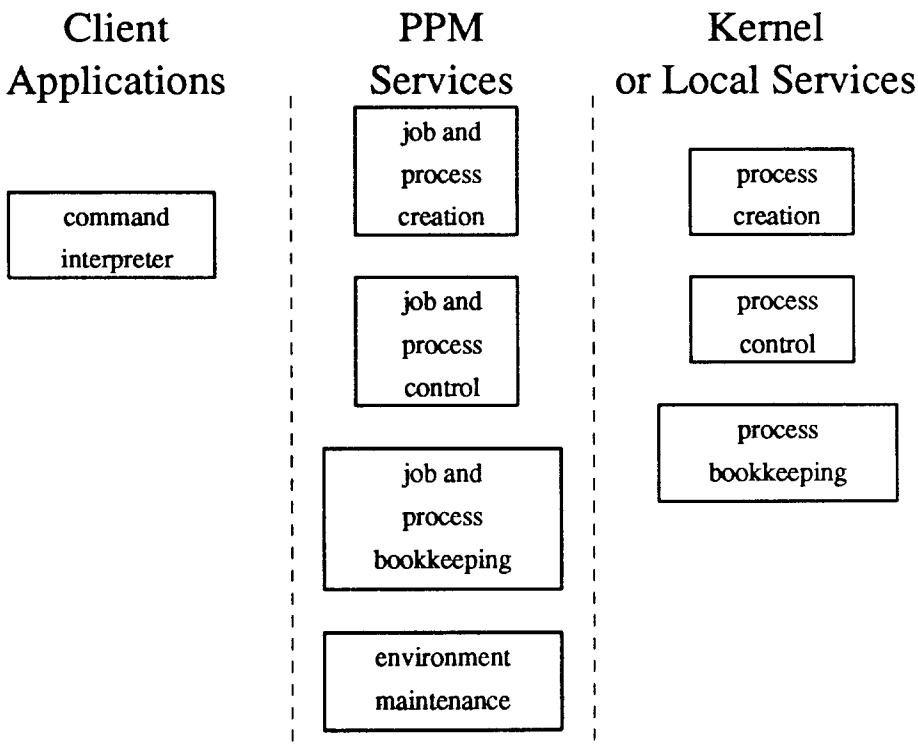


Figure 3.3 PPM divides the shell functionality between applications and distributes services.

3.5.1. The PPM Bookkeeping Service

The PPM Bookkeeping Service supplies information about a user's current jobs. When requested, it notifies clients of the occurrence of events such as job termination, and the creation of new processes. The Bookkeeping Service stores information about jobs, using notification messages received from the kernel and from other portions of the PPM to update its information.

The Bookkeeping Service agents are primarily responsible for keeping information about their local machines up to date. Rather than exchanging information frequently, the agents exchange information as infrequently as possible. When requests for information are made that require information from more than one agent, the request is propagated to the appropriate agents through the local agent, the returned information is collated locally by the requesting process. The collation is done by library routines, so it is hidden behind the programmer's interface to PPM routines.

The bookkeeping information is organized around the concept of jobs. Information is kept about jobs from the time of the job's creation until the job information is explicitly released. The information therefore can include processes that have died. In UNIX, information about dead processes is held in the kernel until collected by the parent; there can be problems in collecting this state, and it can in some cases be mistakenly disposed of. This arrangement is sufficient for running and debugging single process programs, but is inappropriate for multiprocess program development. In PPM, this state is held by the Bookkeeping Service and can be read multiple times. This arrangement makes it simpler for a single process to collect state information for a multiprocess job, particularly when the job is distributed.

3.5.2. The PPM Creation Service

The PPM Creation Service's first duty is the creation of jobs. A job is an abstraction for the instantiation of a command. A job is a set of processes containing threads. The job has a communication structure connecting the threads to sources of input and to sinks for output. The Creation Service instantiates jobs by creating address spaces, threads, inputs and outputs. The service allows the addition of further address spaces, threads, and communication channels to jobs that already exist.

The Creation Service must also perform control operations on threads of control. There are three control operations, setting a thread running, stopping it temporarily, and destroying it. These operations can be performed on a particular thread, on the threads sharing a particular address space, or on an entire job.

3.5.3. The PPM Environment Service

The PPM Environment Service binds values to names allowing the user to later look up the values by name. This name service allows the transfer or sharing of arbitrary information between programs that are able to agree upon names. The set of bindings available to a job provides an *environment*. There are three uses for the environment.

- Because the environment is shared, it can be used for interprocess communication. Operations to lock particular bindings are provided to allow synchronization. Although setting bindings in the environment is not an efficient means of holding conversations, it can be used in setting up more efficient communications channels.
- The environment serves as a state repository, allowing programs to have side effects upon subsequently executing programs.

- Conventions for using this environment allow the command interpreter to provide information to jobs it runs running that can tailor the job's behavior to the user's expectations or desires.

The Environment Service provides a name space with a number of separate *contexts*. These contexts have a rich structure that allows bindings to be set for particular jobs or groups of jobs. This allows different jobs, or threads within different processes of the same job, to see different bindings for particular names. The structure of the name space usually allows PPM Sessions to merge without conflicts in their environments (although this cannot be guaranteed). The organization of the name space is discussed in Chapter 6.

3.5.4. The PPM Session Maintenance Service

The PPM Session Maintenance Service is charged with keeping track of the machines and agents of a particular PPM Session. This service is called by agents offering PPM Services to to extend the session to a new machine, or to drop a machine from the session. It can also be used to notify debuggers or program monitors of changes in the PPM Session. If a session is extended to a machine already part of a different session belonging to the same owner, the service will supervise the merger of the two session's membership lists yielding a single merged session. Clients of the service can request information about the current membership or they can can request notification of all changes in membership. The Session Maintenance Service must monitor extensions of the PPM session, the merger of sessions, or the failure of session members. When membership changes, the detecting site runs a membership agreement protocol that ensures that all the sites that are able to communicate with one another will arrive at consistent membership lists, even when failures occur in the midst of the membership agreement procedure.

3.5.5. The PM Master Service

The PM Master Service acts as a name server for the agents offering the PPM Services. Dropping the initial 'P' reflects the fact that this is a *system* service, rather than a *personal* service. When an agent receives a request that requires an action on a remote machine, the agent will forward the request to an agent on the remote machine offering the proper service. PPM implementations must agree on how such addresses are to be made available. The PM Master Service specifies an interface for obtaining the addresses. This service must be made available by an agent, the PM Master, with a well-known address (or at an address that can be obtained from some other name service).

In our implementation we give the PM Master the additional responsibility of creating agents to offer the other PPM Services. These agents cannot exist everywhere for every potential user all the time. Instead, they are created by the PM Master on demand. When a request arrives for the contact address of agents belonging to some particular user, the PM Master checks whether it has contact addresses for that user. If it does, it returns them. If it does not, it creates a set of LPM agents for the user on the local machine, and records their contact addresses, which it then returns. Thus, each user will always appear to have agents on any machine available to him.

3.6. LPM Agents and PPM Sessions

The PPM Services are offered by processes, which we will call *Local Process Manager agents* (or *LPM agents*). The number of LPM agents required to offer the PPM Services on a machine can vary with the implementation. We assume that each user maintains a separate PPM Session comprising a different set of LPM agents. Client processes make requests of the

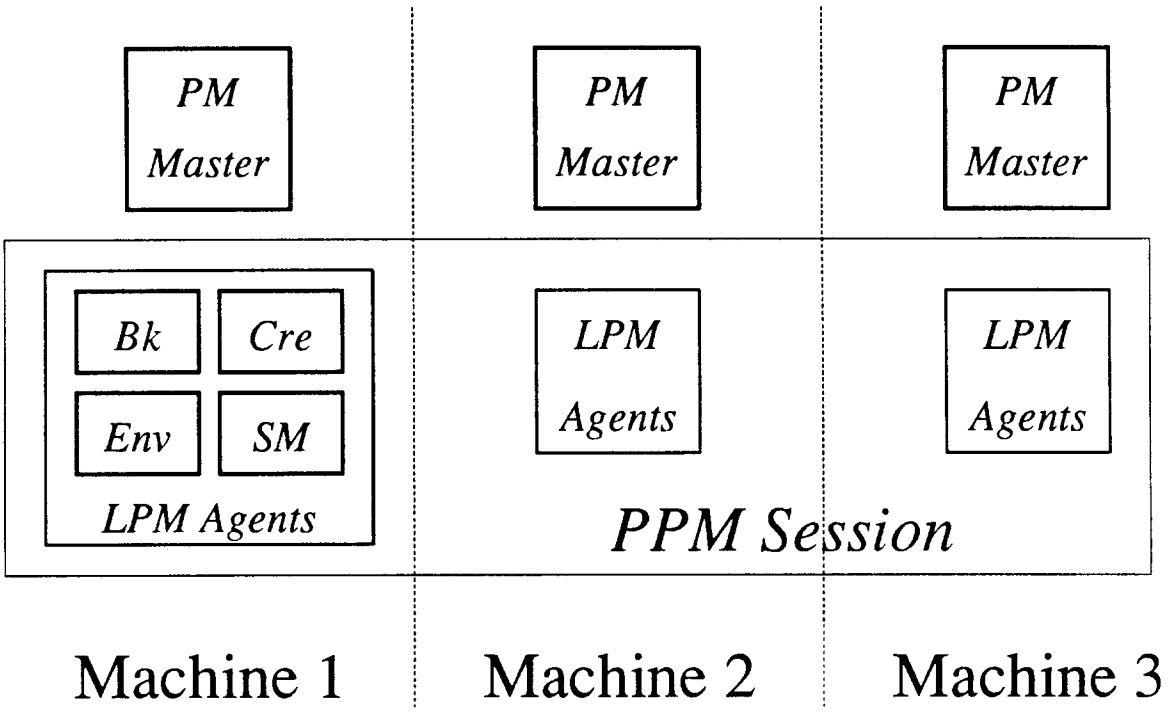


Figure 3.4 A PPM Session spanning three sets of local agents

local LPM agents. Some requests can be handled by the local agents, or by calls from the agents to the local kernel or other local services. Other requests require the local agents to interact with remote agents. For example, data may have to be replicated to ensure resiliency to failure, or a request may need to be made to a remote kernel. If the remote machine is not part of the session, the session must be expanded before the request can be handled.

This, to a large extent, is irrelevant to client programs. *Client programs do not manage the session explicitly*. The session expands to a new machine when a client makes a request that requires an action upon that machine, with new LPM agents created as necessary. The session can withdraw from a machine if, after some time period, the LPM agents on that machine discover that no use is being made of that machine. Again this occurs without prompting by the client program. The LPM agents for that machine can then be terminated. It is even possible for the session to withdraw from the machine where the session started. Although it is thus possible for a PPM Session to flow like an amoeba through a network of machines, most sessions can be expected to have a stable core, expanding to include additional machines when required and then withdrawing.

Ideally, a user will only own a single PPM Session. The user has a single set of LPM agents on any one machine, and those agents participate in only one PPM Session. However, it is possible for a user to own separate PPM Sessions on disjoint sets of machines. This can occur if the user logs into a new machine while an existing session is running. (In fact, every time a session expands to include a new machine, a second session is first created and then merged with the first.) It can also occur if a communications failure partitions an existing PPM Session. In this case, the separate sessions may both supervise continuing work and thus may

both survive. It is also possible that one or both of the sessions will terminate because no work is being done within it. Separate sessions will continue as long as they have work and as long as they use only disjoint sets of machines. Two sessions will merge if they detect one another, for example if a client of one of the sessions makes a request to a machine in the other session.

An owner's LPM agents on a particular machine are started upon demand. Initially, every machine has a *PM Master*, an agent able to create LPM agents for all users and to act as a name server for them. The LPM agents for a particular user have service contact addresses that are stored by the PM Master. A client program seeking an LPM service, or an LPM agent on a different machine seeking to forward a request, can request these contact addresses from the PM Master for a particular machine. Authentication and related topics in the management of PPM Sessions are discussed in Chapter 5.

A PPM Session, then, is instantiated as a set of LPM agents created by PM Masters. Figure 3.4 shows a PPM Session spanning three machines. On each machine there are a set of LPM agents offering PPM services described below. The PPM architecture specifies the services to be offered by the LPM agents, rather than their number or organization. LPM agents offer four personal services, namely, the PPM Site Maintenance Service the PPM Bookkeeping Service, the PPM Creation Service, and the PPM Environment Service. Each has a separately listed contact address, although the address may be shared by more than one service in a particular implementation. In the implementation described in this dissertation, each of the services is implemented as a separate agent.

CHAPTER 4

THE PPM SERVICES AND IPC LIBRARY

4.1. Interfaces to the PPM Services

The previous chapter showed how the PPM Services fit into an architecture for supporting distributed and multiprocess computation. We now describe these services in greater detail, presenting their procedure call interfaces, and some of the design decisions that underlie them. Chapter 8 discusses a prototype implementation of these services. Chapter 7 discusses how these these services can be used by client programs. We also describe the interface of the library that can be used by PPM clients for message-based interprocess communication.

4.2. Identifiers

In later sections of this chapter we shall refer to a number of types of identifiers passed as procedure arguments. PPM requires identifiers for a number of things, including session identifiers, machine identifiers, and so forth. These identifiers are used for most accesses to PPM services. The forms of the identifiers defined by PPM are independent of the form of the identifiers in the underlying system. The identifiers are therefore universal and can be used between clients and servers running on dissimilar operating systems. Machines are assumed to be identified by fixed length unique identifiers such as network addresses. This section describes the form of other identifiers used by PPM.

4.2.1. PPM Session Identifiers

A PPM Session identifier distinguishes between PPM Sessions started by the same user. Each time the membership of a session changes, a new session identifier is created by the PPM Session Maintenance Service as part of a membership agreement protocol. The session identifier has two parts, a session number, retained across membership changes, and a version number. The session numbers include a time stamp. When two sessions merge, which occurs each time a new machine is added to a session, the older session number is retained. (The exact ordering of closely spaced timestamps is not important.) The version number includes an integer and a machine identifier for a member of the session. The version numbers discriminate between the session before and after a membership change. The machine identifier is necessary because of the possibility of partition. If a partition occurs, two sessions will result. The new PPM Session identifiers will both retain the same session number from the now partitioned original session and both will have a version number one nigher then that of the partitioned session. Since the memberships of two distinct sessions cannot overlap, an identifier for a member machine can distinguish between them. (Wrap-around of the version numbers is not considered a problem.) Including the session number in the PPM Session identifier allows the PPM Services to distinguish between the merger of unrelated sessions and the reunification of a partitioned session.

PPM Session identifiers then have the following form:

[SESSION ID] = [SESSION NUMBER][VERSION NUMBER]
[SESSION NUMBER] = [MACHINE ID][START TIME]
[VERSION NUMBER] = [INTEGER][MEMBER MACHINE ID]

4.2.2. Command Session Identifiers

The PPM Environment and Bookkeeping Services hold state information for some number of command sessions. Jobs are started in a command session, inheriting the Environment's bindings for that command session. The command sessions are not localized on any single machine, but are rather divisions of the PPM Session as a whole. They require identifiers that will not be confused with other command sessions of the same PPM Session. This is most easily done by combining an LPM agent identifier with a timestamp. It is actually sufficient simply to use a machine identifier and a time stamp, a since command session identifiers are assigned only by the PPM Bookkeeping Service, so only one LPM agent on any machine creates these identifiers. A command session identifier, therefore, has the form

[COMMAND SESSION ID] = [MACHINE ID][TIME STAMP]

4.2.3. Identifiers for Threads

Threads of control are grouped into processes and processes into jobs. Since jobs have a three-level hierarchical organization, full thread identifiers have three components, a job identifier, a process identifier, and a thread identifier. The job identifier is globally unique, while the process identifier need only be unique within a job and a thread identifier within a process.

Job identifiers are assigned by the PPM Bookkeeping Service. They need to be locally selectable by LPM agents and globally unique. Job identifiers can be created easily by combining a machine identifier with a version number as follows:

[JOB ID] = [MACHINE ID][VERSION NO]

A version number can be produced more quickly than a time stamp. It provides weaker assurance of uniqueness, because LPM agents can shut down and restart, resetting their versions. The LPM agents, however, will not shut down voluntarily while a job with which they are concerned is running. Stronger uniqueness assurance is needed for command session identifiers, since LPM agents on any particular machine can come and go in the course of a command session.

Process and thread identifiers are also assigned by the PPM Bookkeeping Service. They must be locally assigned by an LPM agent and not conflict with identifiers chosen by LPM agents on other machines for the same job. To assure uniqueness between machines, these identifiers can include a machine identifier. The second component can be a version number or some other locally unique identifier; these can include identifiers assigned by the underlying system such as process or thread identifiers. Thus process identifiers have the form

[PROCESS ID] = [MACHINE ID][LOCAL UNIQUE INTEGER]

and thread identifiers have the form

[THREAD ID] = [MACHINE ID][LOCAL UNIQUE INTEGER].

When referring to a particular thread, it is necessary to specify a triplet of job, process, and thread identifiers. A fully specified thread identifier therefore has the form

[JOB ID][PROCESS ID][THREAD ID]

It is possible to refer to all the threads in a particular job using an identifier of the same format by specifying the job identifier together with wildcard identifiers for the process and thread. Similarly all the threads in a process may be referred to by specifying a job and process identifier with a wildcard for the thread. This convention simplifies the procedure call interfaces to the PPM Services.

4.3. The PM Master Service

A client program or an LPM agent can obtain the LPM agents' addresses for any host by calling

```
pmmGetLPM(in: host, out: site, rc).
```

The call returns the addresses for the LPM agents belonging to the client's owner, along with other information, in a "site" structure. If no host is specified, the local host is used as a default. A client has no reason (and is not allowed) to contact LPM agents belonging to other owners. (Authentication is discussed in Chapter 5.)

The PM Master can also be queried for a list of the owners that have active LPM agents on a particular machine by calling

```
pmmGetOwners(in: host, out: ownerList, numOwners, rc).
```

This is the limit of PPM's concern for providing information about other owners. Information about the active users or a particular machine is useful, if, for example, one is searching for a machine with no active users, or one wishes to contact all the users of a particular machine. Other information held by PPM, however, is essentially private.

The PM Master creates the LPM agents, as well as acting as a name server for them. The PM Master must be able to create processes belonging to any owner. The LPM agents are created upon the first request for their addresses. If an owner has no agents on the PM Master's machine, the PM Master creates the agents and returns their contact addresses to the client. Ordinarily, this first call is made either by the login program or by an LPM agent on a remote machine extending a PPM Session. The new agents initially constitute their own separate PPM Session. If the call was made by a remote agent, the new session must be merged with the existing session. It is the responsibility of the LPM agents in the existing session, rather than of the PM Master Service or the new agents, to initiate the merger.

4.4. The PPM Session Management Service

The PPM Session Management Service maintains a list of the peer LPMs participating in a PPM session. Through the PPM Session Management Service, clients can obtain the current session membership list and notification of changes in membership. They can also request that a site be added or dropped from the session. LPM agents for the other PPM Services use the PPM Session Management Service to merge PPM sessions when a new host is to be added to a session. They also rely upon the service for notification when a host withdraws from the session, or when a site fails, and recovery actions must be taken. Other programs may be clients as well, but typically would not request that sites be added or dropped. For example, a command interpreter might keep a list of the user's current sites to choose where to run programs, or a monitor program might display the PPM Session membership graphically. The management of the Session membership, however, is left to the LPM agents.

The interfaces to the services mentioned above are straightforward. A list of the current membership can be obtained through the call

```
smCurrentSites(out: siteList, numSites, rc).
```

The call returns in the first argument a list of structures holding the name of a site and the contact addresses for each PPM Service. The second argument gives the length of the list. The call

```
smSiteListChanges(in: name, out: rc)
```

requests that a client be kept up to date on changes in the session membership. The client names a port on which he wishes to receive the update notification messages.

A site can be added to the session by calling

```
smAddSite(in: site, out: rc)
```

and dropped by calling

```
smDropSite(in: site, out: rc).
```

The call

```
smAuthInfo(in: site, info, out: rc)
```

is used to pass authentication keys or passwords to be used for specific sites to the local agent of the Session Maintenance Service.

4.5. The PPM Bookkeeping Service

The PPM Bookkeeping Service provides client programs with information about existing jobs, and allows clients to register for notification when certain future events, such as a process exiting, occur. The Bookkeeping Service provides more information about the logical relationships between threads, processes, and jobs than is available from the underlying system. Moreover, this information is retained until explicitly released (or a timeout period has passed), making the information particularly useful for *post mortem* debugging.

The Bookkeeping Service is provided by LPM agents charged with keeping track of job resources and events on their particular machines. Requests for information go to the local agent. A client program's request may not be answerable with the information available to the local agent. In this case, the request is forwarded to agents on remote machines. This local responsibility simplifies programming the service and reduces unnecessary traffic between agents. While this choice is, in normal operation, an implementation choice, it has an effect at the architectural level in cases of host failure. When a host fails, information kept exclusively by the local agent is lost. The effects of failure will be discussed below.

Bookkeeping Service agents map local objects into a set of bookkeeping notions that are system independent. The agents, therefore, have two sets of entry routines. One set allows other LPM agents or the underlying kernel to register local occurrences. The other set allows clients to request information or register an interest in future notification.

The specific bookkeeping functions discussed here do not cover every possible requirement. They would, no doubt, evolve significantly if they were used to establish a full set of programming environment tools. Nonetheless, the bookkeeping functions can be modified or extended in a variety of ways without significantly altering the underlying server structure.

Client requests to the PPM Bookkeeping Service are of two types, requests for information and requests for notification of some future event. Requests for information are synchronous, while requests for notification may be either blocking or nonblocking.

A client program can request a list of the jobs running in the system by calling

```
bkJobs(in: site, in/out: time, out: jobList, numJobs, rc).
```

This function returns a list of the jobs that are running at a particular site (or, if no site is specified, for the PPM session as a whole) and at a particular time (or, if none is specified, currently). Clients requiring constant information about the jobs that exist in the system can avoid polling by requesting notification using *bkNews()* as described below.

All the requests are time stamped using either the user supplied time, or the time at the source machine. It is assumed that machine clocks are well enough synchronized to make this meaningful, although there are algorithms in the literature that can guarantee, at greater expense, that a consistent snapshot is obtained [SpK86] [ChL85]. The time stamp used is returned to the user. Since the Bookkeeping Service will throw away information after a few minutes, information for times well in the past is probably incomplete, at best, and may have all been thrown away.

To get more information about a particular job, a client program can call one of the following routines:

```
bkJobStruct(in: job, in/out: time, out: graph, rc)
bkParentStruct(in: job, in/out: time, out: graph, rc)
bkResourceStruct(in: job, in/out: time, out: graph, rc)
```

Each of these routines obtains information about a particular job in the form of a graph. The first routine provides a tree that describes the job in terms of processes and threads. The second routine returns a tree where the nodes are threads and the edges indicate the logical parent-child relations of these threads. The third routine returns a graph where the nodes represent either processes or resources such as communications descriptors. Edges indicate the (sometimes shared) ownership of these resources. Information about descriptors can include established communication channels and open file information, it this is available.

Assembling the information returned by the routines above may require that agents on a number of machines be contacted. This is acceptable if all the machines remain available. In case of host failure, however, information can be lost. In particular, agents on surviving machines may not know exactly what processes have been lost, which may be necessary for an application to handle failure correctly. The loss of information can be avoided by the call

```
bkJobForce(in: job, out: rc),
```

which causes the current structure information for a particular job to be exchanged among the Bookkeeping Service agents at all the sites where job components are running.

Information for a job will be released only after all threads in the job have terminated, and a time out period (held in the environment) has passed or the information explicitly cleared. This may be done by the call

```
bkJobClear(in: job, out: rc).
```

Requests for notification fall into two categories, namely, requests for notifications regarding a particular thread or group of threads and requests for notifications about certain classes of events.

The first category is requested by calling

```
bkWait(in: name, entity, event, out: info, rc).
```

The entity may be a thread, the set of threads within a process, or the threads within a job. The event can be the end of the last component thread, or the next change of state by any thread. This call can be nonblocking, in which case *name* holds a string identifying an IPC address to which to send the notification message. (Strings are used to avoid passing addresses explicitly.

IPC support is discussed in later in this chapter.) If the empty string is used, the call is blocking, and the event change information is passed back in *info*.

The second category is requested by calling

```
bkNews(in: name, event, site, out: rc).
```

Each time an event of some type occurs, a message will be sent to the address indicated by *name*. The events that can be tracked include the stopping and starting of jobs, processes or threads. This can be requested for a particular site or for the session as a whole. In either case, the bookkeeper on the machines affected will send messages to the port associated with the specified name as soon as they learn of the specified event. This same routine can be used to turn off these messages when they are no longer required.

4.5.1. The Effects of Failure

The information available through the PPM Bookkeeping Service is obtained from agents on the machines comprised by the PPM Session. If one or more of the machines were to fail, it is possible that information will be lost. It would be possible for the Bookkeeping Service to replicate information about all processes at all sites, or a large number of sites. This hurts performance by sending many messages that will prove unnecessary in almost all cases. Furthermore, it will not guarantee that information is not lost, since the local agent's information is not updated atomically with the creation of new processes. Rather than trying to perform the update atomically, we choose to accept the loss of some information upon host failure.

Suppose a process on machine A were to start a child on machine B, and this second process were also to start a third on this same machine, all three processes members of the same job. Suppose that machine B were then to fail. The bookkeeping agent on machine A will be notified of a change in the PPM Session membership by the PPM Session Maintenance Service. The bookkeeping agent will have a record of the child created on B, and this child can be declared terminated. A client calling *bkParentStruct()* regarding the job would be returned a graph containing a node for the dead second process, but not for the third.

If machine A had crashed, rather than machine B, the lost process would have been an ancestor of the processes on machine B. It is not hard to store ancestor information, when starting new processes, so the bookkeeper on machine B can mark all the ancestors on machine A dead. Similarly, if the third process had been started on some third machine, C, and B had then crashed, the bookkeeper on C would recognize that B was no longer in the PPM Session, but that A was. As part of reconfiguration, it would inform the bookkeeper on A that the process on C was a descendant of the process on B and, through it, of the process on A. Thus, processes that continue to exist within the session will not be "lost" as a result of a failure.

4.5.2. Exit Actions

When the last thread of a process exits, or when the process is presumed dead as a result of a failure, the LPM agents providing the Bookkeeping Service decide upon the process's exit status. This may be one of a set of constants used by PPM, or may be set by the program through a call to the PPM Creation Service. The agents then check to see whether the process has any exit actions specified. These are stored in the process's environment held by the PPM Environment Service.

Exit actions of the form

```
if (cond) inform name
```

cause an IPC message to be sent to the address bound to *name* in the environment when the condition is true. Currently, the only conditions anticipated are tests on a process's exit status.

Exit actions of the form

```
if (cond) kill entity
```

will result in the specified entity being killed. The entities can include the job, all descendant processes, or a particular process, either stated explicitly, or bound to a name used in place of an entity designator.

4.6. The PPM Creation Service

The PPM Creation Service provides routines that create new jobs, new processes and new threads. The service also provides routines to configure the I/O of processes, and routines to suspend, resume and halt threads. These services are intended to provide a useful set of functions that are implementable on a wide variety of systems.

PPM follows the basic pipeline segment structure used by UNIX (see Chapter 2). A process generally has a byte stream standard input, standard output, and standard error. A process creation request to the PPM Creation Service implicitly carries the request for the creation of these inputs and outputs. Therefore, separate configuration requests are not necessary to create them. The source of the input and destinations of the outputs are specified through the environment. The bindings specifying the standard I/O configuration can be inherited, or can be specified for a particular job or process. The Creation Service evaluates the binding when instantiating the process. The binding values can specify that a descriptor is to be closed, or can encode the appropriate configuration, including connections to terminals, to files, and to pipes.

In addition to evaluating bindings for the standard I/O configuration, the Creation Service evaluates other bindings as part of process creation. Specifically, it evaluates the binding for the target machine, that is the machine on which the process is to be instantiated. In the UNIX implementation, the Creation Service also evaluates the current working directory. In other systems, it is likely that there will be other bindings which must be evaluated at the time the process is created.

A PPM process can be configured to have additional inputs, outputs, and message ports through explicit configuration commands. These additional descriptors can be given names. These names are placed in the process's environment, bound to system-specific information about the descriptor. In UNIX, this information includes the descriptor number and its type (input, output, both) and semantics (byte stream, messages). This scheme allows command interpreters using the PPM Services to provide applications with information about their configuration.

The simplest command to the Creation Service is to run a command. This is done through the call

```
creRun(in: argV, envTable, out: job, rc)
```

This routine runs, as a distinct job, a process executing the file specified in the argument vector passed as the first argument. The file name is evaluated by the PPM Creation Service for the target site. An identifier for the job is returned in the third argument. The second argument is a set of changes that are to be made to the environment that is inherited by the job. Some of the environment variables affect where the job is run and how its input and output is arranged. In some cases these are set explicitly by these environment changes. Otherwise, they are inherited from the calling process (which we call the logical parent).

The *creRun()* function is provided as a shortcut for a frequently occurring special case, that of creating an independent job. It can also be used to create an independently running

process with a single thread. This is done through a specification in the environment table that the process created should be made part of an existing job.

The function just described creates a running process. There is also a routine to create a process in the unrun state. A process in the unrun state can have its I/O configuration changed. In particular inputs, outputs and ports can be added to an unrun process, and inputs and outputs connected between such processes. These functions are performed by the following routines:

```
creProcess(in: argV, envTable, out: id, rc)
crePort(in: name, id, out: rc)
creInput(in: name, id, out: rc)
creOutput(in: name, id, out: rc)
creConnect(in: id, name, id, name, out: rc)
```

The hierarchy of threads, jobs, and processes serves as framework for controlling computations. For PPM, a thread is either *not yet run*, *running*, *suspended*, or *terminated*. Processes and Jobs are also either *not yet run*, *running*, *suspended*, or *terminated*. These are *not yet run* only if every component thread is *not yet run*, and *terminated* if all component threads are *terminated*. A terminated process has an *exit status*, which may be normal, or which may indicate some error condition or failure. A thread will actually run only if it is the *running* state, its process is in the *running*, state and its job is in the *running* state.

Once a thread has been created, its state can be changed through the call

```
creStateChange(in: newState, id, out: rc)
```

The id specified can be for a single thread, or for a process or an entire job. The effects of a control operation on a process or job on the state of components is roughly herarchical. Transitions from *not yet run* and to *terminated* are irreversible. The effect of these transitions is to allow the threads of a process or job to be started, suspended, or terminated as a whole. Resuming a process or job, however, will not automatically mean that the threads of a job will be set running, since this will not change a thread from the suspended state to the running state. This allows, for example, a particular thread in a job to be suspended and left suspended as the job is stopped and started. The state transitions are shown in Figure 4.1.

The exit status of a process can be set by calling

```
creSetExitStatus(in: process, status, out: rc).
```

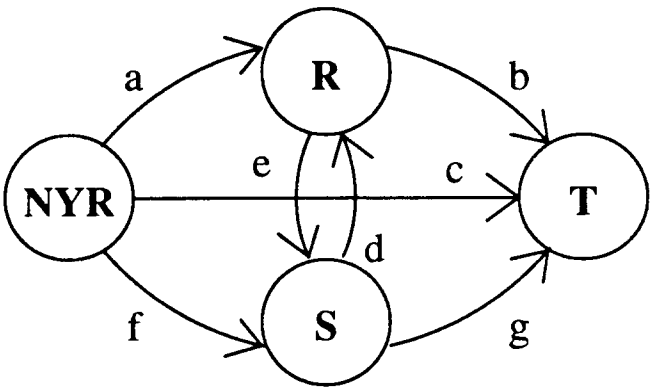
If no process is specified, the calling process is assumed. Ordinarily, this is called to indicate some abnormal event which is causing the process to be terminated. A process's exit status can also be set by the PPM Bookkeeping Service when a failure is detected. A process's exit status may be checked by clients through the Bookkeeping Service. It is also used in deciding if exit actions should be executed.

It is possible to create a job containing no processes by calling the routine

```
creJob(out: id, rc).
```

New processes can be added to the job later by placing a binding for "JOB" in the environment table passed to the Creation Service in *creProcess()*. A new command session can be created by calling the routine

```
creCommSession(out: CSName, rc).
```



Transition	Thread	Process	Job
a	RT, RP, RJ	RP, RJ	RJ
b	TT, TP, TJ	TP, TJ	TJ
c	TT, TP, TJ	TP, TJ	TJ
d	RT	RP	RJ
e	ST	SP	SJ
f	ST, SP, SJ	SP, SJ	SJ
g	TT, TP, TJ	TP, TJ	TJ

Figure 4.1 Transition diagram for threads, processes, and jobs. The states are *not yet run* (NYR), *running* (R), *suspended* (S), or *terminated* (T). The operation to change a thread to the running state is encoded as **RT**, to terminate a job as **TJ**, to suspend a process as **SP**, and so forth.

This returns a string that is the name of the command session. New jobs can be placed in the command session by a binding in the environment table passed to *creRun()* or *creProcess()*.

4.7. The PPM Environment Name Service

The PPM Environment Name Service provides a means of binding names to values, and later evaluating those bindings. The structure of the environment name space and other properties of the environment are discussed in Chapter 6. The present discussion describes the procedure call interface to the Environment Name Service and the basic plan of implementation.

The environment name service provides two basic calls. A name is bound to a value by the call

```
envSet(in: name, value, len, out: rc).
```

A binding can be evaluated by passing a name into the routine

```
envGet(in: name, value, maxlen, out: len, rc).
```

This routine returns the value, if any, bound to the name and its length. The routine *envSet()* blocks until the update is complete (or until a failure is detected). The routine *envGet()* may return either the old or the new value during the time that the *envSet()* operation is blocked.

The value bound to a name may be an array of bytes, or it may be an indirection. An indirection is another name, preceded by the '@' symbol. When looking up a binding with the

calls above, the service will follow indirections, evaluating the name bound as the value of the original name. This process will continue until either a real value is found or a cycle is detected and a failure code is returned. When setting a binding for a name whose value is an indirection, indirections will be found until a real value is found, and that value will be replaced. Again, the detection of a cycle will cause failure.

To set a binding without following indirections, one calls

```
envSetAbs(in: name, value, len, out: rc).
```

To evaluate a binding without following indirections, thus exposing the presence of the indirection, one calls

```
envGetAbs(in: name, value, maxlen, out: len, rc).
```

Binding values are uninterpreted byte arrays. Because a binding's value may be evaluated on a machine architecturally different from that on which it was set, an appropriate data representation should be chosen. In the prototype, ASCII strings have been used for all values, eliminating concerns such as byte order in integers and other data types. Other standard representations of data could be used, such as CCITT X.409, Xerox Courier, or SUN XDR [DeS86]. There is an advantage in using a self-describing format such as CCITT X.409, rather than an implicit format such as SUN XDR. The former encodes the data types in the value, while the latter requires the reader to know the datatypes for the values read. Implicit formats are quite appropriate for remote procedure calls, where the sender and recipient meet an explicit interface specification. The PPM Environment, however, may be used to pass data for which there exists no explicit specification. Under these circumstances debugging is much easier if the data format is self-describing.

Name lookups can be slow if each lookup requires a remote procedure call. Looking up a name frequently can therefore significantly slow down an application. Often these frequent checks are unnecessary, because the binding has not changed. Rather than polling the bindings value, a client can request that it be explicitly informed of a change in a binding's value by calling

```
envCache(in: name, port out: rc).
```

The Environment Service will then send messages to the specified port each time a new value is bound to the name.

To provide synchronization and the ability to update a set of bindings atomically, a binding may be locked. A client requests that a binding be locked by calling

```
envLock(in: name, out: rc)
```

and unlocks the binding either by writing a new value, or by calling

```
envUnlock(in: name, out: rc).
```

While a binding is locked, all attempts to lock or update the binding by other processes will be disallowed. The binding can, however, still be evaluated. This mechanism allows sets of bindings to be updated consistently.

4.8. The IPC Library

In Chapter 3, we pointed out that jobs communicating through messages could be made more portable by making use of interprocess communications (IPC) routines that hid details of addressing and message transmission. In this section we describe a set of routines that do this. They provide an interface that is useful for user programs, and, in fact, they have been used to implement communications between stub routines for PPM clients and the agents offering PPM

Services.

The IPC library routines provide an intermediate level between system IPC primitives and IPC primitives that have been integrated into languages. The interface is portable and supports both request-reply semantics and reliable message passing. The interface does not support typed interfaces or parameter marshaling and unmarshaling. To implement remote procedure calls, these functions must be built into a stub level lying between the client program and the IPC library on the caller's side and between the IPC library and the remote procedure on the called procedure's side. This positioning is illustrated in Figure 4.2.

The basic IPC action is the sending of a message from one process to another. A message consists of a fixed size header and a variable block of data bytes (see Figure 4.3). Each message includes a request code and the identity of the sending process. The recipient of a message can reply to it substituting a new data block. The recipient can include a return code in the reply to indicate whether a request has been successfully completed. The header also has a fixed section for data when the amount of data to be transferred is small (in the present realization this field is four bytes long). The header also has fields that are reserved for use by the IPC library.

A process wishing to send a message has two calls available, *ipcCall()* and *ipcSend()* (see Figure 4.4). The first call blocks the sender until a response arrives or a failure occurs, the second does not allow a reply message, and returns as soon as the target machine acknowledges the receipt of message. The recipient of a message has three options. It may reply to the message if a response is called for, it may dispose of the message if no response is allowed, or it may forward the message to a new server. The first two options are combined in the call *ipcReply()*, while the third is provided by *ipcForward()*. The message passed to these routines must have the same fixed header as the original message; its fields may be changed, however, if they are not reserved for IPC library use, and the variable sized data block may be altered or

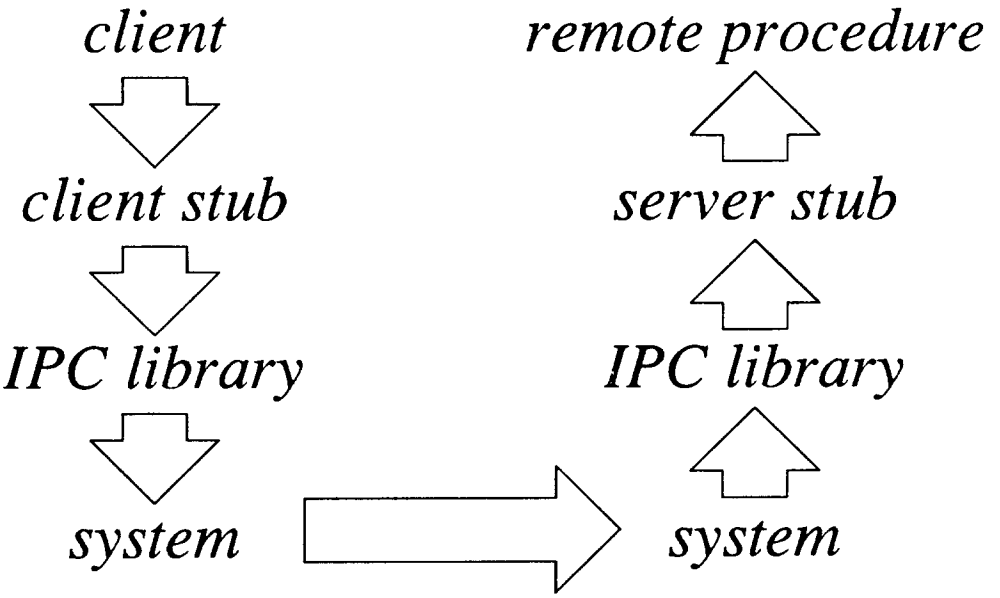


Figure 4.2 Position of IPC library in a remote procedure call

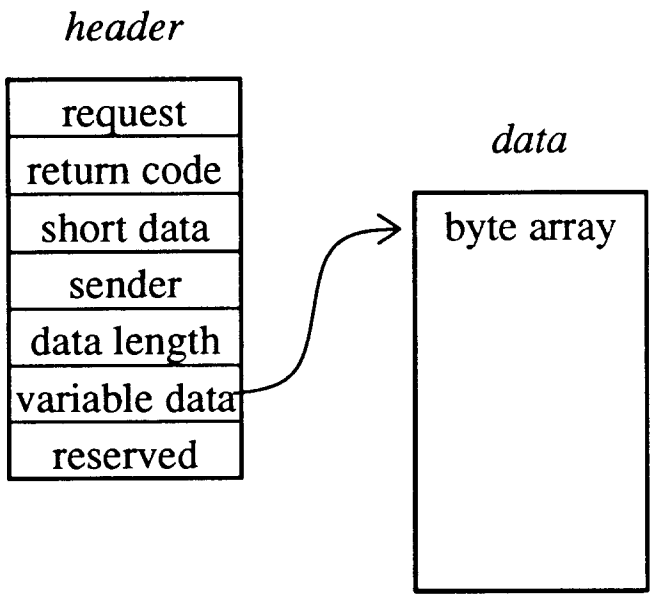


Figure 4.3 IPC library message structure

eliminated. Calling *ipcReply()* will send a reply message to the originator only if the original sender is blocked through *ipcCall()*. If the original sender called the routine *ipcSend()*, and is therefore not blocked, no message will be sent. Although it may not seem strictly necessary to call *ipcReply()* for messages allowing no reply, it is good practice to make the call for all messages, since it may be necessary to clean up the state information associated with the message managed by the IPC library.

It is not difficult to build remote procedure call stubs on top of reliable request-response IPC. It is necessary to build stubs that will perform parameter marshaling and unmarshaling. It may also be necessary to convert the representation of data either to a standard such as CCITT X.409, Xerox Courier, or SUN XDR [DeS86], or to the representation used by the recipient. The stubs used in the PPM implementation were created by hand. It was therefore possible to place additional functionality within the stub routines. In particular, bad return codes could sometimes be handled, and requests could be made to more than one server. Stub compilers could have been used to automate the generation of code to perform parameter marshaling and data representation conversion, had they been available.

The calls *ipcMultiCall()* and *ipcMultiSend()* allow a message to be sent to several recipients. The former blocks the sender until all the recipients have replied. The latter returns without waiting for reply messages. Replicated data is updated within PPM through a two-phase commit protocol. The usual, successful, case is implemented through one call to *ipcMultiCall()* and one call to *ipcMultiSend()*. The former distributes the data and returns acknowledgements. The latter sends a commit.

Messages are addressed to *ports* named by character strings. A port is an abstraction that allows the IPC library to hide the details of addressing from the client program. A process that is to receive messages creates a port with a call to *ipcCreatePort()*. The caller has the option of choosing a port name which is unique among all the threads in the address space, or allowing

ipcInit()

ipcCall(*in:* **portName, msg, replyPortList,** *out:* **rc)**
ipcSend(*in:* **portName, msg,** *out:* **rc)**

ipcMultiCall(*in:* **portList, msg,** *out:* **rc)**
ipcMultiSend(*in:* **portList, msg,** *out:* **rc)**

ipcSelect(*in:* **portList, protection,** *out:* **msg, portName, rc)**

ipcReply(*in:* **msg,** *out:* **rc)**
ipcForward(*in:* **portName, msg,** *out:* **rc)**

ipcMachine(*in:* **msg,** *out:* **machine, rc)**
ipcCaller(*in:* **msg,** *out:* **caller, rc)**

ipcCreatePort(*in-out:* **portName,** *out:* **rc)**
ipcDestroyPort(*in:* **portName,** *out:* **rc)**
ipcMakePortKnown(*in:* **portName, envName,** *out:* **rc)**

ipcFindPort(*in:* **portName, envName,** *out:* **rc)**

Figure 4.4 IPC library interface

the IPC library to choose a name that is guaranteed to be globally unique.

Once a port is created, it must be made known to potential message senders. Within the IPC library a port is some addressable entity such as a Berkeley UNIX socket. The address of this entity must be made available to the IPC library routines in the sending process. There are two ways in which this address may be made known: it may be placed in the PPM-provided environment, or it may be forwarded directly through *ipcCall()*.

The address is placed in the environment through the call *ipcMakePortKnown()*. The address will be bound to an environment name chosen by the caller. By specifying different environment names, this call can be used to publicize an address within various scopes, such as within a job, or for all the user's jobs. This scheme allows servers to place contact addresses for their services in the environments of their clients. A potential client looks up a port in the environment by calling *ipcFindPort()*, specifying an environment name. A later chapter describes the structure of the name space, but it should be pointed out here that the environment names used for lookup need not specify the scope of the name. The client, therefore, does not need to know whether the server has advertised the service universally, or for a single job, as the same lookup name will be used.

The alternative way to inform a potential sender of a port is to place it in the reply port list of a call to *ipcCall()*. A specification of the port address is then piggybacked on the message sent. This piggybacking avoids the relatively high overhead of environment insertions and look-ups. The IPC library at the other site will see this extra information when the message is read. The port name must not conflict with any port names held by the recipient, otherwise an error is returned. The message recipient will not be explicitly informed of the new port, but would be able to use it. Reply ports can be used to allow an asynchronous reply to a call. To do this, a

port address is piggybacked on a message that includes the port's name. The recipient immediately replies to the call, and later uses *ipcSend()* to send a message to the original caller through the reply port. Since servers may receive the names of many reply ports, these names must be chosen to insure uniqueness, or their choice must be left to the IPC library.

The creator of a port can renounce any future attempts to receive messages on it by calling *ipcDestroyPort()*. One of the potential senders, having either received the port or looked it up, can renounce it through the same call.

Messages are received by calling *ipcSelect()*. This call takes as an argument a list of ports created by the process. It returns a message that has arrived at one of the ports together with that port's name. The implementation of this routine should provide some sort of fairness, so that messages at one port will not permanently mask those arriving at another.

A process calling *ipcSelect()* can choose between two levels of protection. The *open* level allows messages from any client to be received. The *private* level rejects any messages sent by a client that is not part of the owner's PPM Session. It is important to note that this means that the sender and recipient have the same *owner*, but not necessarily the same *access control identity*. A process might, for example, have a most-privileged-user identity that can be used by several system administrators, but the process has only one actual owner. Chapter 5 discusses the difference between an *owner* identity and an *access control* identity, and discusses how authentication is done in PPM.

Other forms of access control can be built on top of the IPC library interface. The id of the thread of control that sent the message is in the message header. The recipient can find the machine where a message originated by calling *ipcMachine()*. It can find the access control identity of the message's sender by calling *ipcCaller()*. This is similar to the interface reported by Birrell [Bir85] which allows servers to obtain the identity of the caller from the RPC runtime support.

CHAPTER 5

OWNERSHIP AND AUTHENTICATION

5.1. Login Sessions

A PPM Session is created when the user first logs into a machine and continues until all the jobs in the session have terminated, and a timeout period for submitting new work has passed. In the meantime, the session may expand to include new machines or may contract as failures occur or machines go unutilized.

The login sessions provided by conventional shells, such as the UNIX shells, impose limits on the work done under their aegis. The work can be limited, first, to a particular *machine* and, second, to a particular *identity*. It may be desirable, in a distributed system, to work both on a variety of machines and under a variety of identities. Multiple identities may be useful, or even necessary, in accessing different sets of resources or in exerting different levels of privilege. In a conventional login session model a user can do work under several identities, or on several machines, but only by creating additional login sessions with very little provision made for cooperation between the sessions.

A PPM Session differs from this isolating login session model in several ways:

- The PPM Services support abstractions that are not restricted by machine boundaries.
- The PPM Services allow the owner to run programs within the PPM Session under more than one identity. Allowing the use of multiple identities nondisruptively in a login session makes them a more attractive option in designing systems.
- The remote procedure call interfaces of the services allow a variety of clients to make use of functions that would otherwise be restricted to the shell.

These differences allow command interpreters and other clients to access resources and manage jobs throughout the network, rather than requiring their replication at each site. They also encourage the development of new client applications, rather than the addition of functionality to an existing shell program.

In this chapter we shall discuss the management of PPM Sessions. We contrast the login session model provided by the Shell Services with the model provided by the C shell. In particular, we discuss identities, authentication, and the ways in which multiple identities can be accommodated in a single login session.

5.2. Session Models

A user typically logs into a system by sitting down at a terminal or workstation and typing a password. This act results in the creation of a process able to act on the user's behalf. The process is able to act for the user because it is tagged with an identifier representing the user. We will refer to this as an *identity tag*. We will refer to a set of access rights accorded to some user by resource managers, such as the file system, as a *role*. Although identities can be more complex, for now the reader can think of an identity tag as a single role. When the process makes some request subject to access control, the identity tag attached to a process must be checked. In some cases, the identity tag may be forwarded to the service performing the authorization check. In a networked system, unfortunately, tags are amenable to forgery, by means of

which processes can masquerade under false identities. Authentication, through encryption and other means, is discussed later in this chapter.

5.2.1. The Login/Rlogin Model

In UNIX, when a user enters his password, a shell is created providing him with an interactive command interpreter (see Chapter 2). The UNIX shells run as single processes tagged with the user's identity on particular machines. The processes they create run on the same machine and have the same identity. Some of the standard utilities that can be run from these shells access network services, but the shells themselves do not. The shells terminate when the user logs out.

The C shell (with the assistance of some other UNIX utilities) allows the user to log into several machines simultaneously. Actually, several distinct shells are created, but the inputs and outputs of the shells are subject to redirection so that all of the shells can use the same terminal. This will be termed the *login/rlogin model.* The login/rlogin model allows a user to establish shells with distinct identities. This is illustrated in Figure 5.1. Here a user named Meg has logged on initially to machine M1 using the identity **Meg@M1**. She has started additional shells on the same machine for two additional identities, that of another user, **Bob@M1**, and an administrative identity (or, in UNIX terminology, a *superuser* identity) **SU@M1**. Meg has also logged into the machine M2 with the identity **Meg@M2**, and, through this shell, has logged into a third machine M3 with the identity **Meg@M3**.

Identities such as **Meg@M1** are machine specific. This corresponds to the situation in UNIX, where a user has separate accounts on each machine, and may have separate identifiers

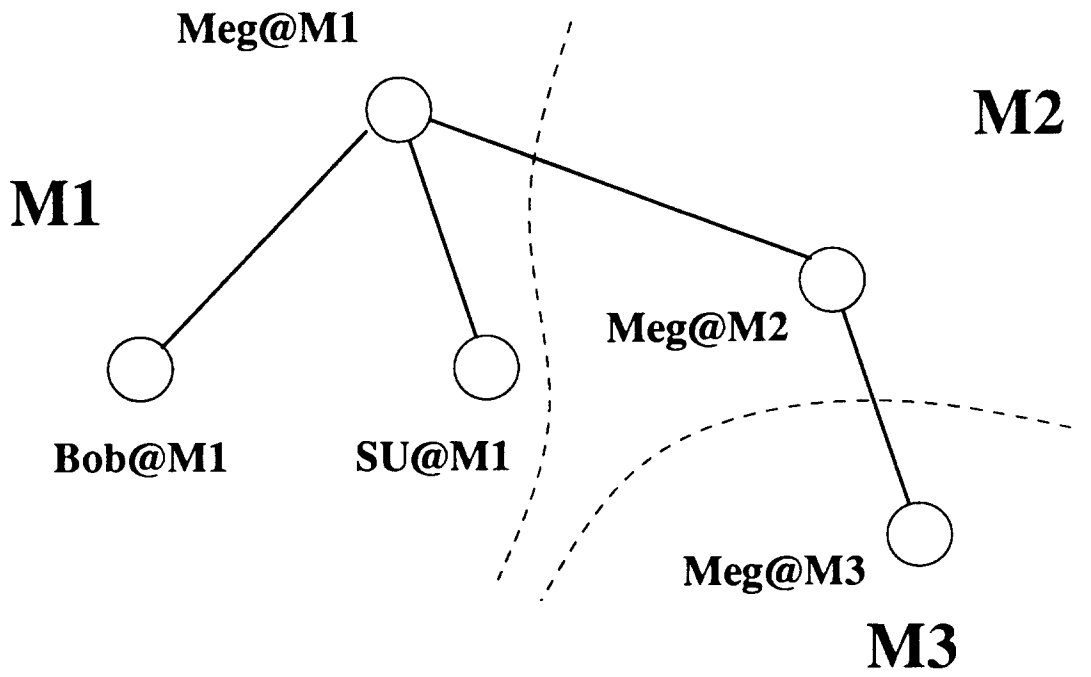


Figure 5.1 Login/rlogin tree

at each, for example Meg might be **Meg@M1** on machine M1 and **Margaret@M2** on machine M2. Because a user may appear under different aliases when making remote accesses from different machines, such systems make it difficult for servers to express access control restrictions and permissions. For this reason, many network systems use machine independent identifiers. These will be discussed later.

Each of the nodes in the tree shown in Figure 5.1 is a separate instance of the C shell. The inputs and outputs of the shells in the tree can be redirected back to the root by a number of programs copying data across stream connections. These *copiers* are represented by the links connecting the shells in the figure. Each C shell treats the source of its input and destination of its output as a terminal. The ability to read from and write to the terminal is conferred on one "foreground" process. In the situation illustrated, only the shell at the root of the tree, labeled **Meg@M1**, has access to the actual terminal. However, by placing the correct set of copiers in the foreground, access to this terminal can be granted to any shell in the tree. Since the input to a node must go through all the ancestors of the node, instructions that return terminal access to shells higher in the tree, can be inserted into the input stream.

One should not make too much of the notion of multimachine login sessions using the login/rlogin model. The cooperation of the separate shells in such a session is, in fact, minimal. Since the shells are separate and commands can be directed only to one shell at a time, login sessions usually have a much simpler structure than that shown in Figure 5.1. In practice, the tree is usually of depth two only, and users rarely do anything more complicated than switching back and forth between the root and its children. Such a session does not provide a good basis for distributed programming.

5.2.2. The PPM Session Model

LPM agents for a user can be created by any authenticated request, whether local or remote in origin. The local requester is generally the login program to which the user types his password. The login program's request to the PM Master Service, authenticated through the password, causes agents for the user to be created. The login program then requests from these agents, through the PPM Creation Service, the creation of a command interpreter that will accept commands from the same source as the login program. The LPM agents are thus distinct from the command interpreter. They are not tied to the existence of this command interpreter. They can outlive it, and even connect to a different command interpreter. They cease to exist when they determine that there is no more work for them to do.

Unlike the login/rlogin model used with the C shell, the agents in a PPM login session cooperate in providing the user with shell services on a group of machines. These agents offer services to a single user, whom we will refer to as the *owner*. Agents acting on behalf of the owner's identities on various machines must merge in a single session. The owner may wish, however, to use resource access rights different from those associated with his identity. That is, he may wish to take on a different identity. For example, Meg may wish to act with the identity **Bob** or **SU**. At the same time Bob may wish to act as **Meg**. This possibility does not imply that Meg's PPM Session should merge with Bob's. Nor is it desirable that Meg's work should be divided between two noncooperative sessions because some work is performed under one identity and some under another. The PPM mechanisms handle sessions where the owner acts under multiple identities by making a distinction between the identity of the session's owner, and the identity tags attached to processes created by the session. A session can include agents with various identity tags, enabling the creation of new processes with a variety of identities. Multiple identities can thus be accommodated without dividing the owner's work between separate PPM Sessions and without placing different owners' work in the same session.

Under PPM, each command session can set its identity independently. Components of jobs within a command session are tagged with this identity. This logical organization is shown in Figure 5.2. The PPM session shown is owned by Meg. Actually, "Meg" may be a shorthand for a set of identities, each specific to a particular machine. For example, it might include **Meg@M1** and **Meg@M2**. It is assumed that a mapping can be made from the owner's identity to the correct machine-specific identity at each site. Meg has created three different command sessions. One has the identity **Meg**, another **Bob**, and the third the superuser identity **SU**. These are also shorthand for a set of identities for different machines. This set is mapped to a specific identity by the PPM Creation Service at the time a process is actually created. To implement this logical organization, the PPM Session must contain LPM agents tagged with different identities (see Figure 5.3). The PPM Creation Service, in particular, uses both master and slave LPM agents. Slave LPM agents must be created through an authenticated request to the PM Master Service. Therefore, Meg must supply the the correct passwords or other authentication information to the master LPM agent. Slave agents are created on the various machines as necessary. This is discussed in Section 6.4.3.

5.2.3. Authentication of PPM Session Extensions

A PPM Session for an owner is started on each machine where authenticated requests arrive to the PM Master Service. There are two types of requesters. A local requester, such as the system's login program, makes use of this new PPM Session. A remote requester, on the other hand, ordinarily is part of an existing PPM Session, which will merge with the new session in order to make use of resources on a new host.

Extension of a PPM Session is ordinarily the result of a request to the PPM Creation Service to create a process on a host not in the PPM Session. The Creation Service then requests

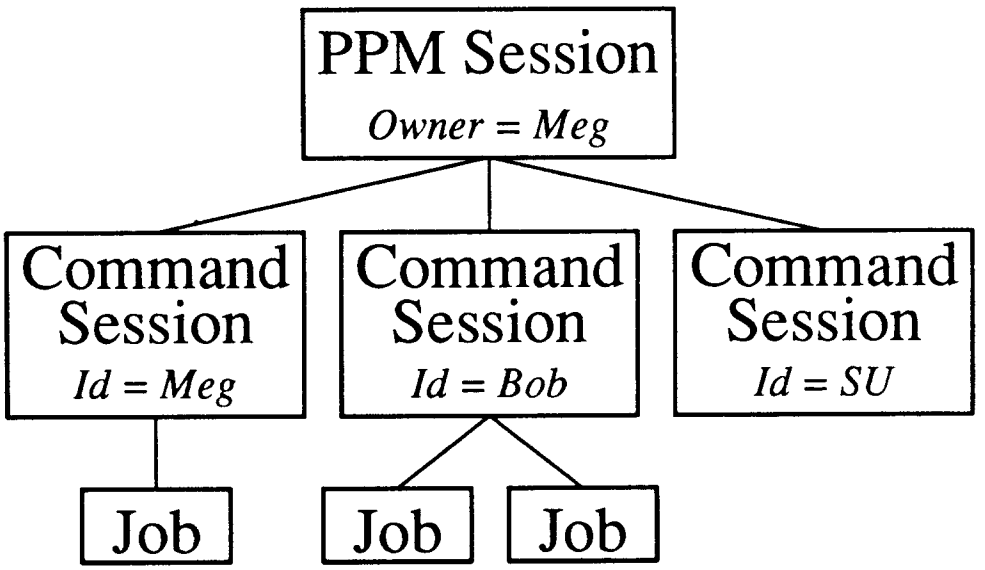


Figure 5.2 A PPM Session for a particular user can have command sessions with different identities.

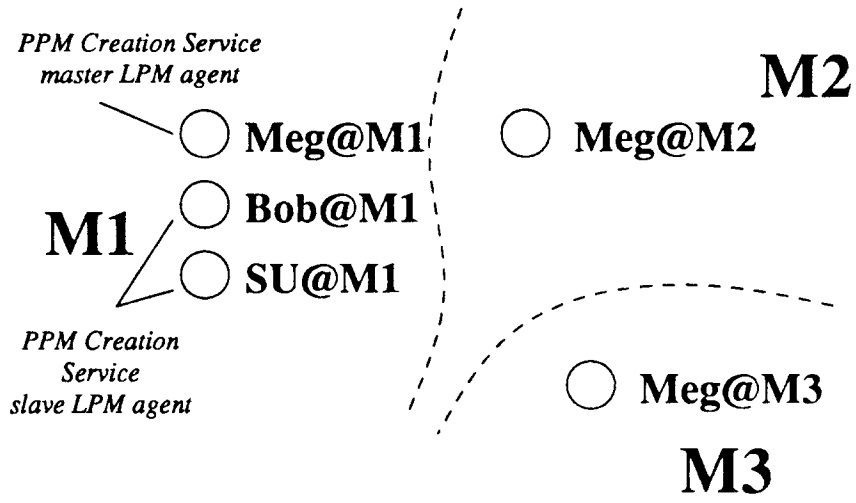


Figure 5.3 LPM agents with different identity tags may cooperate in providing a PPM Session for a particular owner.

that the Session Maintenance Service add the host to the session. The Session Maintenance Service contacts the PM Master on the new host, perhaps causing LPM agents on that machine to be created, and then contacts its counterpart on that machine. The Session Maintenance agents determine the membership of the combined PPM Session and inform the other LPM agents. The Creation Service is then able to create the process on the designated host. Thus, the extension of the PPM Session can be transparent to the client.

The PPM Session extension is not necessarily transparent, however, because the local LPM agents may not have the password, or other authentication information, required at the new host. If an authenticated request cannot be made, no LPM agents will be started at the new machine. In this case, the client's service request will fail, with a return code indicating the nature of the problem. Some applications will be able to prompt the user for a password, which can be made available to the Session Maintenance Service through the environment, and then retry the call to the Creation Service. Other applications may have to give up. The appropriate way to prompt a user for authentication information will vary with the form of authentication used. In a system with an integrated authentication system, the user may not have to be concerned about session extensions. In a system where each machine uses different passwords, the user may want to run a tool that allows him to explicitly add members to the session, typing passwords as he does so.

Clients can also request that the PPM Creation Service create a process on a site that is a member of the PPM Session, but with an identity that has not been used before. The Creation Service must then request that the PM Master create an LPM agent with that identity. This also requires authentication and can fail. If the authentication succeeds, the new agent becomes a slave for the local instantiation of the Creation Service. Again, the correct way of managing identity changes from the user interface point of view may vary.

If a session member site is not being used, the site should eventually withdraw from the session. This can be requested by the agents for the Bookkeeping Service, who detects that no work has been done at the site for some specified amount of time, and that no useful bookkeeping information is being held. The Bookkeeper agent requests that the site be dropped from the

PPM Session, and notifies the PM Master that the LPM agents will exit.

5.3. Identities in a Distributed System

In describing the login/rlogin model and the PPM session model, we have assumed that identities are rather simple tags corresponding to users. We have said that, in the PPM session model a user is able to change his identity for a command session, which is analogous to the ability to log in under a different identity in the login/rlogin model. At this point, we should consider why it is that users might want or need to change identity and whether this is likely to be the case in future distributed systems.

5.3.1. Identities: Mapping from Client to Access Rights

Servers managing resources often wish to limit access to those resources according to some access control policy. In general, the problem of access control is one of mapping a client process to the set of rights to accord that client. Ordinarily, this is not done directly, that is the client process is not specifically granted a set of rights. Instead, an indirection is introduced; the client is mapped to an identity, I, and this identity is mapped to a set of rights. This approach to the access control problem can be expressed as

$$Client \rightarrow I \rightarrow Rights.$$

Access control can be done without the notion of identity by using capabilities as an indirection mechanism, that is by performing the mapping

$$Client \rightarrow Capability \rightarrow Rights.$$

Using capabilities without the notion of identity, however, makes certain reasonable access control policies hard to implement. Capabilities may, therefore, be used simply to encode identities as presented here. Even then, it would still be difficult, for example, to retract a particular users's rights to some set of resources, given to some set of users, without disrupting other users in the set.

The mappings from client to identity and identity to rights may require a number of intervening steps, particularly in a distributed system where security is a concern. In a single site system, the client may be tagged with an identity that is directly accessible by the server. In a distributed system, the client's request to the server may be tagged with an identity by a trusted kernel. However, if the client's kernel is not trusted by the server, the mapping from client to identity requires a more elaborate mechanism. We can express the idealized mapping as follows:

$$Client \rightarrow I_{Key} \rightarrow I_{Ver} \rightarrow I_{Auth} \rightarrow I \rightarrow I_{Acc.Groups} \rightarrow Rights.$$

The client is able to demonstrate its identity, by virtue of having a secret key; thus, we say that it has identity I_{Key}. The key is used by the client to construct a verified identity, I_{Ver}, that can be sent to the server. This verified identity might be encrypted with the key, or might simply include the key as a password. The server must authenticate the request, that is map I_{Ver} to I. Authentication may be done with the assistance of a distinct authentication server, a public key server, an internally stored list of passwords or by other means. For generality we will say that an authentication service allows the mapping of I_{Ver} to an identity I_{Auth}. This identity corresponds to the statement "the authentication service says that the request has identity I." Mapping from I_{Auth} to I requires the server to *trust* the authentication service. Access rights are often easier to administer when granted to groups, rather than individuals. The identity I may carry implicit rights that the server must discover by mapping the identity to a set of access groups. It is the identity $I_{Acc.Groups}$ that is used, finally, to decide whether the client's request is

to be allowed.

5.3.2. Identities: Owners, Roles, Groups, and Authenticators

The identity *I* can carry four types of information. An identity can identify an *owner*, a set of *roles*, a set of *groups*, and an *authenticator*.

owner: a particular *person*. An identity should indicate no more than one owner.

role: a set of rights that a user *explicitly* chooses to use or not to use. Each user typically has a default, or *primary role*, that he ordinarily plays, but he may at times wish to play other roles. He might use a more powerful role to perform some system administrative function, or a less powerful role, for example to check that a resource really is publicly accessible.

group: a set of rights that is *implicit* in a role. It is possible for these to be encoded directly in *I*, relieving the server of the necessity of mapping from *I* to $I_{Acc.Groups}$.

authenticator: an agent that verifies an identity. An identifier for this agent may be explicitly encoded in an identity to allow access control policies to take lack of trust into account.

A client process can be said to have identity I_{Key}, that is it has whatever identity it is able to create verifiable requests for. Identities which cannot be authenticated are irrelevant. This can pose problems when remote interactions are introduced into systems not designed for them. If no authentication is done to verify the identity, complex identities with long lists of roles and groups are not difficult to administer. If identities are authenticated, however, administration is more difficult. One possibility is that each role or group specified requires a separate secret key for it to be included in I_{Ver}. Another possibility is to keep a key only for the owner role; role and group information would be treated by the server as hints. As part of the mapping from *I* to $I_{Acc.Groups}$, the server could verify that the owner has the right to play a particular role or act as a member of a particular group.

The identities actually used in systems generally do not neatly separate owner, role, group, and authenticator information. Figure 5.4 shows the components of identities in several systems. MULTICS [Sal74] and UNIX [RiT74] [RiT78] [CCC84] are influential single site operating systems. V [Che88] is an operating system for use in local networks of workstations. DASH [AFR87] is an operating system, currently under development, for use in very large distributed systems. Kerberos [MNS87] is an authentication server for a network of independent systems. DECSRC refers to a proposal for a global name service developed by researchers at DEC's Systems Research Center [Lam86][BLN86]. Each of these identities carries a certain amount of owner and role information and some include group information as well.

We have spoken of an identity, thus far, as simply a tag identifying a user. Identities in V and in DASH are of this form. These tags are authenticated in different ways in the two systems. In V, all kernels are considered trustworthy, and so the client's kernel can include the user identifier in the header for each outgoing message. DASH limits the scope of trust by requiring remote kernels to authenticate the user identifiers attached to messages through public key encryption.

A user identifier has three drawbacks that have caused many systems to use more complex identities. The first drawback is that the server may wish to base access control upon I_{Auth} rather than *I*, because it wishes to implement a specific trust policy. The second is that simple identifiers have limited capacity for specifying a role. The third is that the use of simple

MULTICS	User + Project **restricted to** Compartment
UNIX I	Real User + Effective User
UNIX II	Real User + Real Group + Eff. User + Eff. Group
UNIX III	Real User + Real Group + Eff. User + Eff. Group + Group1 + . . . + GroupN
V	User
DASH	User
Kerberos	User **restricted to** Instance @ Realm
DECSRC	User @ Realm @ . . . @ RealmN

Figure 5.4 User identifiers for various systems

identities places the burden of mapping from I to $I_{Acc.Groups}$ on the server rather than the client.

5.3.2.1. Identifiers with Explicit Authenticator Information

Both the Kerberos identity and the DECSRC global name space identity are relative to an authentication *realm* or *realms*. Making this realm explicit allows the server to implement its own trust policy as part of access control.

Kerberos identities are contained in encrypted "tickets," which are obtained by the client and sent to the server. Since the server does not directly contact the authentication service, it is necessary to place the realm explicitly in the ticket if the server is to implement a policy other than complete trust for mapping I_{Auth} to I. Kerberos does not specify any particular limit trust policy for the server to follow.

In the DECSRC global name space clients and servers are considered to be nodes in a hierarchical name space, which may be augmented by nonhierarchical links. The name space is spanned by a set of pre-existing secure channels following the pattern of the augmented hierarchy. A client requests a server by specifying a path through the name space. A secure channel between client and server is formed by composing the channels along the intervening path. A server, therefore, can map a client request to an identity I_{Auth} containing identifiers for all the realms used in the establishment of the secure channels. Unlike other systems that (trustingly) map I_{Auth} to an identity independent of realm, this scheme maps I_{Auth} directly to a set of rights. This requires that access rights be expressed in terms of I_{Auth}. If a client can access a server through two different realms, access lists would have to state the rights for the client for both paths. This allows the rights to differ, but makes it more difficult to ensure that a particular client has the correct rights. It is claimed that this can be done, and further, since the path from client to server is self-selected, that the path can be used to encode role information. By providing a variety of paths through the name space from client to server, it is claimed that the client can choose a path that results in an appropriate set of rights.

5.3.2.2. Playing Different Roles

Simple identities have a limited ability to encode role information. If the identity indicates the *owner* of the client, the rights that are exercised must be presumed to be those of the owner's *primary role*. The client may require a different role for one of two purposes. The first is to gain rights ordinarily *unavailable* to the owner, the second is to select from among the rights available to the owner a *subset* that he wishes to exercise.

In some systems a process when executing certain code is granted greater rights than are ordinarily granted to the owner. This is particularly true in older systems where long running server processes were avoided. Instead, mechanisms were introduced to increase privileges. In MULTICS, trusted programs were given additional privileges through the protection rings mechanism implemented in hardware [ScS72] without changing the client's identity. The UNIX system allows programs to be marked as "setuid" by the file system, in which case they are run with the rights of the file creator, rather than those of the process's owner. The UNIX identity available internally to a program includes both a *real user* identifier for the process's owner and an *effective user* identifier for the role being played. This allows a process, running a mail program for example, to implement access controls internally. The real user identifier is not used in implementing the file system's access control mechanism.

As noted, mechanisms like setuid programs are an avoidance of the client-server model. They do not mix well with accesses to remote servers. In Sun Microsystems' implementation of remote service access, the real user identifier is not made available to the server [SSS86]. This means, first, that the server does not have owner information. It also brings up the question of how a client is to demonstrate its identity if authentication is used. A client's identity derives from I_{Key}, that is it can only exercise rights for identities for which it has the appropriate secret keys. It is not clear how a client can temporarily be given additional effective rights only when running a particular program.

The second reason that we may not wish to map an owner to a single set of rights is that the owner may require rights that he does not ordinarily wish to exercise. An owner may at times perform system administrative functions requiring great powers. Investing the user's primary role with these powers, however, can be dangerous. On the other hand, a user might want to deliberately limit the powers of a particular process. He might wish, for example, to check that a file is publicly readable. Or he might wish to run an untrusted program. This is not a pressingly felt need today, but with the rapid increase in the informal distribution of software, sometimes containing "Trojan horses" and "viruses," compartmentalization is likely to be an important concern in the future.

Systems that use simple identities cannot accommodate nonprimary roles. They often, therefore, resort to the introduction of *pseudousers*. UNIX, as has been mentioned, has a most-privileged user identity called the *superuser*. A system administrator wishing to perform functions such as creating new accounts will run jobs tagged with the superuser identity. This is done by creating a superuser login shell. In this case, all owner information is lost. This can make accountability difficult to maintain in a system where more than one user can act as a particular superuser. The lack of accountability is particularly troublesome in a large network, in which a large number of owners may be allowed *legitimately* to play the role of a pseudouser.

The MULTICS system and the Kerberos authentication server both attempt to express restricted rights, preserving owner information in the process. In MULTICS each process includes a *compartment* in a process's identity. The *compartment* can be used to provide subsets of a user's rights. Unfortunately, the compartment was put to other uses as well. A terminal session in MULTICS runs as a single process that calls user programs as subroutines [MMM72]. The

user can run batch processes as well. The compartment is used to distinguish between the interactive process and the batch process, reflecting no difference in the role to be played. This makes access control based upon compartments harder to apply consistently. The Kerberos authentication server for Project Athena enables a client to provide to a server an authenticated ticket containing a user identity. This identity identifies both a user and an instance. The instance can be used to indicate which set of rights the user wishes to exert. Kerberos, itself, does not specify what access control policies servers are to implement.

5.3.2.3. Identifiers with Group Information

We have noted that identities can encode group information. MULTICS includes in an identity a *project* identifier, which adds a set of group rights to those available to the user. It is assumed that each user belongs to a single group. MULTICS access lists allows rights for resources to be given, or denied, to any number of users and projects. The UNIX system [RiT74] (UNIX I in Figure 5.4) at first radically simplified the MULTICS identity and access control schemes, eliminating project identifiers and the granting of rights to groups. Later versions of UNIX [RiT78] (UNIX II) reintroduced the concept of project roles, here called *groups*. A process's identity included a *real* and an *effective group* in addition to the real and effective user. Berkeley UNIX [CCC84] (UNIX III) changed process identities to include an arbitrary set of additional groups specifying additional rights implied by the *effective user* role.

The greater number of groups is required because of a fundamental difference between access control in MULTICS and UNIX. In MULTICS, access lists are lists of patterns and associated permissions. These lists can include many users and projects. MULTICS therefore places responsibility for facilitating sharing on the server rather than the client. A process in MULTICS runs several programs over the course of its lifetime, all with the same role triplet. It is assumed that this triplet, with a single group role, will express all the inherent rights of the user. In UNIX, on the other hand, access lists have only three entries (two in older versions [RiT74]), first the rights of the owner, second the rights of the owning group, and third the default rights for all others. Responsibility for facilitating sharing is therefore shifted from the *resource owner* to the *client process*. Instead of allowing many groups in an access list, many groups are allowed in the identity. This choice creates problems in a system requiring authentication, because each group in the identity must be authenticated. This may require that each group have a separate secret key, as if it were a pseudouser role.

5.3.3. Identities in Future Systems

In the previous sections we have discussed the structure of identities and the purpose of the information that they encode. At this point we will hazard some assertions about the appropriate form for identities in future systems.

- Owner information should be encoded in all identities. Owner information is important for maintaining accountability. No operation should be ascribable only to a pseudouser. Ownership alone should not imply any rights, allowing users to take on identities with restricted rights.
- The user should be able to select a set of roles. A role should be a globally recognized identifier that indicates to a server that the user wishes to exercise a particular logical set of rights. It is undesirable to require separate keys for each role, because such keys would have to be shared, making security hard to ensure, and revocation difficult. Thus, roles should be administered as *subsets* of a user's rights, rather than as *pseudouser identities*. This means that an owner authenticates himself, providing roles as hints. The server must decide whether a particular user has the right to play a particular role. MULTICS and Kerberos

allow the user to express subsets, but are restrictive because they only allow a single role (that is compartment or instance) and because these roles do not have globally valid identifiers.

- Servers should also be able to use private groups to administer access rights. These groups would not have globally valid identifiers, and would not be part of identities. Making the server responsible for mapping from I to $I_{Acc.Groups}$ increases the server's autonomy, allowing more flexible management of resources. Servers must be able to perform this mapping anyway, if roles are to be administered as subsets, rather than as pseudousers with separate keys.
- Authenticator information is important in implementing a trust policy, but mapping I_{Auth} to I should be done as a step separate from mapping I to rights.

Servers in very large distributed systems will need more flexible access lists than are afforded by the UNIX access control model. This is because access rights to many resources will have to be given to many limited sets of clients. The UNIX model grants rights to a resource owner, an owning group, and to the remainder of the world. This is satisfactory only when the "world" is rather small and rather well trusted. This is not the case for resources accessible through very large networks. Thus, rights will have to be expressible for subsets of the world that are not a resource's owning group.

The limitation of user-group-world rights is already apparent in versions of UNIX for networks. Berkeley UNIX performs remote authentication by attempting to map a remote user identity to a local identity, for example mapping **Meg@M1** to **Meg@M2**. This is allowed if M2 trusts M1, that is if M1 is on a list of hosts "equivalent" to M2, or if **Meg@M2** has placed **Meg@M1** on a list of equivalent identities, in the file *~Meg/.rhosts* available at the target host M2. Other identities can also be made equivalent, and this mechanism is often used to allow other users to exercise owner rights over a resource. The new user is simply added to the *.rhosts* lists for however long he may need access. Unfortunately, this mechanism gives the new user all of the owner's rights, while providing no accountability. Furthermore, it is possible to "chain" identities, so that if Meg is in Bob's *.rhosts* list and Bob is in Alice's, Meg can become Alice. This mechanism has been found to allow severe breaches of security. Yet, the *.rhosts* mechanism is the only convenient means of granting to a nonowner resource access rights that are not granted to the world at large. In a large distributed system, a richer means of expressing access rights is therefore needed.

5.4. Identities in PPM

PPM accommodates multiple identities within a single owner's PPM session. This means that processes with a variety of identities may be legitimate clients of the same LPM agent. Hence, the access control used by the PPM Services must be flexible enough to allow this legitimate access. We have noted in the previous section some of the reasons that a user may need to change his identity, and speculated that users of future systems will continue to require the ability to play different roles. In this section we consider the identity attached to a PPM Session, the identity attached to a command session, and the access control provided for the PPM Services.

5.4.1. Ownership of a PPM Session

A user owns a single PPM session that extends across a number of machines, which offers services to the owner's programs through a single interface. Such a session, belonging to a user Meg, was illustrated in Figure 5.2. This session is provided by a set of agents with various identities on each machine running (see Figure 5.3). In extending Meg's PPM Session to a new machine the sole problem is authenticating Meg's identity to the PM Master Service on the new

machine. This authentication allows the creation of LPM agents, including a master agent for the PPM Creation Service.

The first question to be addressed is what identity Meg has on a new machine. In a system with universally recognized identifiers, such as those envisioned by [MNS87] and [AnF88], Meg can have a single owner identifier with a single key. On systems with machine specific identities, such as Berkeley UNIX, an owner will have different names on different machines. These might be quite dissimilar, for example, Meg might have the names **Meg@M1**, **Margaret@M2**, and **MFG@M3** on various machines. We will assume that on any machine the owner will have either no identity or a single *primary identity*, and that some function exists that allows one to map from an owner's primary identity on one machine to that on another.

The second question to be addressed is how the PPM Services handle authentication. This is necessarily system dependent. The PM Master Service on each machine relies on the authentication mechanisms prescribed for the underlying system. This may rely on encryption or may rely upon passwords. The PPM prototype for UNIX relies on precisely the authentication mechanisms used for logins and remote logins. In the prototype, a session is started when the user types a password to a system-owned login program. The session can expand to new machines using the appropriate primary identity on the new machine. The new machine either accepts the identity on the original machine as equivalent (using the equivalent host and *.rhosts* mechanisms) or challenges the original session for a password. This challenge is sent to the LPM agent for the PPM Session Maintenance Service on the original machine. If the service has the correct password, it returns this in a reply message. If the service does not have the correct password, the expansion fails. This in turn may cause the operation that resulted in the expansion to fail, with a return code indicating the reason for the failure.

The PPM Session Maintenance Service obtains the local password from the PM Master Service when it is created. This is a different policy from that followed by the current UNIX login program which throws away passwords. In a client-server system secret keys must be kept. Additional keys can be given to the Session Maintenance Service through the *smAuthInfo()* call. This information is not shared automatically with Session Maintenance agents on other machines. Because the lack of authentication information will cause operations to fail, care must be taken to design user interface applications that will appropriately prompt the user for passwords or other information.

It is possible to extend a PPM session to include a machine where the user has no primary identity. The owner has no rights on this machine, but may wish to play the role of some other user with rights. For example, an owner, Meg, might wish to copy a file belonging to Bob from a machine on which she has no account. In Berkeley UNIX, Bob can add Meg to his *.rhosts* list on the appropriate machine, enabling Meg to remote login as Bob. In PPM, Meg and Bob each have separate sessions that should not be merged. Instead of creating agents belonging to Bob on the new machine, the session is extended under the identity **nobody**, which is a special, unprivileged identity, not requiring authentication. It is possible for the PM Master Service on a particular machine to reject extensions under this identity. LPM agents are created for **nobody**, but are recorded by the Process Master as belonging to a primary identity supplied by the caller. The Process Master will create new LPM agents for new primary identities, but will not create several sets of agents for the same identity. Thus, several users can create **nobody** agents, but a single user will only have a single set of agents. (This requires no enforcement, since it is simply to ensure the correct functioning of the PPM services.) The master LPM agent for the PPM Creation Service has the **nobody** identity and, hence, no privileges. To run processes on the machine, an additional slave agent must be created for the Creation Service with a real local user identity. This occurs when a command session within the PPM

session takes on a real local identity, as described below. The Process Master may choose to destroy a user's **nobody** agents if they fail to acquire a real local identity within some time period.

Thus, a PPM Session contains agents on each machine with the user's primary identity or the special identity **nobody**. When the session is extended to a new machine, the session will merge with any existing session that includes the new machine and has the same primary identity on that machine. Ideally, the session can be extended when this is required by some request to the PPM Services. To this end, authentication information is preserved by the PPM Session Maintenance Service. If the proper authentication information is available, session extensions are not visible to the PPM Session's clients.

5.4.2. Command Session Identities

A command session is a logical entity maintained largely through name-value bindings in the environment. A command session has a representation in the environment that allows the values bound to names to differ between sessions. Therefore, the identity of a session can be set by attaching the identity to a name used by convention for the identity. Moreover, the names looked up can also vary between machines. A command session can therefore have a distinct identity on each machine.

By default, the identity of a command session is inherited from the PPM session. On any machine, therefore, the default identity of a command session is the PPM session owner's primary identity. In this case, the PPM Creation Service will create jobs whose processes have that identity. The user's LPM agents ordinarily have this primary identity and, so, can act with the proper privileges. If the user changes the identity of a command session, the Creation Service must begin creating processes with the new identity. To do so, the Creation Service may have to request the creation of a slave Creation Service agent with the new identity. These agents are created by the Process Master.

When the PPM Creation Service requests a new LPM Creation agent from the Process Master, it must demonstrate that the identity change is authorized. As with the initial request for LPM agent creation, this authorization may be demonstrated with a password or through encryption. Such authorization may also be found in *.rhosts* files or through a "*.rhosts* server" of the type discussed earlier.

5.4.3. Authenticating Client Requests

Client requests to the PPM services are authenticated through the inclusion of a PPM capability, *rCap*, in the request. This key is selected by the Process Master when the LPM agents for a particular machine are created. If they were created through a remote request, *rCap* is passed back to the Session Maintenance Service at the requesting site. The capability is then made available to the other LPM agents on that site. The LPM agents store capabilities for remote machines, and clients require only the capability for the local agents. Capabilities are required if if an owner's command session is allowed to start jobs under a variety of identities and if a variety of users can start jobs under the same identity. In a system that preserved authenticatable owner information for all identities, capabilities would not be necessary. This authentication method requires some means of communicating *rCap* to all the session's processes on a particular machine. The key can, for example, be placed at a particular point in a process address space. The key is used by the IPC library routines discussed in Chapter 3, rather than by user supplied code.

CHAPTER 6

THE PPM ENVIRONMENT AND ITS NAME SPACE

6.1. Purpose of the Environment under PPM

Information that is useful to or necessary for a running program may not be determinable when the program is compiled and linked. For example, a program may need to know

- the login name of the user running the program
- the home directory of the user
- the desired time to wait for some response before giving up
- the type font for its output

The PPM Environment Service provides a means of binding names to arbitrary pieces of information and recalling that information by evaluating the name. It is an alternative to the introduction of special purpose procedures for storing and looking up each new piece of information.

A set of bindings provides an *environment* for a program running in a system. The environment provides both information that is fixed at the start of a PPM Session and information that is updated during the session as part of the user's management of command sessions, or as part of running particular programs. Operations are provided to set and to evaluate a binding, that is to associate a value with a name and to look up the value associated with a name. Binding names are typically character strings that can be conveniently used in programs. Binding values can include a wide variety of things. Values may be constants, references to system objects such as file names or communication port addresses, or names of other bindings. Values can include both character strings and noncharacter information. In heterogeneous systems, values may have to be stored in a system independent format, such as ASCII strings, or a more elaborate external data representation [DeS86] . The basic design, however, treats values as either uninterpreted bytes, leaving to application programs their interpretation, or as indirections through some other binding.

6.2. Basic Design

The design of the PPM Environment Service allows it to fulfill three functions:

Information repository. -- The environment holds information that a program evaluates at runtime. This information would otherwise have to be wired into the program, supplied as command line arguments, read from a particular initialization file, or supplied by some other *ad hoc* mechanism.

Means of communication. -- Because the environment is a shared set of bindings, processes without shared memory have a simple means of communicating. This can be used to help set up other communication channels. It can also be used to update information used by programs, even information that is not expected to change in the usual case.

Point of indirection. -- The environment can serve as a point of indirection in accessing objects in global name spaces. For example, service addresses and file names can be placed in the environment. Instead of using a fixed global name directly, a program can evaluate a binding name to look up the appropriate global name to use. This can be useful

to test a new service or to let each user use an appropriate set of resources.

These goals have been addressed by placing all of the bindings for a PPM Session in a single name space, and using features of the name space to guide binding evaluation. The bindings for processes in a PPM session are organized into contexts, in a hierarchical name space. For the moment, we will ignore the actual names attached to the contexts. Figure 6.1 illustrates the context hierarchy from which two processes, represented by the circles labeled "P1" and "P2," draw their bindings. Figures 6.2 through 6.5 highlight the structure of this hierarchy.

The hierarchy comprises a root context and a set of *machine subtrees* (Figure 6.2). The root context generally holds bindings applicable to all the owner's processes, such as the owner's name. Since bindings placed in the root can be overridden at other levels of the hierarchy, default values for bindings may be placed in the root. For example, the address of a preferred printer might be set in the root, but explicitly overridden for a particular printing job.

Figure 6.2 illustrates four machine subtrees. Two, labeled "M1" and "M2," are ordinary machine subtrees. They contain bindings applicable to processes running on machines M1 and M2. "AllH" and "AllL" are special subtrees, containing bindings applicable to processes running an all machines. P1 and P2 are two processes running as part of the same job on the machines M1 and M2. P1 is therefore affected by bindings in the "M1" subtree and the two "All" subtrees. P2 is affected by the "M2" subtree and the "All" subtrees. Since different values may be bound to the same name in the different subtrees, a precedence order is defined among the subtrees. The "AllH," or *AllHigh*, subtree contains bindings with *higher* precedence than bindings in the ordinary machine subtrees. "AllL," or *AllLow*, contains bindings with *lower* precedence.

The upper region of a machine subtree contains *shell contexts*, while the lower region's contexts are devoted to particular jobs. The contexts for a particular job form a *job subtree*. Figure 6.3 marks the division between shell contexts and job subtrees with a dashed line. Since jobs can be distributed across several machines, a job may have job subtrees in several machine subtrees. This means that the job's processes on different machines can see separate bindings. Thus, we find in Figure 6.3, Job1 has job subtrees for both machine M1 and machine M2, and in the "All" subtrees as well. Since P1 and P2 are members of the same job on different machines, they can share bindings in the "All" subtrees, but not in "M1" or "M2." If a third process in the job, P3, were running on machine M1, it would draw its bindings from exactly the same job subtrees as P1.

Bindings set in a job subtree apply only to processes in the particular job associated with the subtree. The subtree is created when the job is first assigned a job identifier, and is destroyed when the job ends. Thus it is possible to set bindings before any of the job's processes are created, and to evaluate the bindings after all the processes have exited. Once the job is ended, however, all the bindings in the subtree are forgotten. Shell contexts, on the other hand, persist between jobs; setting a binding in a shell context will affect future jobs until the binding is reset, or retracted.

The leaves of the job subtrees are process contexts (Figure 6.4). A process will have such a leaf context in all the subtrees from which it draws its bindings. Thus, process contexts exist for both P1 and P2 in the AllHigh and AllLow subtrees, but only for P1 in the M1 subtree, and for P2 in the M2 subtree. Process contexts contain not only bindings set specifically for the associated processes, but bindings *inherited* from superior contexts. When a process evaluates a binding, it does not look through each machine subtree, specifying each context in turn. Instead, it looks only in its process contexts. When a binding is evaluated, the process's process contexts are searched in precedence order until a value is found. This means that a binding in a

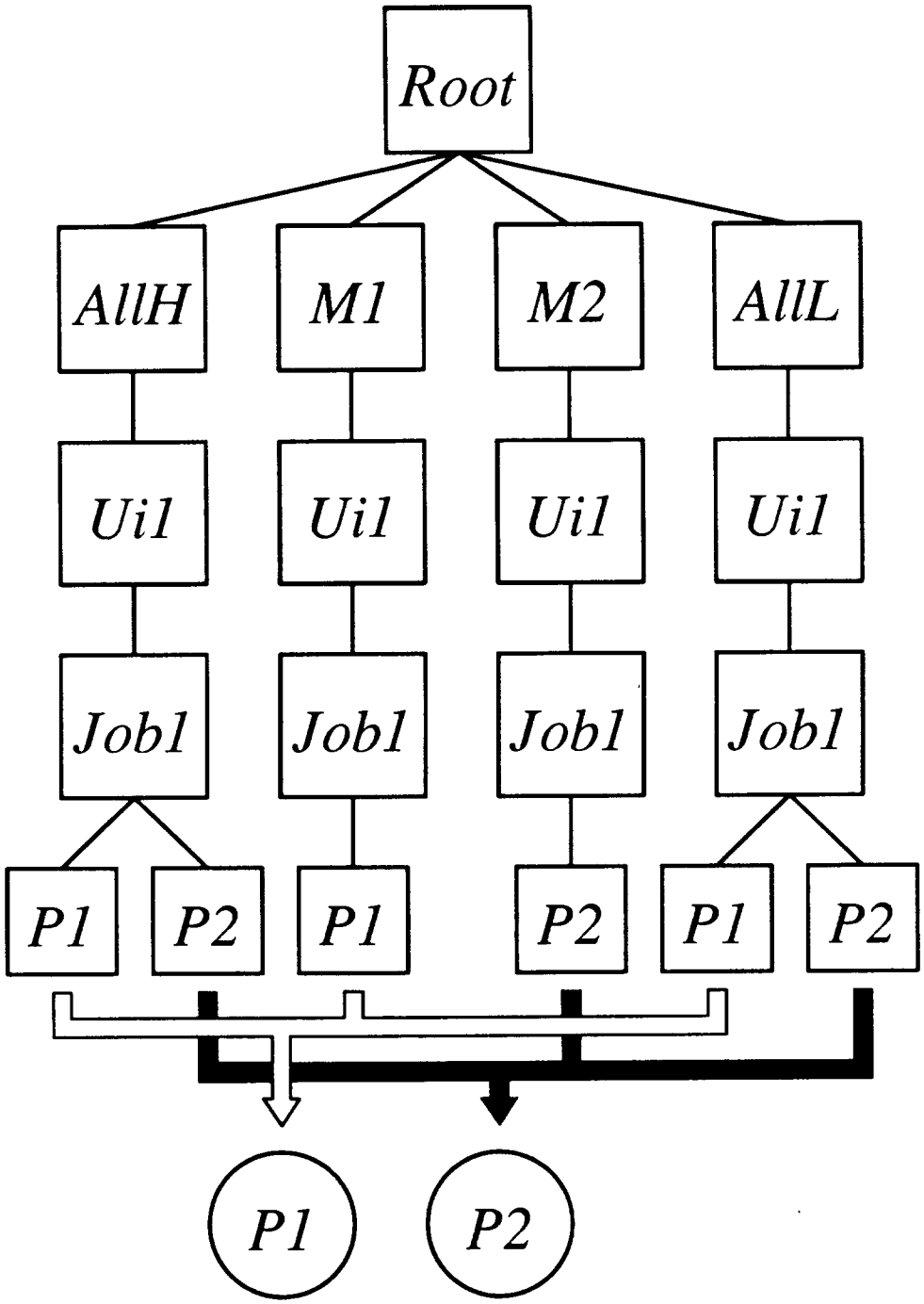


Figure 6.1. ENS naming hierarchy for bindings in the environments of processes P1 and P2.

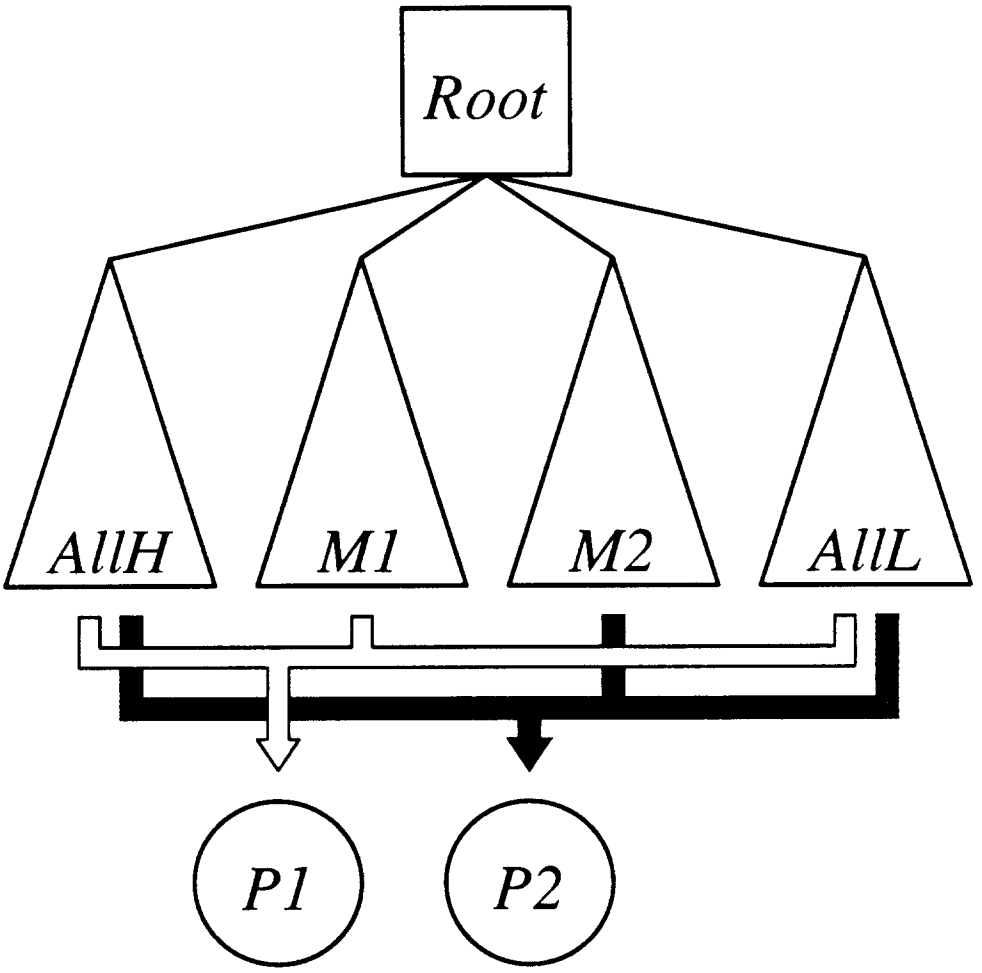


Figure 6.2. The name space comprises a root context and machine subtrees.

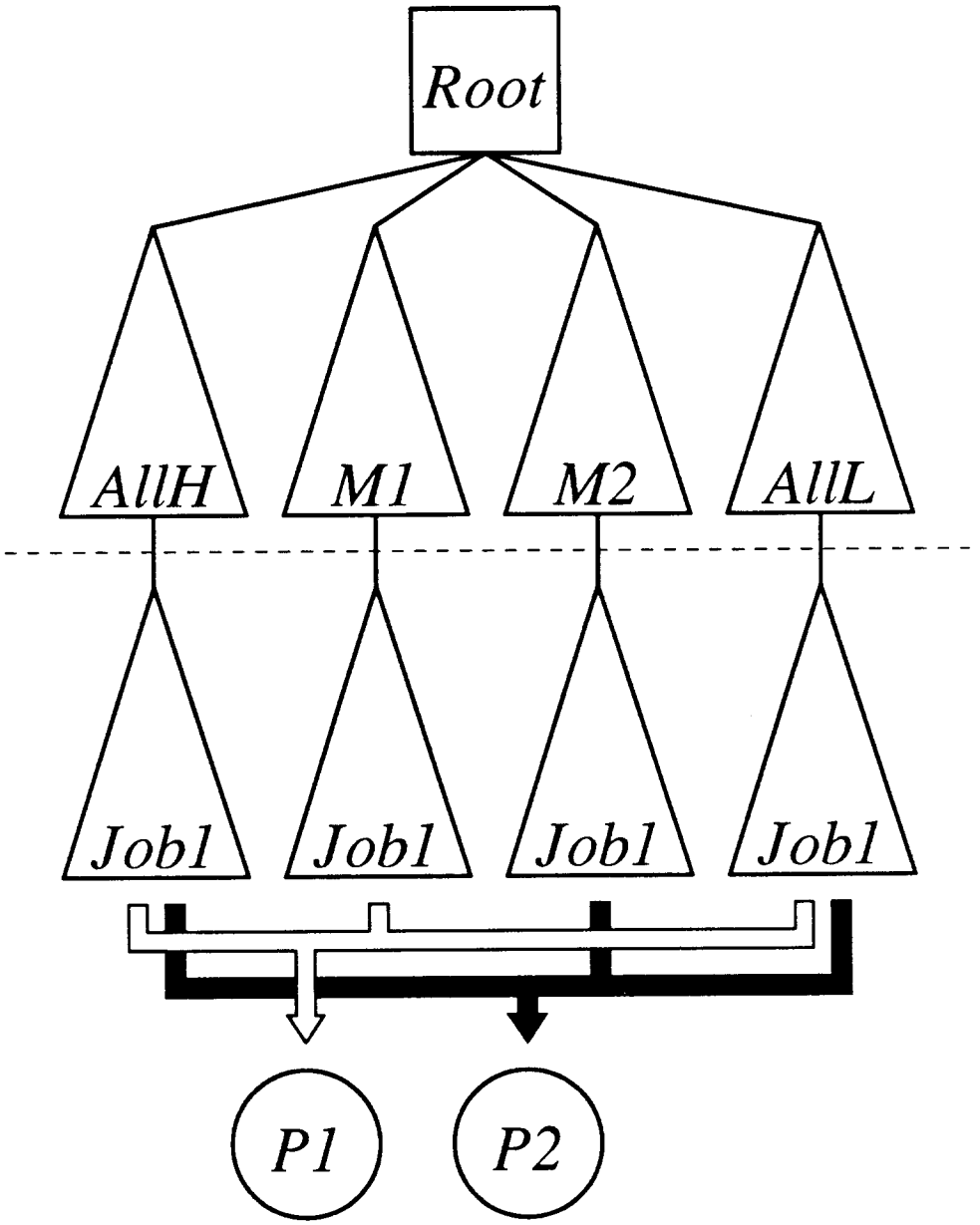


Figure 6.3. The naming contexts are divided between shell contexts and job subtrees.

lower precedence process context cannot override a binding in a higher precedence context.

There are three sources for the bindings looked up in a process context. A binding may be a copy of a binding in a shell context (*inheritance-by-copying*). A binding may be a reference to a binding in the immediately superior job context (*inheritance-by-sharing*). Finally, a binding may be set in the process context alone. When a particular name is bound in more than one

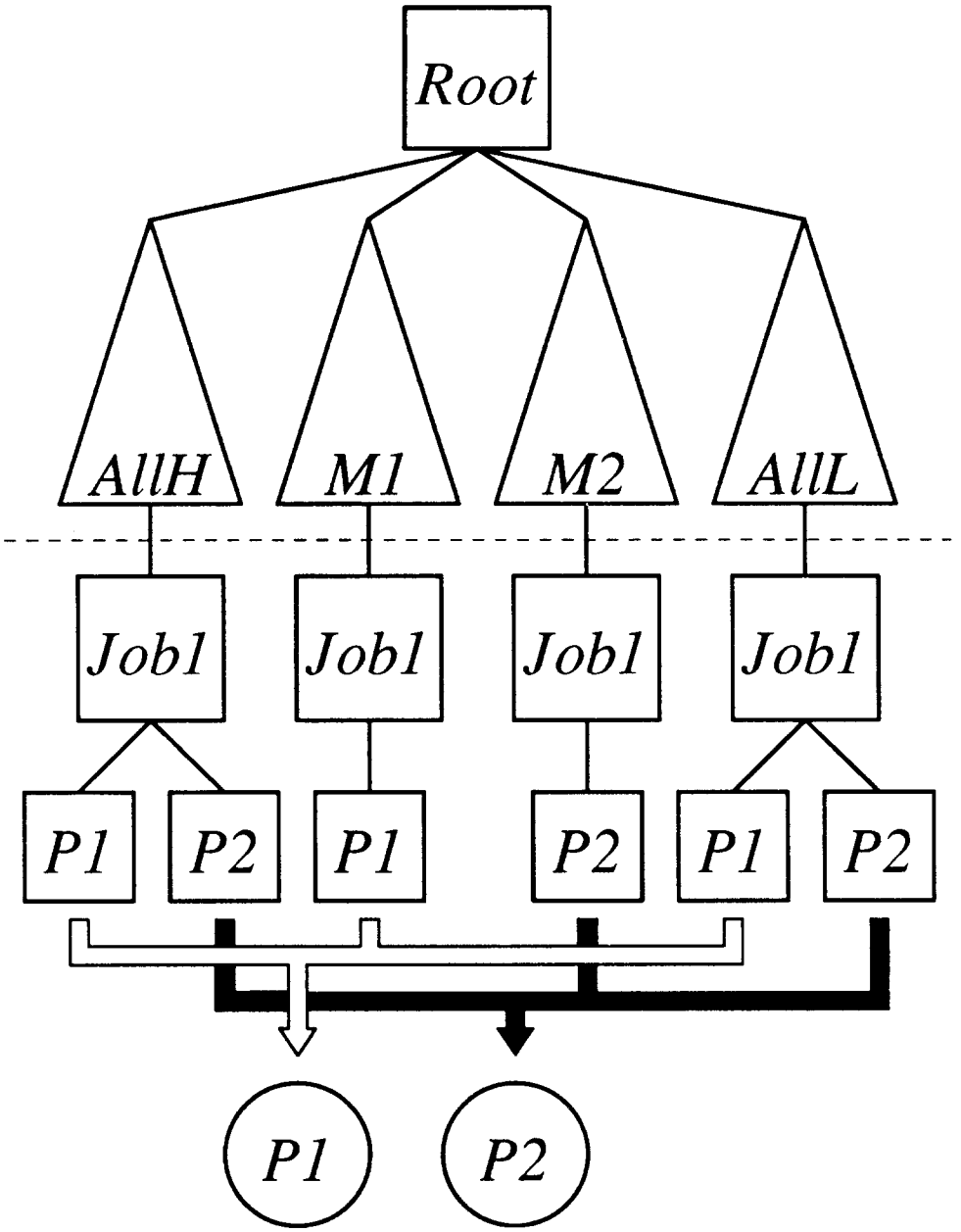


Figure 6.4. Processes obtain their bindings from process contexts. There contexts inherit bindings from superior contexts.

context, the lowest binding is the one seen in the process context, overriding any superior binding. Thus, a setting in a process context overrides any other (in the same subtree), and a binding in the job context overrides any shell binding.

The different inheritance semantics support sharing of bindings within jobs, but insulate process environments from changes in the shell contexts. A process's working directory is a binding normally inherited (by copying) from the shell. If a process changes the working directory binding in the shell context, that process and any other processes in existing jobs will be unaffected by the change, since they have independent copies. New jobs on the other hand would inherit the new binding. A process can set a new binding for the working directory for itself and all the other processes in its job, by changing the value in its job context. This leaves other jobs unaffected. If the process were to set the binding in its own process context, or in another process's process context, only the process associated with the changed context would be affected by the change. Since a process has several process contexts with different precedence, a binding in a low precedence context can be changed without affecting the binding's value as seen by the process. A binding may also be explicitly undefined in a high precedence context, overriding any inherited value, and thus exposing the lower precedence binding.

Figure 6.4 represents the shell contexts within a machine subtree abstractly. Figure 6.5 shows that these contexts form a two level hierarchy. The upper level is for bindings that are to be inherited by all jobs. The lower level is a set of command session contexts. All jobs under PPM are logically part of some command session (there is a default command session for jobs run without a specified session). A command session is often associated with a user interface program such as a command interpreter. Such a program obtains a command session name from the PPM Creation Service. This name is a component of several command session contexts in various subtrees of the name space.

A particular process, job, and command session has a separate context in several different machine subtrees. Because processes are part of particular jobs, and jobs are part of particular command sessions, each machine subtree has the same structure of command session, job and process contexts. This is shown in Figure 6.5, where the Job1 context is subordinate in each subtree to a command session context for the command session CS1. Since the job inherits all shell context state when the job is started, the notion of changing a job's user interface has no effect on the job's environment. For example, the termination of a user interface program, or stopping or starting a job from a new interface, will have no effect on a job's environment.

The command session contexts allow user interfaces, such as command interpreters, to keep their states separated from one another. CS1 and CS2 could be command interpreters, each needing to keep a binding for its own working directory. The commands run from a particular user interface should inherit the appropriate binding. If CS1 keeps its binding in a separate command session context from CS2, the commands run will inherit the correct binding, and changes in CS1's working directory will not affect programs run by CS2.

6.3. Environment Binding Names

The foregoing discussion has indicated that programs have the option of setting bindings in shell contexts, job contexts, or process contexts. How is the context of a binding specified when the binding is set? Since the contexts are arranged in a hierarchy, it is natural to use hierarchical naming. Binding names resemble file names in a hierarchical file system. A binding is referred to by a name such as */A/B/C/D*, which specifies a path from the top of the hierarchy to the context *C* and an entry name, or *simple name*, *D*. A process can both set and evaluate any binding in any context by using such a fully qualified name. Figure 6.6 shows the names of the contexts in the AllHigh subtree from which process P1 draws its bindings. The names of the two topmost contexts are fixed, while those of lower contexts depend upon the names of the particular command session, job, and process involved. These name components are identifiers assigned by the Creator service.

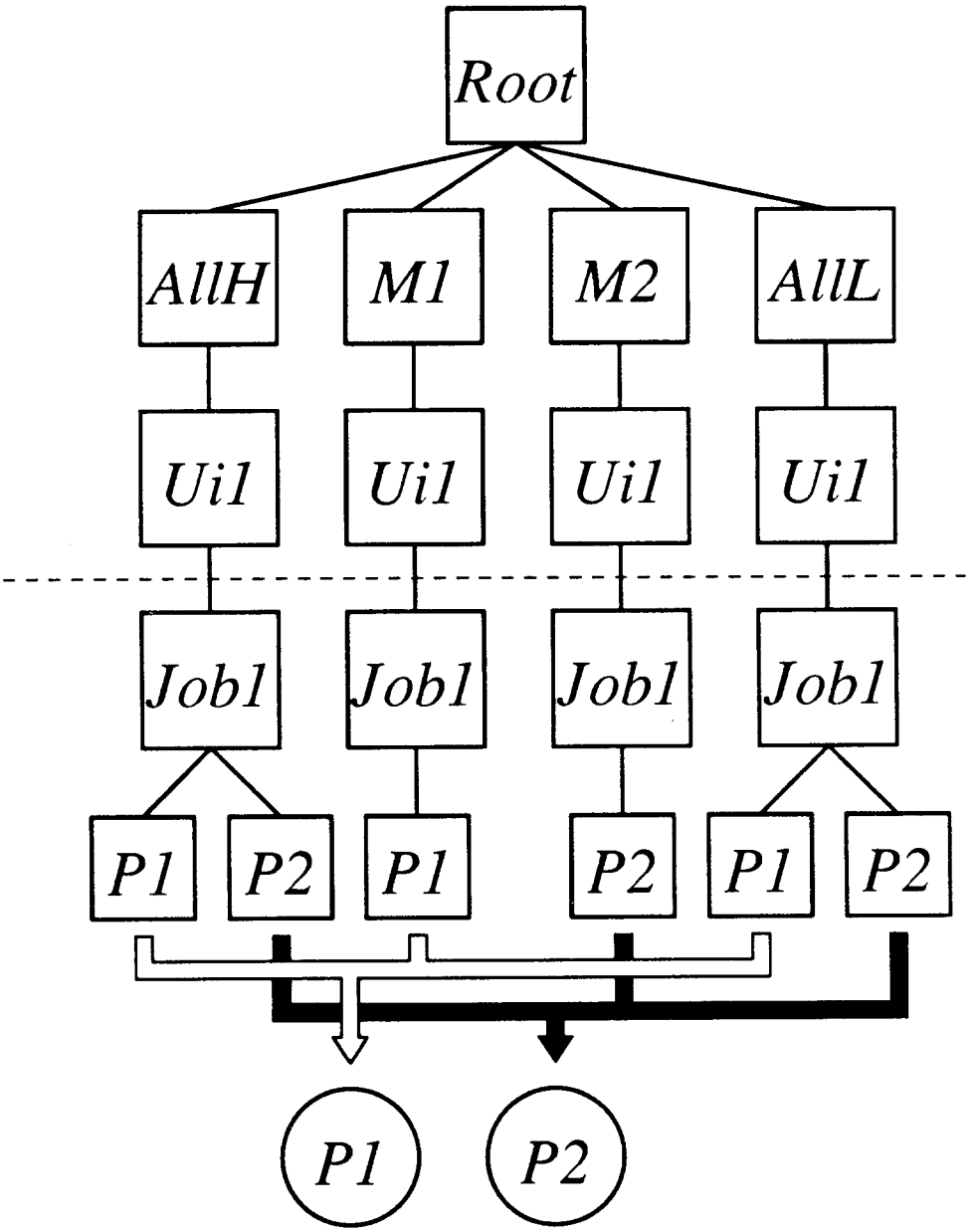


Figure 6.5. The shell contexts are divided between the root context, machine contexts, and command session contexts.

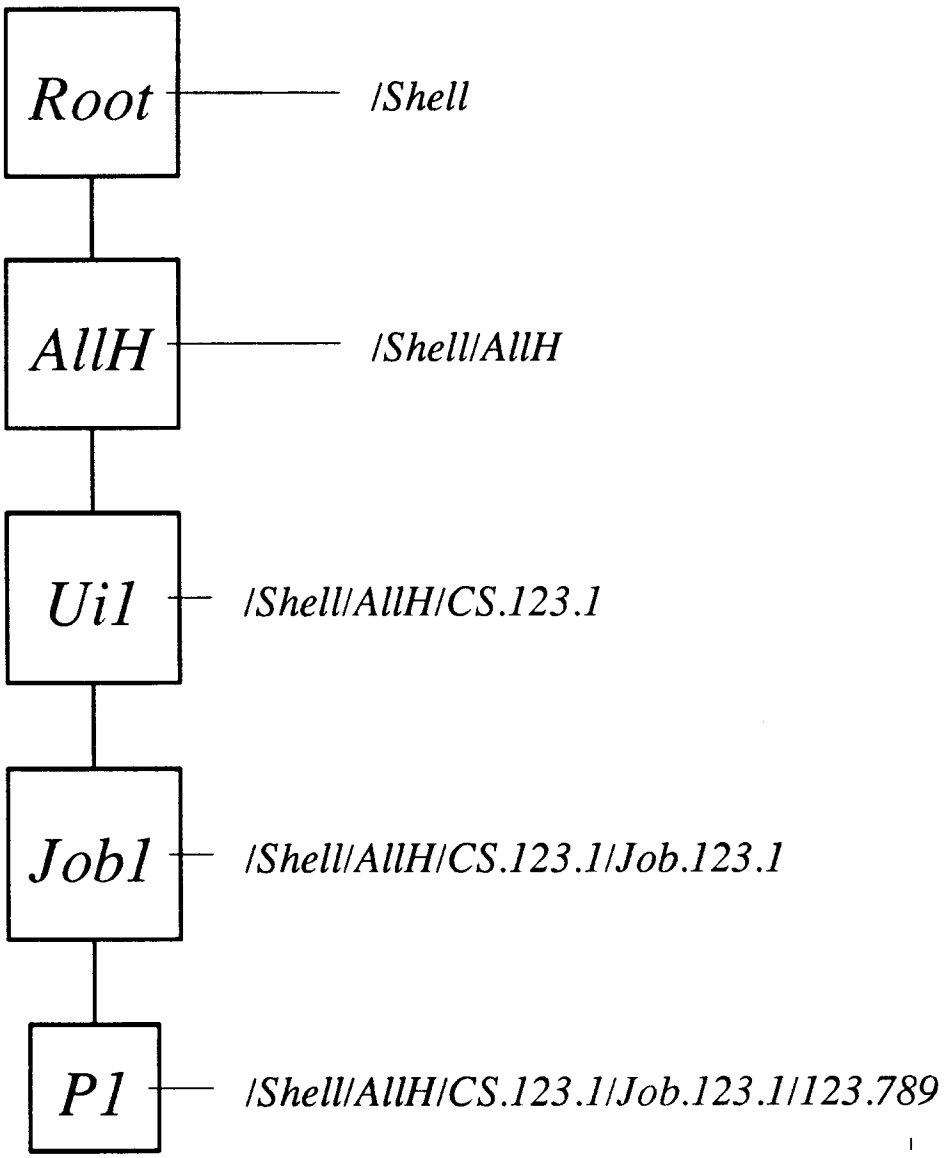


Figure 6.6. Names for contexts in the environment naming hierarchy.

The full names of contexts are unwieldy and difficult to construct. For this reason, contexts are often referred to by *nicknames*. Nicknames are strings, delimited by angle brackets, inserted into binding names. For example, the process context in the AllHigh subtree can be referred to as <*PROC*>. Nicknames for the contexts affecting process P1 are shown in Figure 6.7. By using nicknames, all the contexts affecting a process can be referred to by fixed strings. This makes it much easier for client programs to make use of environment bindings in standard ways. Nicknames also allow the use of names as binding values that are inherited by processes in different jobs under different command sessions. Nicknames thus offer a form of relative naming. It differs from the mechanisms used in hierarchical file systems, because, since each process has a number of process contexts, there is no single "current context."

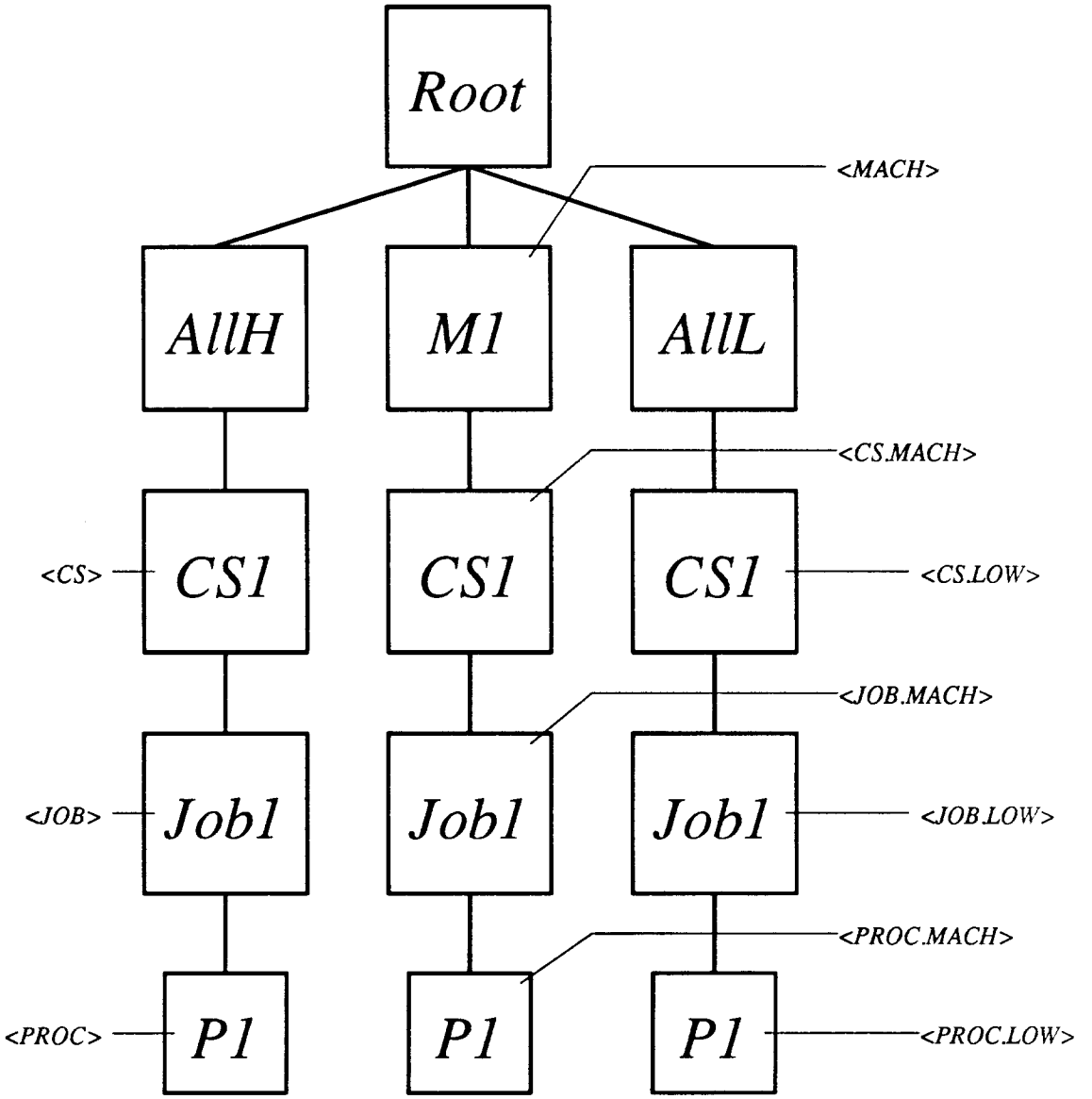


Figure 6.7 Nicknames for contexts allow references to appear as fixed strings.

The user does not have to supply a full context name or nickname, to set or to evaluate a binding. Context names are useful in setting bindings, because they specify the scope within which the binding is to be shared. In evaluating a binding, however, the sharing scope ordinarily is unknown. Bindings are therefore ordinarily looked up by their simple names only.

Two mechanisms, *inheritance* and *path lookup*, are used in evaluating a binding referenced by a simple name. As has been explained, inferior contexts inherit bindings from superior contexts. Thus, process contexts will inherit bindings set for a job or command session. Process contexts are *implicit contexts* for name resolution by the associated process. When evaluating a binding specified only by a simple name, the implicit contexts are tried in precedence order.

When setting a binding specified by a simple name, the highest precedence process context is the implicit context. The list of process contexts in precedence order is similar to the path used by the UNIX shells when expanding command names. The implicit context list for a process is therefore called its *evaluation path* or *E-path*. The contexts of the evaluation path are typically nicknames. The evaluation path is inherited through the environment, and is in most senses just another environment variable, although it is, in fact, stored by the PPM Bookkeeping Service for each process so it can be simply evaluated.

A binding can have special values in addition to the ordinary sets of bytes. The value can be an indirection through another binding name. In this case, the new name is evaluated and its value returned. A binding can also be set so that it will force evaluations to return a value-undefined token to the client. This can be used to hide values that would ordinarily be inherited, and is sometimes useful for testing certain programs. A binding can also be set to hide values in a slightly different way. An inherited value can be hidden, but without stopping the path evaluation. This exposes lower priority bindings. Again, this is useful for testing some programs. This second type of hiding can also be used to eliminate bindings, if the binding is explicitly undefined in the context where it was set.

The context hierarchy is not limited to the organization outlined above; additional subtrees can be added. Contexts are not created explicitly. Instead, if path names for new bindings contain nonexistent contexts, these contexts are created automatically. All contexts other than job and process contexts are shell contexts. Bindings in shell contexts are shared among all jobs (it is their inherited copies that are private). Bindings in contexts that are not ancestors of process contexts will not be inherited by the process contexts. Such bindings can still be accessed explicitly, by using fully qualified names. They can also be used implicitly, by adding more context names to a process's evaluation path. Additional nicknames may also be created. Strings appearing between angle brackets in binding names will, themselves, be treated as binding names. Their values will be substituted into the binding names being evaluated. For example, */Shell/<Program.Name>* will be expanded to */Shell/mail* for a process running the mail program with the binding *<PROC>/Program.Name* set to "mail." Nicknames may be set and evaluated as bindings in the environment. The standard nicknames are set automatically, and cannot be changed. Some extensions of the context hierarchy will be discussed in a later section.

6.4. Goals of the Environment Name Space

The PPM Environment name space design is an attempt to balance complexity with flexibility. The large number of contexts makes a number of sharing arrangements possible, though by no means all. Many design choices were based upon "thought experiments," which is to say a good deal derives from guesswork. It is therefore appropriate to examine the goals that led to the formulation of the Environment name space. Some of the goals were to

- present a logically unified service at the user interface
- keep shell and program state in the environment
- base a program environment on inheritance from a shell environment
- allow consistent copy sharing of bindings within jobs
- allow bindings to be set for different, sometimes overlapping, collections of processes (such as those on the same home machine, those in the same job, *etc.*)
- let names be simple character strings in most cases.

The PPM Environment name space is a hybrid of two simple models, *full sharing* and *strict inheritance*. In the full sharing model all processes share a single set of bindings. In the strict inheritance model, each child process obtains a copy of its parent's environment when the

child is created. The child is then free to make changes in its environment without affecting that of its parent or those of any sibling processes.

Strict inheritance has the advantage of allowing a name to have different values for different processes. However, lacking the ability to set bindings in other processes' contexts, strict inheritance can be very limiting. Values must be known by the parent before the child is created if the child is to inherit them. Sometimes, however, information such as a process id or a communication address is not known before a process is created. This can lead to circular dependencies between setting bindings and creating processes, making it impossible to pass all of the information through this sort of environment. A strict inheritance environment also places responsibility on the parent to set all bindings appropriately. For example, bindings to be inherited by applications from a parent shell must be set by the shell process. This forces commands that might change shell bindings to be built into the shell process. Without a means of changing, or even naming, other process's bindings, there is no alternative. In the PPM Environment model, on the other hand, a command to, say, change the shell's working directory can run as a separate process, no different than the command to list the directory's contents. Even with the ability to change a binding in another process's context, consistently updating a binding for all inheritors of the binding is a problem.

Full sharing allows the sharing of bindings, but at the expense of not allowing processes to insulate themselves from inappropriate changes in bindings. Full sharing provides some help for processes that wish to communicate with other processes in the same job, for example to communicate port addresses. However, this communication is upset if more than one instance of the job is run simultaneously. Processes can communicate with one another only if they can find names that do not collide with names used by other processes, or if they have some other means of synchronization.

Two other models were also rejected for the environment. The *file system* model presents contexts as files. Each process has a file containing bindings of simple names and values. The file name of any process's context can be constructed from its process id. Bindings can be placed into any context by any client that opens it by name and writes new bindings into it. This has the advantage of providing the insulation from inappropriate changes in bindings that full sharing lacks. Unfortunately, it requires that bindings be explicitly set for each process. Inheritance allows bindings to apply to descendent processes implicitly. The file model allows bindings to be shared if at least one process explicitly names the other, but does not allow transparent binding.

The *attribute-matching* model does not organize the bindings into contexts. Instead, each binding has a set of attributes attached to it. Each client process has a set of attributes, such as its job, its machine, the application run, and so forth. A client looks up a binding using a simple name, and is returned the binding that has the most matching attributes with no conflicting attributes. While this has an attractive simplicity, the model sketched does not work too well in practice. Inevitably a hierarchy of attributes comes into play, with, for example, the same process attribute taking precedence over the same job process attribute.

The PPM Environment model involves elements of both the strict inheritance and the full sharing model. Bindings set in shell contexts have the same inheritance-by-copying semantics as provided under strict inheritance, and sharing within job contexts provides the most important benefits of full sharing. The PPM Environment model differs from the strict inheritance model in that it provides shell contexts without specifically associating those contexts with particular processes. This allows the number of and relation among the shell contexts to be separated from the process architecture of the PPM. It also differs in that the model provides inheritance based upon the logical structure of work under PPM (that is through PPM Session,

command session, and job) rather than based upon parent-child relationships among processes. The PPM logical structure does not allow the processes within a job to be grouped into various subjob units with separate contexts. It is likely that name conflicts, and hence the need for subjob units, can be avoided within a job, so a simpler structure was thought preferable.

An important difference between the PPM Environment model and simpler alternatives is the provision of several process contexts with independent lines of inheritance. *Multiple inheritance* is useful when the applicability of properties to sets of objects is predominantly, but not exclusively, hierarchical. The collections of processes in the same job, on the same machine, and running the same program overlap, so any attempt to force these attributes into a strict hierarchical organization would make setting bindings dependent upon an inferior attribute cumbersome. Multiple inheritance has been introduced into the method (or function) inheritance scheme of several object-oriented languages, including Flavors [Moo86], Trellis/Owl [SCB86] and CommonLoops [BKK86]. In object-oriented languages, a class of objects is often a refinement of a more general class, so the definitions of general operations are often inherited by specialized subclasses. Sometimes, however, a class is a hybrid of two or more general classes. In this case the new class inherits aspects of both general classes. Owl/Trellis requires the program to explicitly resolve ambiguities that may arise when conflicting definitions are inherited. Flavors and CommonLoops define a precedence among the ancestor classes for resolving conflicting definitions. PPM Environment similarly uses a precedence ordering among process contexts to resolve conflicts between bindings in different subtrees.

6.5. Environments in Other Systems

The standard UNIX shells, that is the Bourne Shell, the C shell, and the Korn Shell (see Chapter 2), support environments based upon inheritance-by-copying. These shells are single processes, which are parent to most user processes. A process inherits copies of the shell's bindings, which are copied into its address space during a *fork* and preserved across the subsequent *exec*. User processes also pass bindings along to their children, but this is independent of the shell's control. Bindings are kept in the address space of the process associated with a context. UNIX therefore offers strict inheritance with the benefits and drawbacks discussed in the previous section.

The UNIX shells use the environment to pass information bound in shell variables, such as the user's path and home directory. Binding names are simple strings. Although these variables can be used to pass values to particular programs (such as a list of startup commands to editors through the environment variable EXINIT), they are used mostly for storing stable information about the user and display determined when the shell is initialized.

An environment manager has been proposed for MACH [Tho87] that provides each process with a copy of its parent's context or with shared access to its parent's context. The choice is the parent process's, however. A parent process can either allow the child equivalent rights to set and evaluate bindings within its scope, or it can create a separate copy of the scope that is given to the child. This does not allow a child process to change a binding for its parent unless the parent allows for this possibility. A parent shell would thus have to treat a process that changed the shell's working directory differently than the usual case where the child's working directory binding is insulated from the parent. The MACH environment manager allows the creation of empty scopes. A process could acquire access to more than one scope and Thompson suggests that this might allow a process to access either a local scope or a widely shared global scope.

As in UNIX, names in this scheme are simple strings. Because the user may have access to more than one scope, the calls to set or evaluate bindings require that the scope be explicitly

specified. This mechanism allows access to multiple contexts, but does not address the problems of organizing these multiple contexts so as to provide appropriate sharing and inheritance. Using a single shared context makes it difficult to maintain a binding for something like host machine name. This could not be placed in a context shared by processes on different machines; placing it in private contexts, however, makes it difficult to maintain, since it cannot be allowed to be inherited across a machine boundary. The mechanism allows bindings to be copied to remote child processes, but it does not provide a mechanism for separating bindings that should be exported from those that should not.

Cheriton and Mann [ChM84b] developed a distributed name interpretation model for the V system. In the V system, at the lowest level, thread identifiers are used as IPC addresses. Because these identifiers are not known until runtime, the kernel provides a mechanism for binding thread identifiers to logical identifiers. The logical identifiers of standard services are well-known; generally, they are kept in header files accessed by compilers. When a service wishes to establish a binding between a logical identifier and the identifier of a thread implementing the service, a call is made to the kernel. When a client wishes to find the address of a service, it makes a call to the kernel, specifying a logical identifier, and is returned a thread identifier. One of the parameters of the call to establish a binding specifies the "scope" of the binding. The binding is either local to a particular machine, or remote, that is the binding is seen only at other sites, or both.

Cheriton and Mann propose a higher level name resolution protocol that allows the uniform handling of names for a wide variety of servers. A name is generally a string of ASCII characters, that is divided into components. The components are parsed left to right. Often, but not necessarily, the components are drawn from a hierarchical name space. The components are interpreted by a series of context servers. Each context server may map a prefix of the name to a reference to another context server. In this case, the remaining components of the name are forwarded to this new server and name interpretation continues. The initial context server is replicated, one per user. Other context servers may be associated with specific services, such as file servers, are located by multicasts.

The V mechanisms shares some features with an environment mechanism. The low-level mechanism establishes bindings between logical names for services and their addresses, and allows a small amount of control over the scope of the binding. In particular, it recognizes the importance in a distributed system of providing bindings that are specific to a particular site. The higher level mechanism, by starting all interpretations at a per-user context server, allows names to have bindings with user specific values. Cheriton and Mann do not discuss, however, how user-specific bindings should be handled, and, in particular the possibility of bindings that are even more specific, such as per-job or per-process bindings.

Name servers are used in a variety of systems [BLN82] [NeH82] to store bindings between names and service addresses or other values. These name services differ from environment name services, primarily in that the contexts provided by the name services do not provide a grain fine enough that separate bindings can be established for each user process or for each job. To get the effect of multiple bindings, unique names would have to be generated from the names supplied to the user, for example, by extending the name with a process identifier. This sort of policy, however, makes it difficult to share bindings between processes.

CHAPTER 7

COMMAND SHELLS AND OTHER CLIENT PROGRAMS

7.1. Introduction

The previous chapters have described the services that PPM provides to client programs. These client programs include command interfaces as well as application programs.

7.2. A Simple Command Shell

The UNIX shells described in Chapter 2 all fork processes to execute the user's commands. A command shell that uses PPM has a somewhat different relationship with the processes that it creates, since they are *logical* children, rather than descendant processes. This necessitates an organization for a PPM shell, different from those of the UNIX shells.

In UNIX during multi-user operation, a system-owned process runs for each terminal or terminal window. This process runs a program that prompts for a user's name and password, and, assuming proper authentication, then begins to execute the shell indicated for that user in the password file. In the PPM shell, the program that is executed by the terminal-owning process, the *terminal handler*, functions as a copier of I/O between the terminal, and a command interpreter. The terminal handler begins by creating a socket to which other processes can request byte stream connections. It then contacts the PM master to obtain the contact addresses for the LPM agents. These agents will be created at this point, if they do not already exist. The terminal handler then establishes a new command session, of which it is part, and registers its contact address as the default input and output location for the session. It then requests the creation of a process running a command interpreter. This new process is part of the same command session, by default, and thus inherits the environment bindings registered by the terminal handler for the default input and output. These bindings are interpreted by the creation service when the command interpreter is being created. The creation service recognizes the binding value as a socket address. It must form a connection with this address so that the command interpreter will have a descriptor

7.2.1. Shell Structure

The PPM shell has two basic components, a *command interpreter* and a *terminal handler* (see Figure 7.1). The two components typically run on the same machine, which we will refer to as the *home* machine. The command interpreter has access to a terminal or a terminal emulation window [1] through an IPC connection with the terminal handler. Jobs run by the user on the home machine have access to the terminal through the same connection as the command interpreter. The user may also issue commands that result in jobs being run on other machines, as shown in Figure 7.1. These jobs also have access to the terminal, but through different IPC connections. To prevent confusion on the screen, the terminal handler arbitrates access to the terminal, under the guidance of the command interpreter, in a manner analogous to that used to

keep foreground and background jobs distinct.

The connections with the terminal handler are created on each machine by the agent implementing the PPM Creation Service. (The UNIX 4.3BSD implementation of terminal connections requires the creation of auxiliary *copier* processes. These processes serve the same function as those used for *rlogin* as described in Chapter 2.) The descriptors for these connections can be shared with child processes of the Creation Service agent, including both the command interpreter and processes that are part of the user's jobs. The Creation Service agent on a particular machine may hold connections to several terminals, if the user, say, has several terminal emulator windows. The agent interprets bindings in the environment to decide which terminal connections child processes are to be given. The agent may discover that it has no connection to the proper terminal, in which case it forms a new connection.

7.2.2. Starting the Shell

In UNIX during multi-user operation, a system-owned process runs for each terminal. This process runs a program that prompts for a user's name and password, and, assuming proper

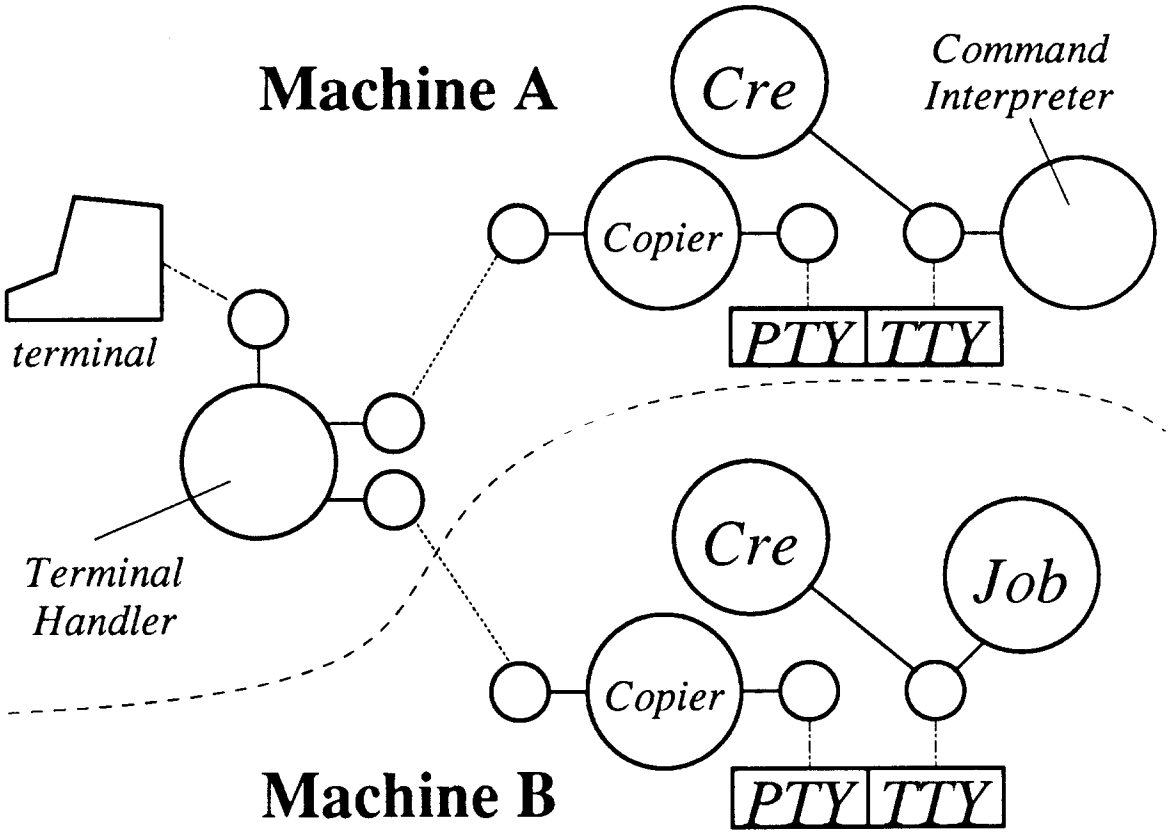


Figure 7.1 Structure of a shell using PPM

[1]I shall use the word *terminal* to mean any sort of device that acts as a terminal, including a terminal emulation window.

authentication, changes its ownership to the specified user and begins to execute the shell indicated for that user in the password file. In the PPM shell the original *login* process begins executing the code for the terminal handler. The terminal handler begins by creating a socket to which other processes can request byte stream connections. It then contacts the PM master to obtain the contact addresses for the LPM agents. These agents will be created at this point, if they do not already exist.

In the previous chapter, we introduced the concept of a *command session.* Associated with a command session is state information that will be inherited (by default) by the jobs that are part of that session. The terminal handler establishes a new command session, to provide a distinct context for jobs using the terminal. The terminal handler registers the new command session with the bookkeeper, and is registered as a new job under this command session. The terminal handler registers its contact address as the default input and output location for the session under the name *<CS.LOW>/StdInIs.* This is in the low-precedence default context for the command session (see Figure 7.2). This information is used by Creation Service agents who have no terminal connection for this command session. The agent on a particular machine creates such a connection the first time it is needed, and stores information about it under the name *<UI.MACH>/StdInIs.* This overrides the previous binding on the agent's machine. On other machines the binding has the original value, or some other machine-specific value.

After creating the new command session, the terminal handler then requests the creation of a process running the command interpreter. This new process is part of the same command session, by default, and thus inherits the environment bindings registered by the terminal handler. The Creation Service agent looks up the binding *StdInIs* for the new process. Since the command interpreter is the first process created for the new command session, the agent will create the necessary terminal connection. After the command interpreter has been created, the terminal handler copies I/O between the terminal and the terminal connections with processes on various machines.

7.2.3. The Command Session

The command interpreter prompts the user for commands and executes them. Some programs that have to be built into the UNIX shell can be written as separate programs, because the command interpreter itself does not need to keep internal state information. Instead, such information is kept in the environment, principally in the contexts *<CS>*, *<CS.MACH>*, and *<CS.LOW>*. This state information can be accessed and modified by other programs.

This externalized state makes it easier to fit together different interactive programs to provide a more flexible user interface. Because the user often works at a windowed interface, commands may be selected from menus as well as being typed. Information may be displayed continuously, rather than simply in response to commands. It is much simpler to provide this flexibility through a number of separate programs, than through a single program with many options.

Programs can be executed in the foreground or background. When a job is executed, it is given access to the terminal, both as a source of input, and as a destination for output. This requires that the terminal handler direct output from the job's terminal connection to the screen, and input from the keyboard to the job's terminal connection. This is arranged by instructions passed from the command interpreter to the terminal handler (by a message, rather than through the terminal connection). These instructions include the identity of the foreground job. The user should be able to terminate or suspend this job by typing a combination of keys (typically <CNTRL>Z to suspend, and <CNTRL>C to terminate). In UNIX, these key combinations generate signals through the terminal driver. Under PPM, these key combinations are intercepted

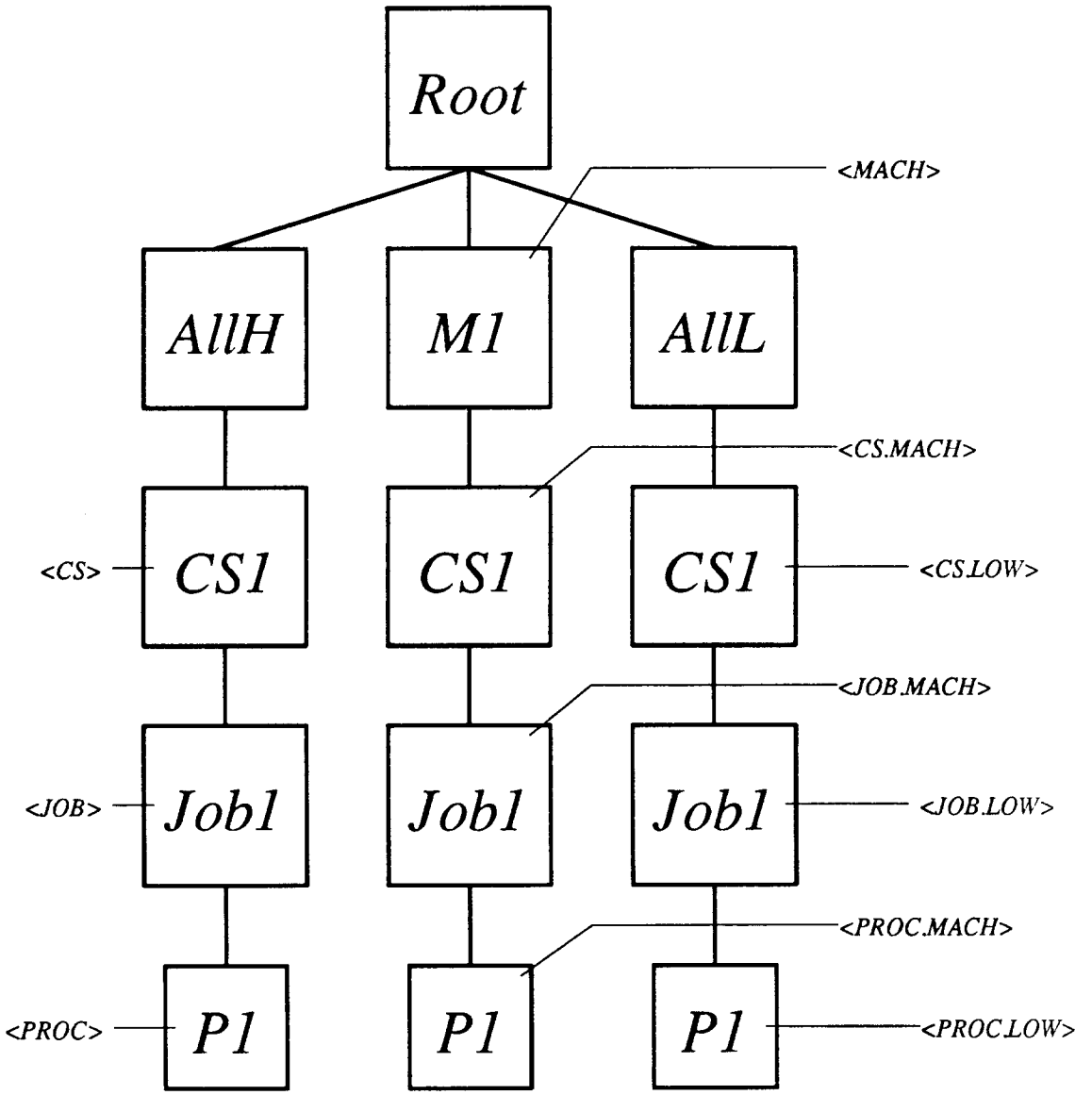


Figure 7.2 Environment contexts (identical to Figure 6.7)

by the terminal handler, and a request is sent to the creation service to suspend the foreground job.

When the foreground job terminates (or is suspended), the Bookkeeping Service is informed, and it, in turn, informs the command interpreter, which registers its interest when creating the job. The command interpreter instructs the terminal handler to direct the terminal input to the home machine's terminal connection. It is possible that the user may have typed characters, while the job was running, that are intended for the command interpreter. It is necessary to clear these characters from the remote machine's terminal connection and either redirect them to the home machine's connection, or to throw them away (effectively disallowing "type

ahead''). This task is done by the copier process for the terminal connection, under the direction of the terminal handler. When switching the destination of output, the terminal handler sends a request for the copier process to read all available input from the destination descriptor, and send it back to the terminal handler, where it is rerouted to the correct recipient. Ordinarily, there is no such input.

Background jobs are not allowed to read from the terminal, and will, in UNIX, receive a SIGTTIN signal. The output of a background job is ordinarily placed into a file, rather than printed on the screen. These files are deleted at the end of a command session if not sooner.

With window systems, it is often more convenient for the user to start jobs in separate windows, rather than multiplexing a single terminal interface between foreground and background jobs. Programs that run in separate windows may be given access to a new terminal emulator window created by the command interpreter.

7.2.4. Termination of a Command Session

Eventually, the user will want to end a command session, and will type ''exit'' to the command interpreter. Before terminating, the command interpreter instructs the terminal handler to close its connection to the terminal, to delete any output it has saved in files, and to exit. The command session does not end with the exit of the command interpreter, but persists as long as the Bookkeeping Service holds information about any jobs in the command session. The command session state includes bindings in the environment and descriptors held by the Creation Service. Information about this state is retained until the command session ends.

Each time information about a job is removed by the Bookkeeping Service, a check is made to see if this is the last job of the command session. If none of the Bookkeeping Service agents of the PPM session holds information about any other job in the command session, the Bookkeeping Service sends messages to the Environment Service and the Creation Service, authorizing them to release the state information they hold for the command session. Until the session ends, it is always possible for a new job to be started as a member of the command session, even after the original command interpreter terminates.

Ordinarily, the command interpreter is the last job left running in a command session. Most job information is released by the Bookkeeping Service in response to a request by the command interpreter. Information may be held longer, however, particularly if jobs are left running when the user exits the command interpreter. Information about these jobs will be held some period of time after the jobs end, but will released by the Bookkeeping Service after the time out period has lapsed.

The failure of a machine results in the reconfiguration of the PPM Session. It is necessary, as part of this reconfiguration, to test whether command sessions, for which state information is held, still exist. The previously existing command sessions are known to the Environment Service, but not necessarily to the surviving agents of the Creation or Bookkeeping Services. Therefore, when reconfiguring the PPM Session, the Session Maintenance Service obtains a list of command sessions, and queries the Bookkeeping Service to discover whether job information exists for the command session. If no job information is being held, the command session is terminated, and the Environment Service and the Creation Service eliminate the state information held for that session.

7.3. Command Language Issues

There are many ways in which one might issue commands to a computer. Most systems provide some sort of textual command interpreter, but more recent systems provide gestural

interfaces, where one issues commands using pointing devices such as mice. These interfaces may need to be be modified to support jobs as defined by PPM. This section is primarily concerned with ways in which textual interfaces can accommodate PPM jobs.

An interactive command session under PPM differs from an interactive session in a conventional UNIX shell, such as the Bourne Shell, the C shell, or the Korn Shell, in that (1) the job model supported by PPM is more general than that of the others, (2) the environment supported by the shell has a rather different structure, and (3) the target machine for a command session under PPM is not fixed. Relatively small modifications to the command languages used by these shells can overcome these differences to a large degree. While it is possible to build even greater flexibility into the language, this flexibility would come at the expense of programming difficulty incommensurate with an interactive language.

The following are issues for the command language:

- job units
- scoping of environment changes
- sequencing of commands
- configuration
- terminal access

These are addressed in the following sections.

7.3.1. Job Units

A job in PPM is a set of threads that are controlled as a unit, and which share a common job context in the environment. Threads will be part of the same job, by default, if one is a logical ancestor of the other. Threads can also be created as parts of existing or new jobs. The command interpreter normally uses this option to create new jobs for each command. Some commands have multiple components to be run as a single job. The command interpreter can implement this by first creating a job with no threads, and then adding threads to it for each component of the job. It is necessary, of course, to express to the command interpreter the fact that the components of the specification are to be executed as a single job. There are two syntactic options by which textual commands can be associated. The commands can be *connected*, as, for example, the components of a pipeline are connected by vertical bars in UNIX shells, or the commands can be *bracketed* by enclosing syntactical units. One could, for example, enclose a job specification by braces, such as parentheses, or by keywords, such as **jobBegin** and **jobEnd**.

The UNIX shell pipeline syntax has proven a useful way of expressing a certain class of multiprocess jobs. It can be retained in a command interpreter for a PPM shell, but does not serve as a model for expressing more general multiprocess jobs. Instead jobs are bracketed using square brackets (that is the characters '[' and ']'). A second set of brackets can be nested within the outer set. The outer set brackets a set of commands that are to be part of one job. All processes created in executing the bracketed commands are treated as a single job for job control, and share the same job context in the PPM Environment. The inner level of brackets allows the manipulation of the *<PROC>* context of the processes started by the commands they contain. In executing commands within these inner brackets, the command interpreter stores a set of bindings that are to be copied into the *<PROC>* context of any process created by these commands. It is possible that more than one process will be created by these commands, each with its own copy of the *<PROC>* context. For example, the command

[set A = 'x'; prog1; prog2; [set A = 'y'; prog3; set A = 'z'; prog4]]

will start processes to run *prog1*, *prog2*, *prog3*, and *prog4* as part of a single job. All will share

the binding of *<JOB>/A* to 'x', but *prog1* and *prog2* will have no binding for *<PROC>/A*, while *prog3* will have *<PROC>/A* bound to 'y', and *prog4* will have *<PROC>/A* bound to 'z'. This is discussed further in the next section.

7.3.2. Scoping of Changes to the Environment

The UNIX shells take advantage of the way the UNIX environment is inherited to provide subshells. The UNIX environment is copied from parent to child. The child process may exec some user program, or it can continue to operate as a command interpreter. Because this command interpreter is simply a separate process, it can change its environment (and also its working directory) without affecting the parent shell. When the child command interpreter exits, the parent continues as before. Commands within subshells therefore have localized side effects. For example, if a UNIX shell user types

```
cd /usr/a/smith
(a; cd ..; b)
ls
```

the commands **a; cd ..; b** are executed in a subshell. Therefore, the **ls** command will list the contents of the */usr/a/smith* directory, rather than those of its parent.

The PPM environment is quite different from the UNIX environment, in that it is shared by all the user's jobs. Moreover, it does not provide inheritance through copying. The fixed hierarchy within the name space does not allow command sessions to have "sub-command sessions" or jobs to have "subjobs." Therefore, the PPM environment structure will not allow subshells to be implemented as in the UNIX shells.

There are two alternatives to subshells that are allowed. First, the user can create a new command session. Second, the name space hierarchy can be used to provide appropriate restrictions on the effects of environment changes.

While it is possible to arrange for a command interpreter to create a new command session, this approach has some distinct drawbacks. When a new command session is created, it is given an independent environment context. Bindings for the command session of the creator are therefore not inherited. For example, the working directory for the new command session would be the user's home directory, rather than the current working directory of the creator's command session. It is, of course, possible, for the creator explicitly to initialize the new session, either with the values of particular bindings, or with indirections through the bindings of the parent command session. Another problem arises because each job is part of a single command session. The jobs run in the new command session are kept distinct by the bookkeeping. This means that, if one were to print out the jobs running in the parent session, any jobs running in the new session (which would include the command interpreter) would not be listed.

The second alternative is to use the structure of the PPM environment to achieve an appropriate set of semantics. PPM's structured environment provides several contexts from which a binding might be inherited. The command language must provide a means of setting bindings in these contexts so that they are shared among the proper processes. One means of specifying a context, of course, is to specify completely the binding that is to be made. The command "**set <CS>/A = 'x'**," for example, would bind the value 'x' to the name *A* in the command session context for the command interpreter. A second possibility is that the context for a binding is implicit. For example the statement "**set A = 'x'**" would ordinarily bind 'x' to *<CS>/A*, but if the statement were part of a job specification, the value would be bound to *<JOB>/A*.

The PPM environment does not provide a means of restoring the environment to a previous state. Furthermore, the environment is shared among all the users jobs, which may be running (and changing bindings) simultaneously. It is therefore not possible, after a job has completed, to remove all the bindings that the job might have made. There are two means of controlling the effects that a job will have on the environment. The first is to create a separate command session in which to run the new job. This is somewhat like the creation of subshells in the UNIX shells, with differences that are discussed in a later section. The second is to run the job within the existing session, but to provide information to the components of the job, so that they will choose to set bindings in the job context rather than the command session context.

7.3.3. Sequencing of Commands

Commands may be run synchronously. In this case, the command interpreter waits for each command to finish before running the next command. Because a command may comprise several threads, either created by the command interpreter or descendants of threads so created, we must define what is meant by the command "finishing." We shall say that each command will result in the creation of one thread that is the "leader." (This is analogous to the process group leader in the C shell.) When the leader exits, the command is considered finished. This choice allows commands to leave processes active after they have finished, running "in the background."

Commands may also be run asynchronously. When a command is run asynchronously the command interpreter does not wait for the termination of any thread before processing the next command. Asynchronous commands are not generally allowed to read from the command session terminal since that would allow races with other commands and with the command interpreter itself.

When a job is described by a series of commands, it may be necessary to separate the creation of a thread from the beginning of its execution. This is because configuration information for the job may follow in the instruction stream. A command can therefore be issued with delayed execution. The threads for the command are created at that point, but not set in execution. Execution begins when a later syntactic marker is reached. Separating the creation of the thread from its start of execution allows a job configuration to be described by a number of commands. These commands can include changes to the environment that should not affect the processes already created.

7.3.4. Configuration Commands

A multiprocess job must have some way of specifying its communication configuration. The UNIX shells offer the pipeline as a basic command format. The pipeline provides a simple syntax for expressing both the components of the command and their interconnection, but only for a restricted class of configurations. We do not try to extend this syntax to allow the expression of arbitrary interconnections, probably a hopeless task. Instead, we provide a set of configuration commands that can be bracketed with the other commands of a job specification. Such a format may be cumbersome to type interactively, but is easy to use in scripts.

Four configuration commands are provided: **input, output, port**, and **connect**. The command **input** creates an open descriptor from which the target process can read. The target process is the most recently created process whose execution has been delayed. The command takes two strings as arguments, the first is the *internal name* for the input, the second is the *external name*. The internal name is bound in the process context for the target process to information about the descriptor, including the descriptor index. The external name is used to refer to the descriptor when connecting it to a data source, and must be unique within the job. The

command **output** is similar, but creates a write-only descriptor. Once an input and an output have been created, they can be connected using the **connect** command. The **connect** command has one of the following three formats:

connect inputname outputname
connect filename > inputname (or **connect inputname < filename**)
connect outputname > filename (or **connect filename < outputname**)

The last two forms allow the connection of files to inputs and outputs. This may require the creation of auxiliary copier processes if the file is not accessible on the machine where the input or output has been created. The **port** command creates a message port for the target process. This command also requires an internal and an external name. The external name is bound to the address of this port. If the command is bracketed as part of a job, the binding is placed in the job context. If the command is not part of a job specification, the binding is placed in the command session context. It can be placed in other contexts either by copying it, or by placing an indirection in one of the more global contexts.

7.3.5. Terminal Access

When using a terminal or terminal emulation window, it is necessary to give programs access to that terminal. When a command is run synchronously, the command is allowed to read from and write to the terminal. When it is run asynchronously, it is allowed to write to, but not to read from the terminal. The result in this case is that only one process is allowed to read from the terminal, but several processes may write to it. This arbitration is accomplished by a series of commands sent to the terminal handler. The terminal handler directs input to the current foreground process and combines the output of the various processes to send it to the screen. Actually, several processes on a machine may share a single descriptor, so the outputs from these processes will be mixed before reaching the terminal handler, as in current UNIX shells. The PPM terminal handler will also mix the outputs of processes on different machines.

This form of sharing a terminal has clear drawbacks. If the user has access to a window system, it is often more appropriate to create additional terminal windows. Sometimes this choice will be built into the application program. A picture editor, for example, could be expected to set up its own windows. Some programs, for example a program to simply print the date, could be run either in the original window or in a new window. The choice of output destination should be encoded in the command; the command interpreter should ensure that the output is correctly displayed. In some systems, the responsibility for setting up a new window, when necessary, will fall, to the command interpreter. In others, the command interpreter will simply place bindings in the job's environment context, leaving window creation to library code in the application program. One way of encoding the choice for display is is to bracket commands to be run in new windows with a separate **newWindow** command.

7.4. Commands

This section discusses how some important commands are implemented for the shell described in this chapter. The examples include commands to

- print the working directory
- change the working directory
- change the command session's default machine for running jobs
- list the status of each thread belonging to a current job
- terminate one of the current jobs through menu selection

Example 1: print working directory

The current working directory is part of the state information for a command session kept by the environment service. A program to print the working directory looks up the value bound to the name *DIR*. Under UNIX 4.3BSD, each machine has a distinct file system, so the current working directory is machine specific. The working directory can therefore generally be set in the context *<CS.MACH>* (see Figure 7.4). Since a job can start processes on other machines, where *<CS.MACH>/DIR* has not been set, it is important that there always be a default directory on every machine. Until the working directory is explicitly changed, jobs should operate in the user's home directory. We can set the user's home directory when the Environment Service agent is created and initialized. The binding is read from an initialization file and placed in the context *<MACH>*. The initial value for *DIR* is also placed in this context. This could be either the same as the home directory or, as in the figure, an indirection through it. Until the working directory is changed, looking up *DIR* returns this default value. Once the "change directory" program has created a binding for *<CS.MACH>/DIR*, the default value is overridden.

If a machine works in a shared file system, the setting of the current working directory is done somewhat differently. In a shared file system, the current working directory can stay the same, even when the machine on which jobs are started changes. Therefore, instead of having a different working directory for a command session on each machine, there is a single consistent working directory for all the machines using a particular file system. This is accomplished through the addition of an indirection in the name space.

Figure 7.5 shows an example for a shared file system on the server **snow**, which has client machines **sleepy** and **sneezy**. By comparing Figure 7.5 with Figure 7.4, we can see that an extra level of indirection has been added to the context *<MACH>*. The initial value for *DIR* is set in *<MACH>* for both **sleepy** and **sneezy**, to be an indirection through *<CS>/SNOWDIR*. The initial value for *<CS>/SNOWDIR* is set in *<AllH>*, to be an indirection through *<MACH>/HOME*. The first time that the working directory is changed, the binding *<CS>/SNOWDIR* is set to indicate the new working directory. The same binding will therefore be seen by jobs on any of **snow**'s clients for the command session. Even if the session is

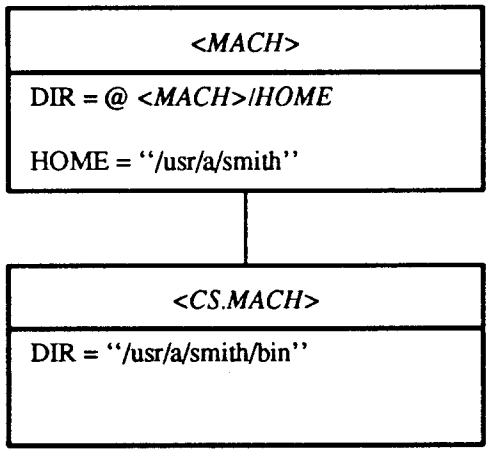


Figure 7.4 Bindings used for current directory

extended to new machines, jobs on the new machine will use the same working directory binding as jobs on the other machines. The bindings in *<MACH>* are set when the Environment Service agent on a machine is initialized. The binding in *<AllH>* is set (or reset to the same value) each time a new agent is initialized.

Example 2: change working directory

The "change working directory" command applies one of the strategies described above to create a new binding for the string *DIR*. The program runs on the current "target" machine, and changes the working directory that applies to processes created on that machine. Working directories that apply to machines using different file systems are not affected. Rather than writing a single program that handles changing directories in both shared and unshared file systems, we can can write two different programs and place the different binaries in the two sorts of file systems. The program that runs in response to the change directory command will then apply the correct strategy.

Example 3: change target machine

A command session has a single binding for a "target" machine. This is the machine on which processes are to be instantiated (unless the Creation Service is explicitly instructed otherwise). The name of the current target machine is bound to the name *<CS>/TARGET*. (If no target binding is found by the Creation Service, the local machine is used. Thus, it is unnecessary to initialize *TARGET*.) The change target program replaces the value of this binding with the name of the new target machine. Before this is done, however, the program attempts to contact the Creation Service on the new machine. If the user's LPM agents already exist on the new machine, this is a simple matter. If not, the agents must be created. The user may be prompted for a password at this time. It is possible that the user will be denied access to the new machine, in which case the target machine binding is left unchanged. Otherwise, the new binding is set.

Example 4: find thread status

Finding the status of threads is done through requests to the PPM Bookkeeping Service. First a list of jobs is obtained, and then lists of processes and threads for each job are obtained. The first call returns a time stamp that is used on the subsequent calls. Thus, the information obtained is for a particular moment in time on each machine, and, assuming a reasonable degree of clock synchronization, these moments should be close in real time. These calls return information about all threads registered with the Bookkeeping Service. It does not return

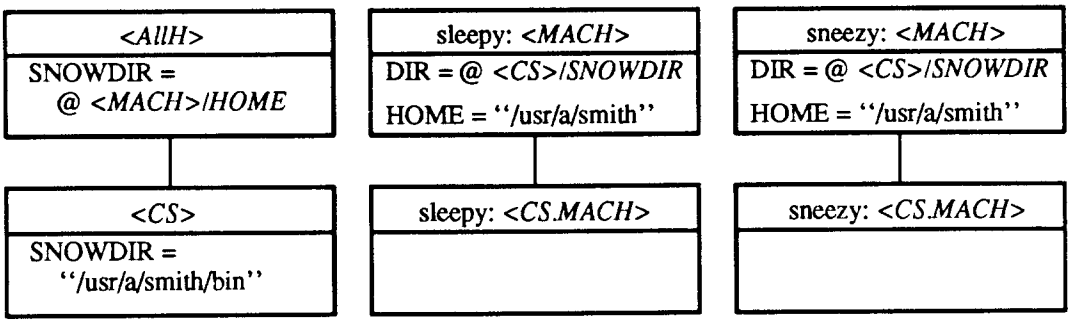


Figure 7.5 Bindings used for current directory

information about threads in processes belonging to other users. It may be possible to obtain this information through other calls, such as the current UNIX process status program, which searches the system process table on a particular local machine. This may not reveal information, such as parentage, that is available through the Bookkeeping Service for a user's own threads. Since this detailed information is helpful primarily for debugging, it is not needed for threads or processes belonging to other users.

Example 5: create menu to terminate jobs

In a single process UNIX shell, creating a new form of command interface, such as a menu for selecting a job to terminate, is difficult. First, there are no external interfaces that allow other programs to use shell functions, save by issuing shell commands. Information obtained through these commands is often in a difficult format to parse. Second, important information may be held by a particular shell, and it may be difficult or impossible to submit a command to that particular shell. With the client-server approach taken by PPM, it is much easier to add capabilities to the user interface. A program to create a menu of current jobs simply has to contact the PM Master to find the address of the Bookkeeping Service, obtain a list of jobs from the Bookkeeping Service, and display it on the screen. If the user chooses to terminate a job from this list, the program can issue the command to the creation service and can itself exit.

7.5. A Simple Distributed Program

A simple distributed program is shown in Figures 7.6 and 7.7. This program calculates the average time that it takes a server to process a request from a (possibly) remote client. The user invoking the program specifies a number of repetitions and the name of the machine on which the client is to run. After initializing the IPC library, and finding the addresses of the PPM Services, the server process (Figure 7.6) prepares to start the client by creating and registering a message port. The port is registered in the *<JOB>* context. This will allow the child process to look it up, but allows the program to be run several times simultaneously as different jobs without confusion. The server registers an exit action that will apply to all members of the job. It specifies that, if any component should exit with any error status, the entire job will be terminated. The server then changes the *target* machine for the current job to that specified as an argument. This binding is evaluated by the PPM Creation Service when *creProcess* is called, and the client process is created on that machine. The number of repetitions to be made is passed to the child as an argument. It could also be passed through the PPM Environment.

The client process (Figure 7.7), after initialization, looks up the port to which it will send requests. If no server port is found, there must be a bug in the program, for example the server and client might be using different names for the port. If the client cannot find the server port, it requests that the entire job be terminated. Assuming that no such bug has occurred, the client can formulate a request packet, and find the time taken to make the specified number of requests. When done, the client sends an exit request to the server and exits. The server, upon receiving the exit request, terminates itself through a call to *creStateChange()*.

This application illustrates several points about PPM. Many of the difficult tasks of a distributed application are hidden. The address of the server is made available to the client, without ambiguity, without requiring a unique name. A process is started on a remote machine without having to formulate a command in some command language. The entire job, spanning machine boundaries, can be terminated by a single call. The job is terminated if either client or server fails, without constant checking for this possibility. PPM can provide a high degree of transparency. The program can be run both as a distributed program, on two machines, or as a non-distributed program on a single machine. Yet, the programmer generally does not have to be

```
main(argc, argv)
    int argc;
    char *argv[];
    {
    char *argV[3];
    struct MSG *currentReqP;
    char *servList[2];
    struct hostent *hP;
    int req;
    int rc;

    struct envTable *envTableP = NULL;

    ipcInit();              /* Initialize IPC library */
    ppmClient();            /* Find addresses of PPM Services */

    ipcCreatePort("myPort", &rc);
    ipcMakePortKnown("myPort", "<JOB>/myPort", &rc);

    /* Register exit action, to ensure that the failure
     * of one component will result in the failure of the other */
    envSet("<JOB>/exit.action",
      "if (status != gNoError) kill JOB", 33, &rc);

    /* Start child on machine given as argument */
    envSet("<JOB>/target", argv[2], strlen(argv[2])+1, &rc);

    argV[0] = "testClient";
    argV[1] = argv[1];
    argV[2] = 0;

    creProcess(argV, NULL, &id, &rc);
    creStateChange(RUNNING, id, &rc);

    /* Process requests until told to exit */
    for(;;)
      {
        servList[0] = "myPort"; servList[1] = 0;
        ipcSelect(servList, &currentReqP, &rc);
        req = currentReqP->req;
        if (req == TEST)
          /* Perform Service */ ;
        ipcReply(currentReqP, &rc);
        if (req == EXIT)
          creStateChange(TERMINATED, SELF, &rc);
      }
    }
```

Figure 7.6 The server process for a simple distributed program.

```
main(argc, argv)
    int argc;
    char *argv[];
    {
    struct MSG req;
    struct timeval tv1, tv2, time;
    int i, n;
    int rc;

    n = atoi(argv[1]);

    ipcInit();          /* Initialize IPC library */
    ppmClient(); /* Find addresses of PPM Services */

    ipcFindPort("myPort", "myPort", &rc);
    if (rc != gRCNoError)
      {
      /* If we cannot find the server, terminate the job */
      creStateChange(TERMINATED, JOB, &rc);
      /* Not reached */
      }

    req.req = TEST;   /* Create request for TEST service */
    req.rc = 0;
    req.len = 0;

    gettimeofday(&tv1, 0); /* Print average time taken for n requests */
    for (i = 0; i < n; i++)
        ipcCall("myPort", &req, &rc);
    gettimeofday(&tv2, 0);
    timeDiff(tv1, tv2, &time);
    timeDivide(time, n, &tv1);
    printf("Avg. Service Time: %d.%06d sec.0, tv1.tv_sec, tv1.tv_usec);

    req.req = EXIT;          /* Tell server to exit */
    req.rc = 0;
    req.len = 0;
    ipcCall("myPort", &req, &rc);
    }
```

Figure 7.7 The client process for a simple distributed program.

concerned with the issues this raises. The exit action specified means that the failure of one machine will end the job without requiring elaborate checking within the program. The IPC library hides machine addresses altogether.

CHAPTER 8

A PPM PROTOTYPE FOR UNIX 4.3BSD

8.1. Prototyping the PPM Services

Several versions of the PPM were implemented as ideas about its function and organization matured. The design described in this dissertation differs markedly from the design described in [CSC86], for example. In this chapter we shall describe how particular operations for the PPM Services are implemented in their most recent version. Particular attention is paid to the remote procedure calls and other time consuming operations, and performance measurements are given.

There were four primary stages in PPM's development. In the first, the concern was in providing facilities for remote execution and distributed bookkeeping that would be failure resistant and scalable. In this design, services similar to those of the present PPM Creation Service and PPM Bookkeeping Service were offered by a single agent. This was the design described in [CSC86]. A second implementation used a coroutine package to systematize the agent's activities. Since an agent would accept new requests while waiting for responses from remote agents, it was important to keep the overlapping activities logically separate. This version was much simpler than the original to modify and to debug. The third stage came with the separation of the services between distinct agents. This made the design much simpler, and the code easier to work with. This coincided with the development of the PPM Environment Service, and a change in the communication paradigm used. The earlier versions had attempted to use byte stream communication channels between agents. This led to a number of problems that disappeared with the use of connectionless communication protocols. The change in communication paradigm necessitated the development of the PPM Session Maintenance Service. With the stabilization of the PPM Service definitions, development in the fourth stage could turn to other aspects of PPM, such as the structure of shells using PPM, authentication, and the nature of the overall structure that ought to be imposed upon a user's work.

8.2. Performance

There are several questions that we can ask about the performance of the PPM in its prototype implementation. Important questions include the following:

How long does it take to install a new binding in the environment?
How long does it take to look up a binding in the environment?
How long does it take to start a single process job?
How long does it take to obtain a list of running jobs?

The answers to these questions are dependent upon the implementation, upon the speed of the machines where the PPM session is running, and upon the system load on those machines.

Because the prototype uses different LPM agents to provide the various PPM Services, the fundamental determinants of the speed of a PPM operation are the number of messages exchanged and the speed of these exchanges. Most messages in PPM are sent as part of remote procedure calls (RPCs), either within a single machine or across a network. RPC times can

System	Hardware	Local (ms)	Remote (ms)	Reference
V	Sun-2/120	0.93	3.32	[Alm86]
V*	Sun-3/75	0.48	2.54	[Che88]
4.2BSD UNIX	VAX-11/750		38.2	[SoM86]
4.3BSD UNIX	MicroVAX	7.2	12.4	[CMC88]
Clouds	VAX-11/750	10.	40.	[DJA88]
Argus	MicroVAX-II		17.5	[LCJ87]
Sprite	Sun-3/75	2.05	4.26	[WeO88]
Sprite*	Sun-3/75		2.8	[OCD88]
QuickSilver*	IBM PC-RT 25	0.66	9.0	[HMS88]
Sun RPC	Sun 3/50		11.01	[Ros88]

Table 8.1 Times for RPC and RPC-like communication in current systems. (Times for systems marked by asterisks are for Request-Response IPC.)

vary considerably between implementations, but can be made quite fast on today's machines and networks. Table 8.1 shows the time taken for RPCs and RPC-like communication for various machines using various systems. The precise semantics for the message exchanges vary between systems, as does the hardware used. No doubt, some of these numbers are somewhat optimistic, because they were measured on lightly loaded machines. Nonetheless, the important point to note is simply that recent systems claim very fast (< 1 ms) message exchanges between processes on the same machine and fast (<5 ms) exchanges between processes on machines connected by local area networks. It is therefore reasonable to assume that systems will in the near future will be able to support rapid RPC in functioning systems.

The PPM prototype uses an RPC package built on top of UDP datagrams as implemented in UNIX 4.3BSD. Figure 8.1 shows RPC times for null RPCs. The graphs show RPC time as a function of system load (as reported by the UNIX *la* command) rather than simply a best case, or an average case time. On the slower VAX 785, local RPC times were somewhat unpredictable, although 15-20 ms is common when the load average is below 4, while 20-35 ms is common for higher loads. On the faster VAX 8600, local RPC times fell in the range from 4 to 15 ms. RPC times between the machines were most frequently in the range from 12 to 25 ms. The source of this variability is the rescheduling delay for the RPC recipient. This rescheduling delay is more variable for slower machines, in which processes are more likely to exceed their scheduling time slice without giving up the processor on some system call. Because of this variability, most of the discussion of performance that follows will refer to the number of local and remote RPCs required, rather than to the time taken.

The measurements in Figure 8.1 show that for slower machines, local RPCs can be somewhat expensive if the system load is high. Fortunately, faster machines, and faster RPC implementations are becoming available. Nonetheless, PPM could be made faster by combining PPM services in a single LPM agent. Local RPCs, and their attendant delays, would then be eliminated, since the services would be able to communicate through shared memory, rather than through messages. This sort of optimization is not important with respect to PPM functionality. For relatively fast machines these sorts of optimization may not be necessary, since PPM provides interactive services to humans, for whom delays of fractions of a second are unnoticeable.

In the following sections we shall discuss how some important operations are accomplished by the PPM Services. The speed with which these operations can be executed depends

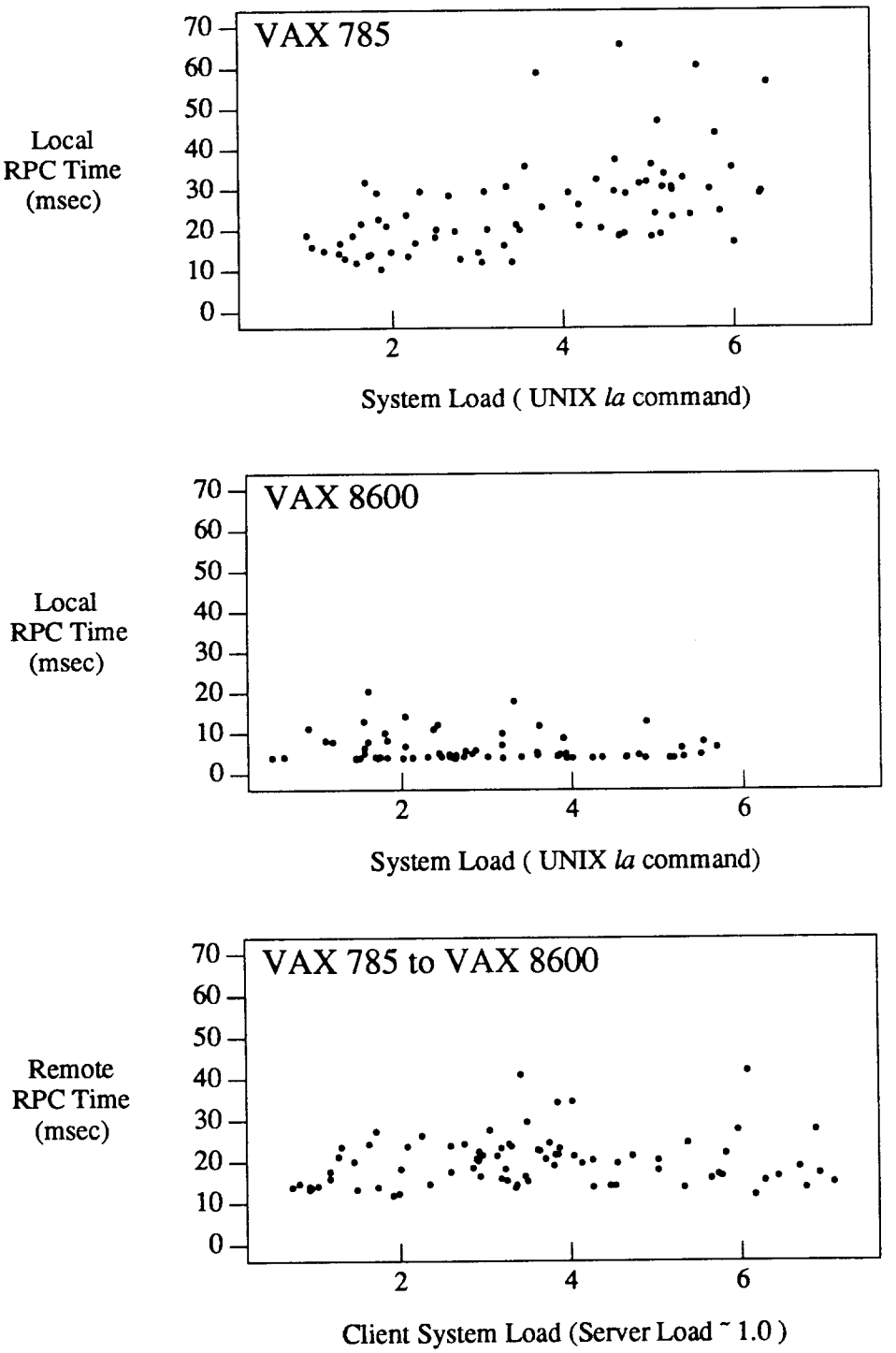


Figure 8.1 Scatterplots of RPC time *vs.* system load for prototype.

upon sizes of tables, lengths of lists, and other factors, but the contribution of these factors to the time taken is dwarfed by the time taken for RPCs. We shall, therefore, concentrate on describing operations in terms of the amount of interprocess communication that they require.

8.3. The PPM Environment Service

The PPM Environment Service is provided for each user by a separate LPM agent on each machine. These machines keep consistent copies of almost all bindings. The exceptions are a very small number that are used by the PPM Creation Service, and the PPM Bookkeeping Service, but which can be read through the Environment Service interface. For example, the name of a process's home machine can be obtained from the Bookkeeping Service, but it is simpler to get it from the Environment Service. The Environment Service's main function is to allow the user to set and evaluate bindings. The time taken for these functions, especially the latter is critical in the functioning of PPM.

Setting a binding in the environment.

To set a binding in the environment, the user calls a stub routine, which makes a local RPC request to the LPM agent providing the PPM Environment Service. The agent must first decide what the fully qualified name of the binding will be, if this is not specified. To do this, it is necessary to obtain from the PPM Bookkeeping Service the client's evaluation path, along with its job and command session identifiers. This requires a local RPC. Once the name is established, the binding must be locked for every participating LPM agent. This can be done by separate RPCs to each of the participants, but this incurs unnecessary delays. Instead, the calls can be overlapped, with all the requests preceding the replies, as a replicated RPC. The lock request can also hold the new value for the binding. Assuming that the request to lock all copies of the binding is successful, a commit request is sent to all sites as a replicated RPC. This request commits the new binding, and releases the lock.

The time consuming operations required to set a binding are thus

local RPC to request service (initial request)
[local RPC from environment to bookkeeper (to resolve simple name)]
remote replicated RPC to other environment on other sites (lock copies)
remote replicated RPC to other environment on other sites (commit update)

The operation in brackets is not required for fully qualified names.

Figure 8.2 shows the time taken to set a binding in the environment from a VAX 785, when the environment session has one, two and three participants. The additional machines are lightly loaded VAXes. The VAX 785 is the coordinator of the update, and, because the other machines are lightly loaded, it is always the bottleneck. It can be seen, first, that the time taken can be quite variable due to the delays in rescheduling processes after they send messages. Second, the expense involved in the remote two-phase commit protocol is considerable, and rises steeply with load on the controller, at least for a machine in the VAX 785 class. However, the remote operations are overlapped, and, therefore, additional sites can be added to the Environment session without suffering additional rescheduling delays.

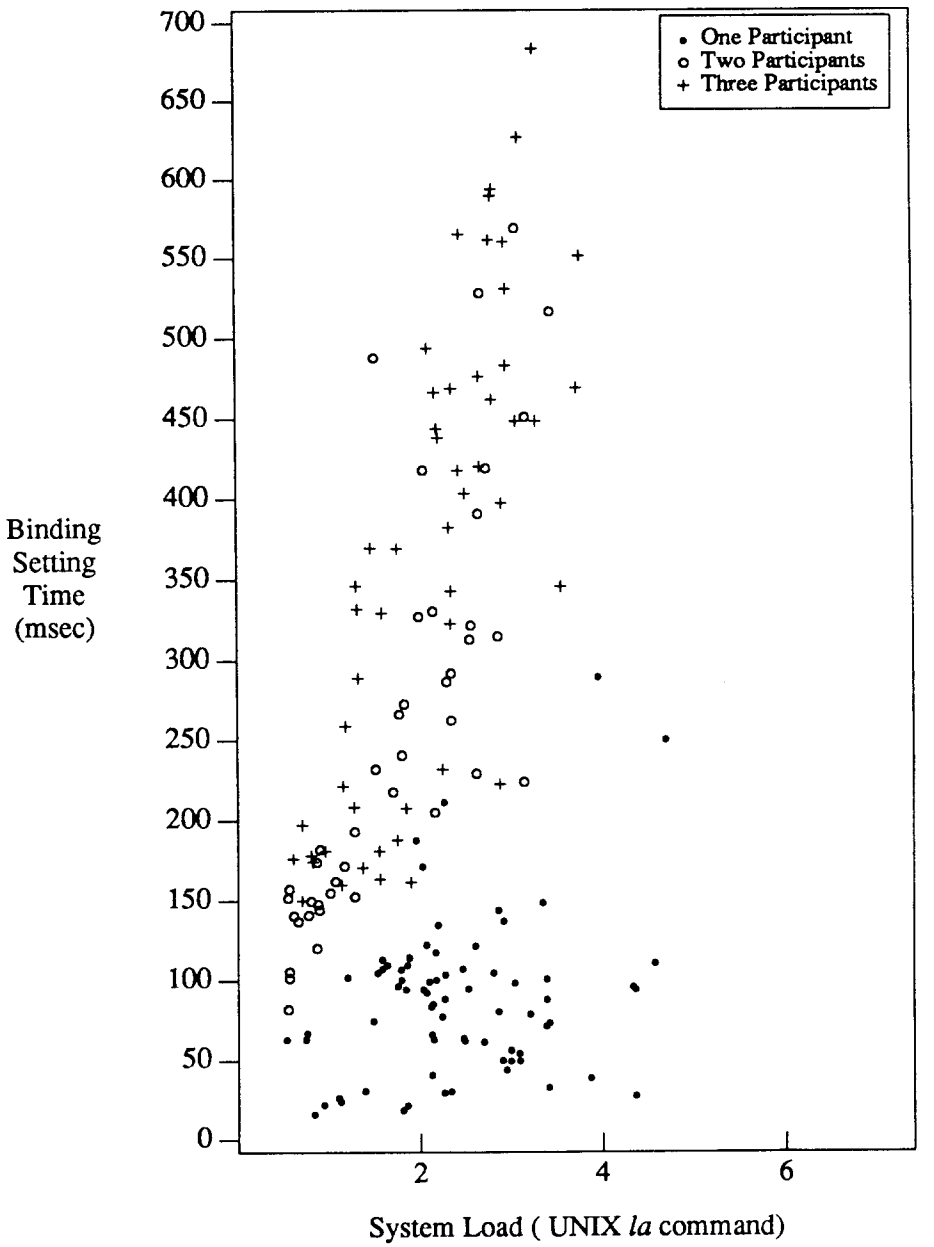


Figure 8.2 Scatterplots of binding setting time *vs.* system load for prototype.

Evaluating a binding in the environment.

To evaluate a binding, the stub routine makes a local RPC request. If the name is not fully qualified, the PPM Bookkeeping Service must be contacted to obtain the client's evaluation path, job identifier, and command session identifier. Thus, the following operations are required:

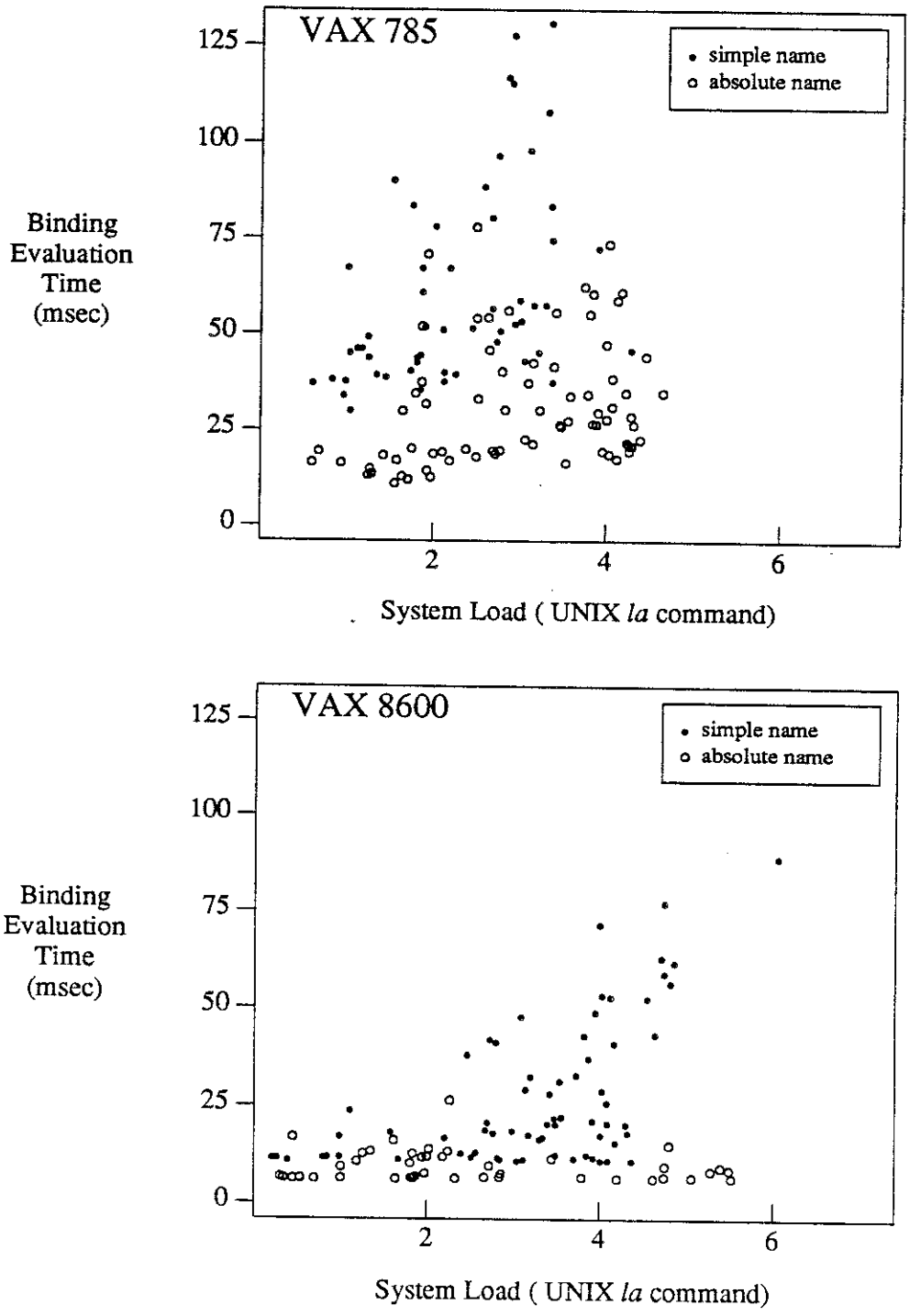


Figure 8.3 Scatterplots of binding evaluation time *vs.* system load for prototype.

local RPC to request service

[local RPC from environment to bookkeeper]

The time taken to evaluating a binding at different system loads is shown in Figure 8.3. Since simple names require an additional RPC, their evaluation is more subject to the vagaries of scheduling than that of fully qualified (absolute) names. Clearly, avoiding the expense of the additional RPC by combining the LPM agents for the Bookkeeping and Environment Services in a single address space would make evaluations faster. The value that would be derived depends upon how frequently evaluations are performed, and event the times taken for evaluating absolute names are probably too high to allow polling. Caching specific items, however, can reduce the cost of repeated evaluations to that of a call to a short local procedure. Thus, more experience will be necessary before it can be judged how critical the time taken to evaluate a simple name is.

Providing clients with new values for cached bindings

By calling *envCache()*, a client can request that it be sent a message each time a particular binding takes on a new value. It is simple for an LPM agent to assist a client caching bindings with absolute names. The names of cached values may be stored and the names of updated bindings compared with these names. If the names match, the agent sends a message as soon as the update commits. The situation is more complex if the cached name is relative. This is because the value associated with a relative name may derive from any of a number of absolute names. Therefore, it is not sufficient to look for a correspondence between the absolute name being updated and the absolute name for the last value cached. This will not detect all updates. Instead, if the last component of a cached name matches the last component of the update, it is necessary to reevaluate the binding on behalf of the client doing the caching. If the update supersedes the cached value, a message is sent to the caching client. Note that locking by the agent prevents update messages to caching clients from being sent out of order.

8.4. The PPM Bookkeeping Service

The PPM Bookkeeping Service is offered to the owner by an LPM agent at each site. The bookkeeping on each machine can be done in isolation, except when answering a client query or when a client on one machine uses the PPM Creation Service to create a logical child on another machine. In this case, the bookkeepers on both machines record the logical parent-child relationship. To obtain information about the global state of the PPM Session, it is necessary to contact the local agent, which then contacts the agent of the Bookkeeping Service on each of the other sites. Representative examples of the operations offered by this service are obtaining a list of the user's jobs, and finding the processes that a particular job comprises.

Keeping track of a user's work

To provide the user with accurate information about his jobs, the Bookkeeping Service needs, somehow, to obtain accurate information about the creation of new jobs and processes, the termination of processes, and the like. The prototype implementation relies upon the user calling library routines when creating new processes. To monitor programs that make free use of system calls and facilities outside the PPM Creation Service, more powerful monitoring capabilities are needed. The UNIX kernel can be modified to provide reports from the kernel to user processes in the form of IPC messages. This was done by the author for a version of UNIX 4.2BSD [MMS86]. A system call was added to the system that allowed a monitoring process to be specified for each process. Messages were sent to the monitoring process whenever a significant event occurred. Significant events included process forking, *exec()* calls, and process termination. Optionally, messages reporting communication channel establishment, message transmission and reception. A monitoring capability of this sort makes possible a wide variety

of tools for distributed programming, such as debuggers and program monitors (see, for example, [GrZ83] [GJK84] and [MMS86]).

Obtaining a list of the user's jobs

To obtain a list of the jobs that the user has running, it is necessary to contact the agents for the PPM Bookkeeping Service at every site in the PPM session. The request is sent by the application to the local agent. This agent obtains a list of participating sites from the PPM Session Maintenance Service, and makes a remote RPC to the agents at every site. These requests are timestamped and the agents return a list of the running jobs at that time. Thus, the operations are

local RPC to request service
local RPC to Session Maintenance Service to obtain list of sites
remote RPC to each of the sites to obtain a list of jobs

It is possible that clock skew will render this snapshot inaccurate. Other algorithms in the literature [SpK86] [ChL85] deal with the problems of guaranteeing a consistent snapshot, but at a greater cost in terms of messages. These guarantees are probably unnecessary for interactive users and for most applications, and have therefore not been implemented as part of the prototype.

Obtaining a list of the processes in a job

To find out the composition of a job, whether obtaining a list of its component processes, their parent-child relations, or their communication structure, it is necessary to contact all the machines where the job has components. This is normally done by contacting the site where the job originated (this is encoded in the job identifier), which returns its local job information, and a list of any other component sites it knows about. These are contacted in turn, and they return their local information and lists of any component sites they know of. When all the sites have been contacted, the originating site will be able to construct a snapshot of the job to return to the user. These steps are

local RPC to request service
remote RPC to each of the sites where processes of the job have run

If one or more of the component sites fails, the agents of the Bookkeeping Service on the surviving sites contact one another to construct links among the records of the surviving sites' processes. If the originating site survives, all the information can be found by the method just described. If the originating site fails, it is necessary to contact all the sites to find some component of the job, if any still exist.

8.5. The PPM Creation Service

The Creation Service is provided by LPM agents created by the PM Master. These agents create processes, and issue process control commands to the system. Since UNIX restricts the permission to terminate processes to processes with the same identity, the PPM Creation Service, unlike the other PPM Services, may require more than one LPM agent at each site. In this case, the service may be offered by a master LPM agent, with the assistance of some number of slaves, with different identities. The PPM Creation Service is charged with creating and configuring jobs, as well as suspending, resuming, and terminating jobs and their components. We shall therefore discuss the implementation of these operations and the time that they require.

Creating a new single-process job

To create a new job, the local master agent for the PPM Creation Service is contacted. This is true even when the job is to run on a different machine since the local agents are involved in the bookkeeping. The local agent ordinarily obtains several pieces of information from the PPM Environment Service. It is possible for the client to supply this information explicitly, which decreases the time required. Most importantly, the Creation Service must decide on what machine to instantiate the job. If the job is to reside on a different machine, the request is forwarded and the agent waits for completion, whereupon it registers the creation with the local agent for the PPM Bookkeeping Service. If the creation is local, it must be decided whether the new process is to have the same identity as the master LPM agent, or whether it is to have some other. The latter possibility may require the creation of a new slave agent with the appropriate identity. The request is forwarded to a slave agent if the master agent's identity is not to be used. The master or slave proceed identically in any event. (The current UNIX implementation does not incorporate the ability to change identities. Cooperation required between a master and a slave agent is quite similar to that between a local and a remote agent, which has been implemented.)

The LPM agent at the target machine, using the correct identity, can now proceed to create the job's process locally. The new process's working directory must be found from the Envronment Service, along with the standard input source and standard output and error destinations. If the file name for the job's process is a relative name, the file system must be checked to find the correct file to execute. The parent process's execution path must be used to generate a series of file names to try. The execution path is one of several bindings held internally by the Creation Service, rather than by the Environment Service, since they are accessed almost exclusively by the Creation Service. (This is probably a case of premature optimization, since the decision to hold the path in this way was not made on the basis of any measurements of performance.) Therefore, no RPC is needed to find the path. Once the executable file is found, the process can be created. In UNIX this requires calls to *fork()* and *exec()*. The process and job are then registered with the Bookkeeping Service. The main steps of this process are the following:

local RPC to request service
[local RPC to environment to find the appropriate identity]
[local RPC to environment to find target machine]
[remote RPC to different execution site]
[local RPC to environment to find working directory]
[local RPC to environment to find stdin, stdout, and stderr]
[file system search to find executable file]
process creation
local RPC to bookkeeper to register job and process
[register the explicit parameters with the environment]

Figure 8.4 shows the time taken to create a new single-process job locally on a VAX 785 and a VAX 8600 at a variety of system loads. Clearly, the slower VAX 785 can suffer significant delays due to the rescheduling delay for each remote procedure call. The faster VAX 8600 requires a more uniform amount of time, without regard to system load. Much of the expense of the job creation is due to the transfer of information between LPM agents for the various PPM Services. Most of this expense could be avoided if the LPM agents shared their address spaces.

On the faster machine, creating a new job requires 100-200 msec. On the slower machine, it often requires more than a second. This is adequate for interactive command interpreters, but rather slow when judged by machine, rather than human, standards. Fortunately, much of the delay is suffered because of the use of the environment as a state repository. This is a convenience for human users, which may not be needed by programs that create new processes. The time taken for each step in the creation of a local job on the VAX 8600 for a typical run is shown in Figure 8.5. The topmost line shows the time taken when the client does not explicitly provide information about the execution site, the working directory, the standard I/O connections, and the fully qualified name. The middle line shows the improvement if the fully qualified name is given, while the bottom line shows the time taken if all the information is explicitly provided. (As noted above, the implementation does not allow the choice of a new identity.) Programs can often provide the additional information explicitly, while for human users it is an annoyance.

Creating an input, output, or port

To create an input for an existing process that has not yet begun executing, one sends a message to the local agent for the Creation Service. The message may have to be forwarded to some other site, which is encoded in the process's identifier. At the process's site, a socket is

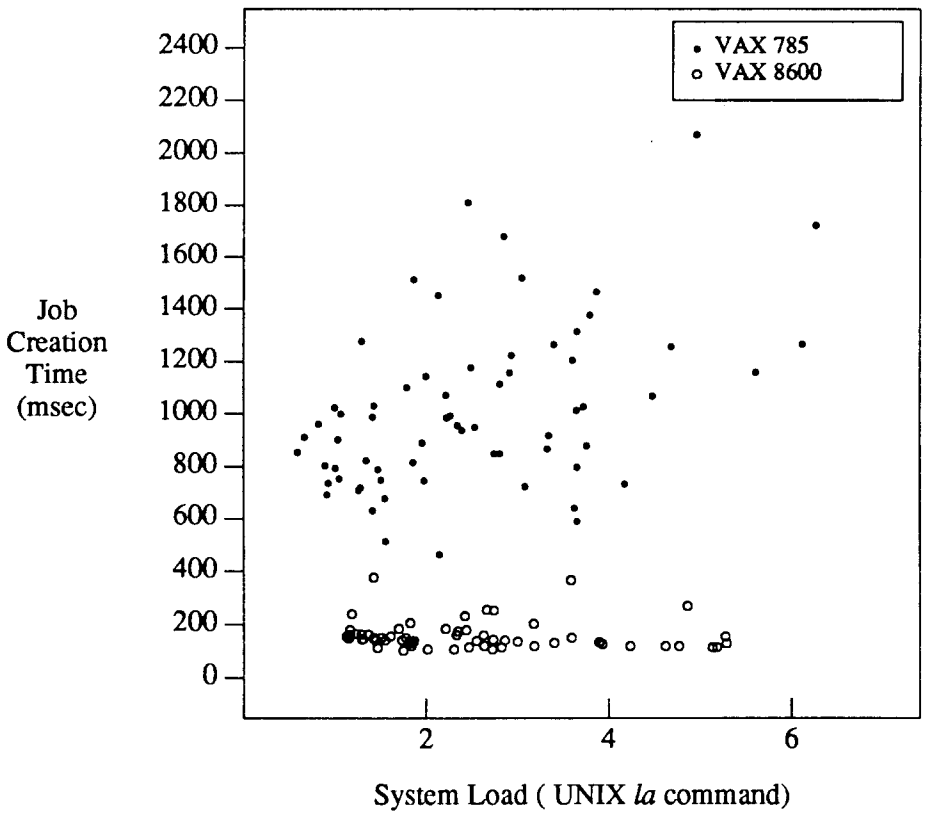


Figure 8.4 Scatterplots of job creation time *vs.* system load for prototype.

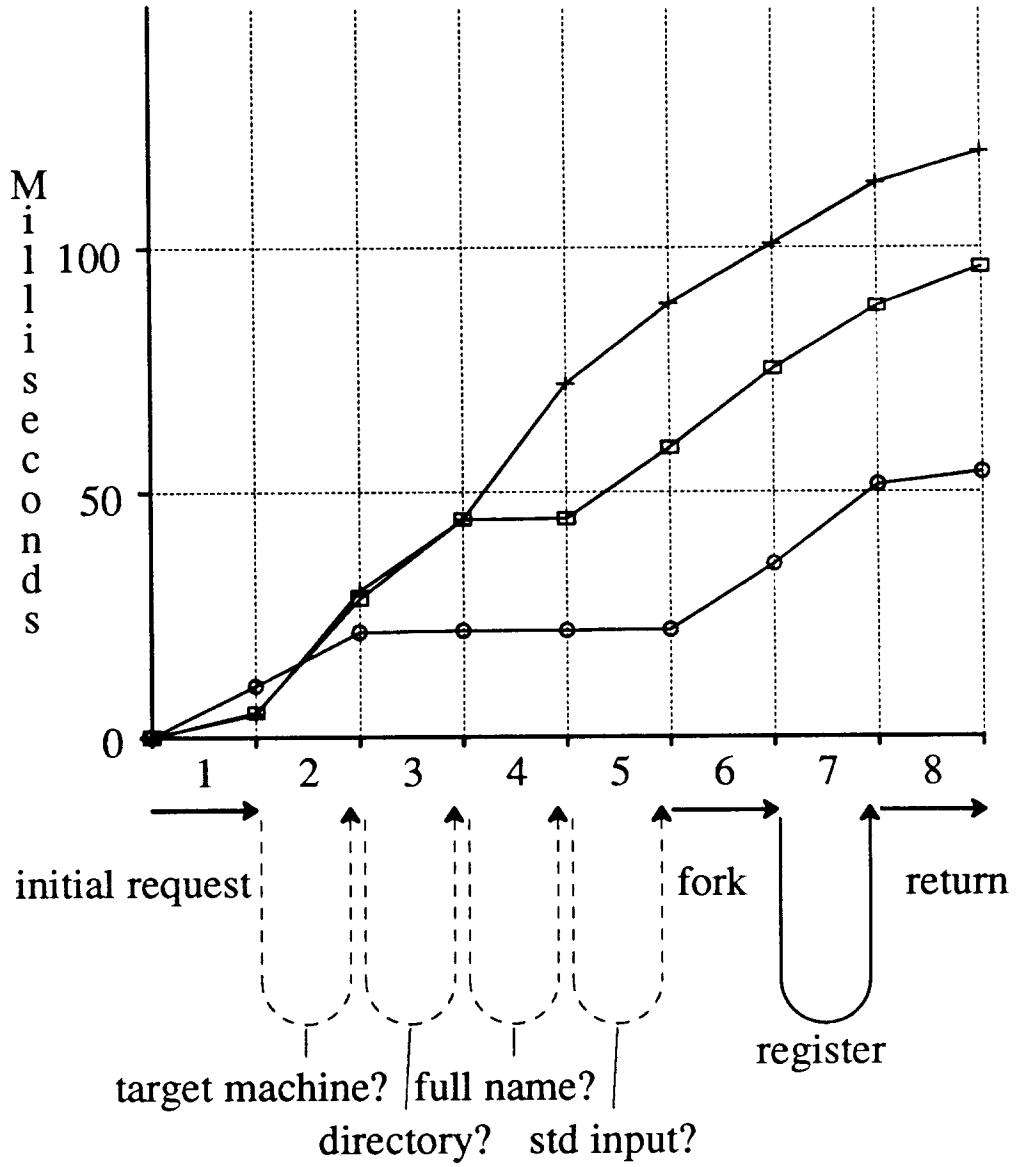


Figure 8.5 Cumulative process creation time for prototype implementation.

created by the process, which has been forked, but which is still executing the code of its parent, the Creation Service agent. Once the socket is created, its address and other information is placed in a binding in the environment. The steps are thus

local RPC to request service
[remote RPC to process's execution site]
request forwarded locally to forked child process
request to environment to set binding

Connecting an output to an input

To connect an existing output to an existing input, the request is sent to the local agent, whence it may be relayed to the agent on the output's site if this is remote. On the remote site, the request is forwarded to the child process owning the output, which still runs the code of a Creation Service agent. The child process looks up the input's address in the environment and a connection between the output and input is established.

local RPC to request service
[remote RPC to process's execution site]
request forwarded locally to forked child process
local RPC to environment to find input
connect output socket to the input

Both creating communication endpoints and connecting them is slowed down by the fact that UNIX does not allow one process to configure the I/O descriptors of another. Thus, it is necessary to forward requests to the forked child process and suffer the attendant rescheduling delay. This same limitation also prevents the configuration of I/O endpoints of processes that have already begun executing. This would be useful, primarily, for debugging computations.

Stopping a job

To stop a job, a request is sent to the local agent. If the job is purely local, the job is stopped locally with UNIX signals. If the job is not entirely local, a local request is made to the Session Maintenance Service for a list of the session constituents. A replicated RPC is made to all sites to stop the components of the job. While it would be possible to discover the sites where the job is running and send messages only to those sites, this would be somewhat slower than simply sending the message everywhere.

local RPC to request service
[local RPC to Session Maintenance Service]
[replicated RPC to Creation agents on all sites]

A job is terminated at a particular time stored with the PPM Bookkeeping Service. Once a job has been terminated and a timeout period set in the PPM Environment has passed, information stored about the process by the Bookkeeping and Environment Services can be disposed of. These method of stopping a job is rather brutal, since it gives the job's processes no opportunity to clean up resources.

8.6. The PM Master Service

Under Berkeley UNIX many network services are made available by long-lived system-owned processes called *daemons*. Not all system daemons are created when the system is started. Instead, a single server, called the *inet daemon*, is started, which is capable of starting the other daemons upon demand. The inet daemon receives all the messages sent to the well-known addresses of system services for which there exists no daemons. After the daemons' creation, they receive the messages sent to these addresses directly. Under Berkeley UNIX, therefore, the PM Master can be given a well-known address and added to the set of system daemons created by the inet daemon.

8.7. The PPM Session Maintenance Service

The PPM Session Maintenance Service provides its clients with consistent lists of the session membership and notification of membership changes. The service can be provided by a set of LPM agents, one per machine in an owner's session, running a protocol to enforce consistency between their membership lists.

Updating session membership information

The Session Maintenance protocol follows that used by Birman and Joseph to implement reliable multicast communication [BiJ87]. This protocol allows the participants in a PPM session to maintain consistent lists of participants. These participation lists are used within PPM services to update information through multicasts and to force consistency checking to take place when two sessions merge. The lists are also used by client programs, for example when inquiries must be sent to all the bookkeepers for the session.

In the Session Maintenance protocol a site (the *coordinator*) that wishes to effect a change announces the new constitution of the session to all sites that are to be in the new group. (We distinguish here between the *session*, which has an unchanging identifier whatever changes occur in its membership, and the *group*, which is an instance of a session's membership tagged with a sequence number.) If all the sites consent, that is they positively acknowledge the change, the change is committed. If any of the sites disagrees upon the constitution of the group, it negatively acknowledges the update, sending back to the coordinator a list of differences with the proposed group. In the event of such a negative acknowledgement, the coordinator reformulates the group membership and tries again. Sites must store all uncommitted groups until one commits. Any group with a sequence number lower than that of the committed group may then be thrown away.

The membership of the session changes when a new site is added, when a site withdraws, and when a site fails. Sites are added at the specific request of some LPM agent who seeks to communicate with an LPM agent on the new site. The requester is the coordinator in this case. When an LPM agent wishes to withdraw from the group a message is sent to some other agent, who then acts as coordinator. In the event of a failure during site agreement, including the failure of the coordinator, a new coordinator may begin an attempt to formulate a new group, based upon the most recent group that coordinator has received.

Each new group formulated is assigned a higher sequence number than the last committed group known to the coordinator or any subsequently received formulation. This assignment is not necessarily unique, but if any recipient receives two different formulations with the same sequence number, the second will be negatively acknowledged. Assuming that failures and additions are intermittent, a group will eventually be formulated that all sites agree upon and will be committed. It is possible that more than one coordinator will be proposing new groups and that, once brought to convergence, several coordinators will send commit messages for the convergent group. When a group is committed, all groups with lower sequence numbers are disposed of. All uncommitted groups with greater sequence numbers are retained.

Detecting host failure

One reason that a PPM Session's membership may change is that host machines in the session may fail. The Session Maintenance Service must ensure that such failures are detected. In the prototype host failures are detected by timeouts. The LPM agents for the Session Maintenance Service form a ring, ordered by network address. Each periodically checks to see it the next agent in the ring will respond to a message. The LPM agent initiates a reformulation of the group through the site agreement protocol. Once the new group has been formulated, messages from the old site will not be accepted, except those which seek to reformulate the group once again. Thus, in the case of falsely detected failure, two sessions will result, which must then merge. During the partition two things may occur that make this merger difficult. Changes may have occurred in the environment, in which case any conflicts must be resolved. Also, actions may have been taken in response to a perceived process failure when the process has not in fact exited. If two sessions both have records of jobs, and the membership of the jobs

are in dispute, that is a process on one site is believed to be dead, when it is in fact alive, the disputed processes are killed.

8.8. Using PPM

The prototype implementation of PPM can be used to run programs on a network of VAXs running Berkeley UNIX 4.3BSD. The current RPC stubs do not use a machine-independent format for data, hence a PPM Session cannot extend to a heterogeneous collection of machines. Previous versions built a format conversion into the RPC stubs, enabling cooperation between heterogeneous architectures, but this made the stubs harder to modify, and was left out of later versions. Future versions should use a standard format such as Sun Microsystems' XDR (see [DeS86]).

The prototype is fast enough to use interactively on a lightly-to-moderately loaded VAX 785. As the machine becomes more loaded, however, the delay due to local RPCs becomes quite long, and several seconds may elapse between typing a simple command and receiving a response. A VAX 8600, a rather faster machine, can be loaded more heavily before delays become noticeable, let alone objectionable. For general use, the PPM Services on a machine with the speed of a VAX 785 would have to provide better response. This could be done by implementing the LPM agents as threads in a common address space (except for slave agents of the PPM Creation Service, which have to be separate). On faster machines, this optimization would be helpful, but not clearly necessary.

A fully operational PPM system will require many new client programs. The prototype has only a primitive command interpreter and a few utility programs. These are sufficient for testing, but do not, as yet, take full advantage of the possibilities inherent in the PPM Services. One important area of work will be the integration of the distributed program management facilities of PPM with load sharing facilities. Another area will be the development of more flexible command interfaces that take better advantage of bitmapped displays and pointing devices, such as mice. A third area will be the development of program monitoring tools, that give the user a better picture of the state of the computations in his PPM Session. A fourth area is the use of PPM to integrate computing in networks of machines running heterogeneous operating systems. This is discussed in the following chapter.

CHAPTER 9

PPM IN A HETEROGENEOUS WORLD

9.1. PPM and nonUNIX Systems

The preceding chapters have discussed the services provided by PPM and their relationship with the UNIX system. PPM, however, offers functionality that is needed by a variety of systems. Moreover, since the services are offered by processes outside the kernel, they can offer compatible constructs and interfaces on these systems. This allows the creation and control of jobs spanning heterogeneous systems, provided the underlying systems are able to send messages to one another. The current PPM prototype has not been ported to other systems, although its design has been heavily influenced by experiences in programming program management facilities on the QuickSilver system [HMS88]. Nonetheless, it is appropriate to consider the relation between the facilities of PPM and the needs of nonUNIX operating systems.

This chapter considers the appropriateness of the PPM Services for three recent operating systems, Mach [BRS86] [Ras86], V [Che88], and QuickSilver. Appropriateness has three aspects. There must be a need for the services, the need should not be filled by the current system, and the services should be implementable in the system. In general, all of these systems allow distributed programming, but lack the sort of high-level program management facilities provided by PPM. Each introduces innovations in program management, many of which can be useful in implementing the PPM Services. These management facilities, however, are not sufficient in themselves, or in concert with current shells, to provide an environment supportive of multiprocess and distributed computing. In each of these systems, it is appropriate to provide program management through a more sophisticated, distributed shell, like PPM.

9.2. The MACH System

MACH is a UNIX variant that offers support for multiprocessors, a virtual memory design supporting shared address space segments and memory-mapped files, capability-based interprocess communications, and RPC-style client-server interfaces [Ras86]. In many ways, however, MACH continues to use the abstractions provided by UNIX, as discussed in Chapter 2. MACH, therefore, can benefit in many of the same ways as UNIX from the functionality and abstractions of the PPM Services.

The MACH system introduces into UNIX the idea of threads and processes (threads and *tasks* in MACH terminology) as distinct units. MACH supports thread control through explicit kernel calls to suspend, resume, and terminate threads, and process control through explicit kernel calls to suspend, resume, and terminate all of the threads in a process. The system, however, does not introduce any new multiprocess unit, such as PPM's job. For job control, MACH's current implementation continues to rely upon (and their kernel interface offers no substitute for) UNIX process groups, despite their limitations in a distributed system. Similarly, while MACH introduces bookkeeping functions to obtain the state of threads within a process, there is no bookkeeping support for keeping track of the processes that may be cooperating as a

larger unit.

MACH supports remote procedure calls and the construction of server processes in many ways, but relies upon two simple mechanisms for binding logical names with addresses. Each process has access to a network name server and an environment manager. The name server allows server addresses to be bound to simple names. These bindings are shared by all the processes on a given machine, and may be exported across the network. An environment manager [Tho87] provides environments through a server process, but allows only total sharing of the environment with other processes, or an environment copied from a single parent environment, or an environment with no shared or inherited bindings.

The functions of the PPM Services are all useful in the MACH system and, largely, do not replicate functionality that is provided by the underlying system. In MACH, however, some of the services may be easier to implement than in UNIX. This is partly because of the support for multithreaded computations, and partly because of the new system calls provided, and partly because of the structure of the system.

The foremost advantage of the MACH structure for implementing the PPM Services is the IPC interface to the kernel. This allows a Bookkeeping agent to receive notification messages from the kernel, with less intrusive modifications than those required in UNIX. The existing MACH system allows holders of sending rights to receive notification of the destruction of ports [BBB88]. Each thread or process has a port, called its *kernel port*, which is used for thread or process control. By giving the Bookkeeping agent send rights to this port, it can receive notification of the destruction of a thread or process. While this is helpful, it does require that the PPM Services obtain rights to this port. This is simple for those threads and processes created by the PPM Services, but requires kernel support if it is to be done automatically for other threads and processes.

The MACH system calls include calls to obtain the status of threads and processes, which are helpful for implementing the PPM Bookkeeping Service, because the Bookkeeper agents would not have, as in UNIX, to rely upon kernel notification or more expensive means of looking into the kernel address space to obtain this information.

MACH supports both multithreaded computations within the same address space, and address spaces that share segments. Both of these could be used to improve the performance of the PPM Services. In particular,

- the Environment Service would have access to information currently held by the Creation Service, allowing faster binding evaluation;
- the Creation Service would have access to binding information, allowing faster process instantiation;
- the Creation Service would have access to the bookkeeping information, allowing faster registration of new processes.

The IPC models of Berkeley UNIX and MACH are distinctly different, but both can be used to implement the PPM IPC model. MACH IPC occurs through ports, from which one process can read and to which some variable number of processes can send. These have a close relationship to the PPM port construct. A send capability for a MACH port can be distributed through the PPM Environment Service to allow the proper set of processes to access it. The receive capability can be placed in a process's private environment context. One-way byte streams are constructed less directly in MACH, but can be created using an extra buffer process, or through a file system service as in V [Che87].

9.3. The V System

The V system relies upon its team servers for program management. The team server offers some services similar to those of the PPM Creation and PPM Bookkeeping Services. It is responsible for the creation of threads and processes, for thread control, and for keeping track of the threads' resource utilization. The team server, however, is entirely a manager of local resources only. Job control is facilitated by placing a group of threads in a single process group. The membership of the group is not kept at any one place. Instead, the team server on each machine keeps track of the group members on that machine. Job control is exercised by messages multicast to team servers. The recipients then perform the operation upon the local group members.

V's multicasts are not always reliable. Messages can be undelivered, and, even more likely, the responses can overrun the available buffers. Cheriton and Zwaenepoel, in fact, found that on a machine with two network interface buffers, they lost roughly one-in-three response packets for groups of size three and two-in-four for groups of size four [ChZ85]. Presumably, this trend continues for larger groups. This has the curious effect of making it harder to find out the membership of a group than to terminate the group. Because of this unreliability, operations like *DestroyProcess()*, which relies upon reception of the outgoing message rather than upon replies, must spend significant amounts of time retransmitting in case outgoing messages are lost. Operations like *QueryGroup()* (obtain membership list), which relies upon the reception of replies, are even harder to manage, because of the impossibility of detecting packet loss, and the consequent uncertainty about whether information gathering is complete.

The lack of higher level, distributed management facilities in the team servers results in unnecessary delays, in difficulty in obtaining accurate information about job status and membership, in the possibility of orphan processes, and other problems. These difficulties are of limited importance when distributed and offloaded jobs are rare. In systems where such activities are more commonplace, it is more important to avoid these problems.

PPM addresses these difficulties by providing a distributed program manager. Job membership information is kept explicitly, although still in a distributed format. PPM Session membership is also kept explicitly, so that there is no doubt about whether information is being lost due to communications failure. The PPM Service is able to take on some other functions of the team server as well. Notably, it provides bookkeeping and notification services. V does not inform servers when clients fail, assuming that they will be able to handle failure detection and recovery. Programming failure detection as part of every distributed application, however, would be absurd, since it would result in missing, malfunctioning, or redundant mechanisms. PPM assists client applications through notifications from the Bookkeeping Service and through exit actions.

The PPM Service agents are not simply team servers with additional functionality. They differ from team servers most strongly in that they provide services to a single owner, while team servers offer services to all clients. The reasons for this organization have been explained in previous chapters, but we can quickly review the three main reasons.

- Clients can be authenticated based upon capabilities, rather than upon access control identities. It is clear what information a client is to be given access to and what actions he is to be allowed, even when the user plays different roles. Moreover, a client is able to create processes and jobs for different identities without having to hold the authentication information for those identities.
- Dividing responsibility for bookkeeping between per-owner servers can allow more flexible policies for information storage and disposal. In particular, the servers can be more reliant

upon and more trustful of their clients in managing the information held. Per-machine servers are forced to reclaim storage more aggressively, and, since they are very long running, cannot tolerate the accumulation of garbage.

- The Bookkeeping and Creation Services are able to rely upon the Environment Service as a repository of information. The Bookkeeping Service looks up timeout values in the environment. The Creation Service looks up information on the default configuration for jobs from the environment. This simplifies the interface to the Creation Service.

The PPM Environment has a pronounced difference with the V name resolution mechanisms, and they should not be confused. The V system uses character string names to refer to objects. These names are resolved through a series of context servers that generally parse the name from left to right, implementing a hierarchical context system. The initial context server is a per-user server that can allow aliasing. Name resolution in V, unlike binding evaluation in PPM, occurs as part of the invocation of operations on the objects. This was specifically avoided in the design of the PPM Environment, because this would necessitate, in systems like UNIX, the interception of kernel calls, and would confound the PPM Environment name space and the file system name space. PPM therefore separates the evaluation of bindings and the invocation of operations, even when used in a system like V. Currently, V uses only a simple environment structure derived from the UNIX shells, so the advantages of the PPM Environment in other systems also apply to programming in V.

The V system has many features that can help in implementing the PPM Services. Multicast communication is one example, provided that future implementations can provide greater reliability. Some functions of the PPM Creation Service and the PPM Bookkeeping Service are simply expansions of functions undertaken by V's team servers. The model of threads and processes, for example, are in good accord, and the model of job and command session present no problem.

The greatest difference in the models presented by V and by PPM is in the configuration of computations. Unlike UNIX (and MACH), V does not represent open files, communication channels, or message ports as explicit objects managed by the kernel. Instead, every thread is a message port, and open files, devices, and communication channels are represented by information held within a process's address space. Consequently, while the initial configuration of a job will be known to the PPM Creation Service, subsequent changes, such as new communication channels or additional open files, may not be known. They will not be known to the kernel, so we cannot rely upon the kernel for notification. They could be made known by library routines run by clients as part of configuration changing operations. They could also be made known by the servers themselves, including the file server, the pipe server, and so forth. The former solution is probably preferable on the grounds that the PPM mechanisms should not be forced upon users, and because of the difficulties involved in informing servers of the addresses of a user's LPM servers.

9.4. PPM and the QuickSilver System

In Chapter 2 we described the way the notion of transactions was built into the QuickSilver system. A transaction is an identifier for an ongoing activity, which should ideally be *well-behaved.* Well-behaved activities are atomic, irreversible, and, perhaps, serializable in their effects on resources that outlast the activity. These properties are guaranteed by resource managers that interact through message exchanges with a distributed transaction manager.

A process in QuickSilver, is a resource associated with a *default transaction*. It is created by a server process (the Process Master) that acts as manager of all the local processes. Other resources used by the process may be associated with the default transaction, such as open files or display windows. A process's default transaction provides resource managers, including the Process Master, with a consistent means of handling the process's termination. The process is a critical resource for its default transaction. The termination of the process, therefore, causes the Process Master to call for the termination of the default transaction. When a process terminates normally, its default transaction is committed. If the process terminates because of site failure, a process control operation by some other process, or an error, its default transaction is aborted.

Aborting a transaction causes resource managers to clean up the associated resources. Since a process is a resource associated with its default transaction, the Process Master terminates the process as part of cleaning up. Thus, transaction commitment and abortion provides an alternative to process control operations for terminating processes, but it is probably more reasonable to think of process termination as a side effect of abandoning an activity. Transactions do not provide any reasonable analogue for process suspension and resumption. This is particularly important in multiprocess jobs, where two interdependent processes may have to be *terminated* together without implying that they ought be *suspended* together.

A multiprocess job can be viewed as a single activity represented by a single *job transaction*. The processes are resources associated with this transaction, but in contrast with the case of a single process job, they are not necessarily *critical* resources. The resource manager cannot, therefore, automatically link the failure of a process with the abortion of a job transaction. A different strategy is needed.

The possibility of other management strategies is eliminated if the job transaction is used as the default transaction for all processes of the job. The failure of any process requires the abortion of its default transaction, so that servers, such as the file server, holding state information associated with the process can clean up. Therefore, if a single default transaction is shared by the component processes, the failure of any process must cause the abortion of the whole job.

If component processes are either critical or noncritical, the job transaction can be used as default transaction for all critical components and some other transaction for all noncritical components. Components that wish to receive notification of the failure of noncritical components can participating in their default transactions. For this to function correctly, however, a manager must keep track of the nesting relationship between these transactions. Aborting the job transaction, after all, must terminate the noncritical as well as the critical components. This would be handled by the underlying transaction managers if QuickSilver supported *nested transactions* [Mos85], but without support from the transaction manager, the nesting relationship must be explicitly handled by the resource manager. The interface to the Process Master, however, does not allow this relationship to be expressed, since there is no support for a multiprocess job abstraction. This, of course, PPM provides.

LPM agents for the PPM Creation Service can provide the job management that the Process Master lacks. It is possible that they could supplant the Process Master altogether. This, however, would depend greatly on as yet unspecified security arrangements. Alternatively, nested transactions could be supported in QuickSilver allowing dependencies to be expressed through the transaction hierarchy.

PPM follows a program management strategy that is somewhat more flexible than can be implemented, even with nested transactions. Through exit actions, PPM allows the mapping of process exit status to job and process control actions. These mappings can change, so that a

process can be changed from critical to noncritical, or *vice versa*. This could be useful, for example, for a server process that is critical until it starts a backup server process. The PPM approach also allows the development, should it prove necessary in the future, of richer classes between critical and noncritical, such as groups of processes of which it is critical that at least one survive. Exit actions also allow more flexible subsets of processes to be controlled as a group. While a process can have only one default transaction, exit actions from any process could terminate it. Exit actions that specify that a notification message is to be sent upon failure can change the target of the message. Which process sets exit actions is also flexible using PPM, since they are set in the environment, and may be inherited. Using transactions alone, a parent process must specify whether its child is critical, while the recipient of notification messages must explicitly request participation in any default transactions of which it needs to be apprised.

It is not clear how important future programmers will judge flexibility of the sort provided by PPM. At present, we can only point to the differences with program management based upon nested transactions and to the fact that PPM-style management hides the use of transactions and can, in fact, be implemented without an underlying transaction-based recovery management system.

9.5. Interoperability

The preceding sections have argued that the program management facilities of PPM are appropriate for several modern operating systems, and that versions of the PPM Services could be offered on these systems with relatively small changes in the underlying kernels. A PPM Session can include machines running different operating systems, provided that the LPM agents on the different machines are able to communicate with one another and agree upon the protocols for various operations. All operations in PPM are undertaken by agents at the affected site, so all remote operations are mediated by a remote agent. This means that agents on one machine do not interact with the system-provided services of another site, even when the underlying system makes its services remotely accessible. Thus, LPM agents generally do not need to take account of operating system heterogeneity. There is a slight exception, however, in that the PPM Session Maintenance Service must understand how to authenticate itself to the PM Master Service on a remote machine, using mechanisms that may be system specific.

The design of the PPM Services is for the most part system independent. The PPM Environment Service stores uninterpreted bytes. The PPM Creation Service implements operations on a job following a model defined by PPM. The PPM Bookkeeping Service stores and reports information in a format based upon this model. The information it stores may, however, be somewhat system dependent. For example, PPM identifiers for threads must be mapped to the identifiers used by the underlying system. The Bookkeeping Service may also store information that lies outside the PPM job model, for example reporting two-way byte stream connections between processes. We have not specified the interfaces through which clients will be able to retrieve this system specific information. Care must be exercised in the design of these interfaces to keep system dependencies to a minimum.

One of the advantages of having a uniform job model on dissimilar systems is that programs using the model can be more easily ported from one system to another. This has been the case with many UNIX utilities following the pipeline model. It may be the case with future applications following other models, particularly those basing interactions on remote procedure calls. PPM provides remote execution facilities and configuration facilities that, in conjunction with library routines, can hide many of the system specific features of a multiprocess or distributed program. This should make them easier to port between systems. The PPM Environment

can also be used to hide information to some extent, but, since the value bound to a name may be used in system specific ways, this does not guarantee easy portability. The current PPM prototype has not been ported to different systems, although portability has been a concern in its design. Therefore, evaluating gains in portability due to the use of the PPM models remains in the realm of future work.

CHAPTER 10

CONCLUSIONS

10.1. Evaluating PPM

The development of PPM has involved many design decisions. Some were made early and are basic to the conception of PPM, while others were made only after alternative choices had been tried and discarded. This final chapter singles out some of these design decisions and comments upon their correctness or convenience in the case of PPM and upon their importance for designers of future systems incorporating similar ideas. The concluding section points out some of the accomplishments of PPM and identifies some areas of future work.

10.2. The Design of PPM

10.2.1. Basic Organization

As PPM has developed, much has been added and much has been changed. Certain features of PPM, however, have been kept consistent, since they were fundamental to our view of what PPM should provide. The following are such fundamental design choices:

(1) PPM is a system-independent set of services, rather than an extended kernel.
(2) PPM provides personal services for individual users.
(3) The services are offered by agents created and destroyed transparently.

(1) Client-Server model

PPM provides services through a procedure call interface, rather than attempting attempting to provide network operations by distributing a particular kernel. This choice was made because it is necessary to allow dissimilar systems to cooperate across a network and to allow the definition of multiprocess constructs not supported by the underlying systems.

This approach stands in contrast to other attempts to provide access to a distributed environment, which strove to preserve an existing system interface, such as LOCUS [PoW85] and NEST [AgE85]. While these other projects do provide access to a distributed environment, they are best suited to offloading single process jobs, rather than developing distributed applications. The interfaces that these systems support simply do not provide sufficient support for many multiprocess jobs.

(2) *Personal Services*

The PPM Services support jobs run and controlled by a single user. The services do not offer support for running multiuser jobs, or for inquiring about jobs and processes belonging to other users. This decision is quite reasonable for the Creation Service. This service accommodates multiple identities, without confusing the question of identity with ownership by an individual. The Bookkeeping Service maintains this distinction, providing bookkeeping only for the jobs and processes that a user owns. This means that some other form of bookkeeping is necessary if the user is to have access to information about processes belonging to other users. The type of information that will be needed by (or that is appropriate to hand out to) users other than the owner is different than that held by the PPM Bookkeeping Service. Information of

concern to other users regards the resources being used by some person, rather than the logical structure of that person's jobs. A natural request about other user's work would be a request for a list of processes at a specific site. Job relationships are irrelevant here, as is information about processes that have terminated. Public information can be kept locally at each site; distributed bookkeeping is unnecessary. The Environment Service also has a user specific function. A separate system-wide name service is needed to provide a means of sharing bindings between users. Such a name service can be integrated with the Environment Service, but has fundamentally different concerns, in terms of performance, reliability, and naming.

(3) *Transparently Available*

Client programs requiring one of the PPM Services contact the PM Master to obtain the contact address for the service's LPM agent. The PM Master creates the agent is necessary, transparently to the client program. Clients therefore do not explicitly call for the creation of LPM agents, nor do they call for their termination. Both are in the hands of the services themselves. This approach is useful, because it frees clients from any administrative duties, making them easier to write. It also makes the administration of the PPM Session independent of any client failures or errors.

While the decisions described above were consistently followed, other decisions evolved as different prototypes were programmed. The answers to the following questions changed as PPM evolved:

(1) Which services must be offered?
(2) How should session membership be maintained?
(3) How should agents be structured?

(1) *Services to be offered*

The original conception of PPM included only the functions of the Bookkeeping Service and the Creation Service. These services are sufficient to create and control distributed programs, but fail to address many of the issues that arise in distributed programming and in making use of distributed systems. These issues became clearer to the author when working on the QuickSilver system, in which the use of services is more common than in UNIX, and hence service addresses must be dealt with in more flexible ways. The potential of the environment as a means of tailoring the use of resources to individuals struck us as an important part of making distributed systems more usable. The construction of an appropriate environment for each process is an interesting problem, which the PPM Environment Service addresses in a novel way.

(2) *Knowledge of session membership*

The first versions of PPM did not keep consistent membership information through the PPM Session Maintenance Service. It was originally thought that keeping such global information should be avoided, because it would limit the scale of the PPM Session. Instead, the user's LPM agent at each site kept a list of known peers, with whom communications channels were established. Sending messages between LPM agents was a somewhat difficult task, involving routing through intermediate agents. This approach was motivated partly by scalability considerations, and partly because the only reliable protocol available required point to point connections, which were a limited resource in earlier versions of UNIX.

In the most recent version of PPM, it was recognized that PPM Sessions of sufficient scale to strain the ability to keep consistent lists were unlikely. An explicit Session Maintenance Service was therefore added. It was also decided that communications should be based upon protocols that will reliably deliver messages.

(3) *Implementation of agents*

The original implementation of PPM did not distinguish between the Creation and Bookkeeping Services. Both were provided by a single LPM agent. This was complicated to program, because it was desired that the agent be able to start new operations while awaiting the results of distributed operations. This evolved into an implementation using lightweight processes to facilitate the overlapping of operations. When separate agents were provided for the different services, it became easier to structure the agents so that parallelism was allowed, but without the assistance of a lightweight process package. Lightweight processes (or coroutines) are a useful programming concept, but the same effect is available by other means. At present, the combining the agents for the various PPM Services in a single address space is seen as a possible optimization, rather than as an important feature of the design.

10.2.2. The PPM Environment Service

Designing the Environment Service entailed a number of interrelated decisions. The four principal decisions were

(1) to separate the context structure from the process structure;
(2) to base evaluation on path lookup and inheritance;
(3) to build into the context structure regions for command sessions, jobs, and processes;
(4) to allow shared bindings.

There were alternatives to all of these decisions, which are discussed below.

(1) *Independent Context Structure*

In UNIX, bindings for a process are held in the process's address space, tying the contexts to the processes. By providing contexts that are independent of processes as in the PPM Environment, the contexts are granted an independent existence. Granting independent existence to multiple contexts allows the construction of a more flexible and useful environment, but requires two further decisions, namely, what determines that a context should be created or destroyed, and how should the contexts be related to processes.

The PPM Environment Service creates contexts that belong to a single user, and which do not ordinarily persist after the end of the PPM Session. Contexts are created and destroyed implicitly. There is no need for a special context creation operation, similar to a directory creation in the file system, because contexts provide no attributes such as ownership. There is no need for an operation to destroy contexts, because they are not persistent, and can always be eliminated by removing all of the bindings they contain.

(2) *Path Lookup and Inheritance*

Once we have decided that each process will not have a private context and that more than one process exists in a PPM Session, it is necessary to determine where a process will find its bindings. The PPM Environment Service uses both path lookup and inheritance. Path lookup is an important principle, because it allows a process to obtain bindings on the basis of multiple attributes. Inheritance is similar to path lookup, since inheritance can be implemented by looking in each context between the specified context and the root. It differs from path lookup in that path lookup is not used in evaluating the binding for a fully qualified name, while inheritance is. Inheritance is important because it allows the overriding of particular bindings without explicitly resetting all other existing bindings.

(3) *Layers of Contexts with Specific Functions*

The name space for the PPM Environment allows the specification of sharing patterns when a binding is placed in the environment. The name space we have been discussing has four distinct horizontal divisions: the global region, the command session region, the job region, and the process region. A single command session context and a single job context lie between the global region and any particular process context. This is somewhat rigid, since it disallows sub-command sessions and subjobs, which are useful concepts. This rigidity was quite intentional, since it provides a clear model. It has been possible to work around this rigidity, but at the expense at times of increased programming complexity.

(4) *Shared Bindings*

The fourth design choice was that shared bindings should be allowed. This facilitates the use of the Environment as a means of communication. This is an important function and so cannot be dispensed with. The decision to allow updates of any binding provides flexibility, but increases the expense of evaluation. Evaluation can be made less expensive by caching values in the address space of the client, but the initial evaluation, which is the only one performed for most bindings, is still expensive.

10.2.3. The PPM Creation Service

The principal choices to be made in designing the Creation Service had to do with the definition of a job and the way in which interprocess communication would be supported. They were

(1) A job is defined as simply a set of processes.
(2) One-way byte streams and ports are the only supported forms of IPC.
(3) IPC configuration is only allowed before a job starts running.

(1) *A job is simply a set of processes*

The definition of a job as simply a set of processes was not the original choice. At first, it was thought that it would be better to define a job as a tree of processes related by logical parentage. A subjob would be a subtree. While this is a reasonable default, there is no reason to require that all the processes in a job be logical descendants of a single root process. A job can be a forest of trees of related processes. This has the benefit of allowing the addition of a process to a job without the necessity of either convincing an existing process to create the new process, or of somehow patching the process into the tree as if it were related. Originally, it appeared that the relatedness condition would be important if the UNIX environment was to be used, but the PPM Environment's context structure does not require this.

(2) *Limited forms of IPC*

PPM takes a rather conservative view of IPC mechanisms. Support is given for one-way byte streams and for message ports. These are clearly two useful forms of communication, but other forms of communication exist. For example, one might choose to support two-way byte streams, multicast groups, or message ports with a choice of reliable or unreliable semantics. Support for some of these other forms of communication can be added to the Creation Service as they prove necessary. Expanding the set of IPC mechanisms to be supported, however, could result in problems, if the mechanisms are unsupportable in some systems. Even if a mechanism is supportable, it might be excessively troublesome to do so. Two approaches can be taken to this problem. The first is to divide the IPC support into required and optional categories. The second is to simply define PPM's IPC support as limited to the required models. The second approach is more practical for prototype systems.

The PPM Creation Service as presented supports only two types of IPC. Support is given to one-way byte streams and reliable message ports. It is argued that these are appropriate forms of IPC to supply at the command language level. Other forms of IPC might be supported by the PPM Creation Service. This is an area which will evolve as communication mechanisms evolve.

Configuring communications in PPM has two aspects: the creation of communication endpoints and channels, and the insertion in the environment of information describing the configuration. This reflects the idea that, to build flexible software, it is as important to have a consistent means of informing the running program of its configuration as to be able to actually configure the communications. The important information to communicate to the program includes the number of communication objects, their types, and their addresses, descriptor indices, or similar information.

The communication configuration facilities that the PPM Creation Service provides are limited. They are intended to allow the configuration of multiprocess programs before the programs run. The services do not perform functions that could not be performed by a user written program assisted by the PPM Environment Service. What they do is release the program from the responsibility of setting up its communications for itself, in the same way that the pipeline commands of the UNIX shells relieve programs from the need to set up their standard input and output connections. Rather than providing the configurations only through the command interpreter, though, these functions are made available to any client through calls to the Creation Service.

(3) *Configuration before execution*

The configuration facilities are not intended to allow the reconfiguration of running programs. It is not clear how useful such facilities would be. They are also difficult to provide on many systems, and are therefore far less portable than the static configuration facilities provided by PPM. With greater experience in everyday distributed programming, we shall see if dynamic reconfiguration is needed.

10.2.4. The PPM Bookkeeping Service

The bookkeeper keeps track of information reported about jobs either by the other PPM Services or by the underlying kernel. The following decisions were important in its design:

(1) *Rely on the kernel for information.*
(2) *Use distributed, rather than centralized, bookkeeping.*

(1) *Reliance upon the kernel for information.*

By and large, the kernel requirements of PPM are met by existing systems. The Bookkeeping Service attempts to provide accurate information about the user's jobs in the system. However, it must be informed of the user's actions. Some of this information can be delivered to the Bookkeeping Service by the client programs, if they use library routines which send messages reporting on their actions. Some information, however, cannot be obtained in this way. For example, in UNIX it is hard for a process to obtain information about termination and state changes of processes that are not its direct children. This information can be provided by the kernel as was done in [MMS86]. Most of the development of PPM has been on unmodified UNIX systems, with certain restrictions on the accuracy of the information available through the Bookkeeping Service. These restrictions, however, might be unacceptable in a production version of PPM, but, with rather modest changes in the kernel, they are unnecessary.

(2) *Use of distributed bookkeeping*

The interface to the PPM Bookkeeping Service does not compel the implementation to use distributed bookkeeping, but this is the most reasonable choice. A centralized scheme, where information about a user's jobs were held at one site, would make client queries trivial to answer, but would have two distinct drawbacks. First, updates and the most common queries would be slow for all but one site, since they would require a remote operation. The queries that would be speeded up are the presumably infrequent requests for information about the user's jobs. The queries that would be slowed down are the more frequent exchanges between the LPM agents for a particular site. Second, centralized bookkeeping is subject to disruption with the failure of a single site. Unless sufficient local bookkeeping is provided, important information such as job membership will be lost irretrievably.

10.3. A Final Assessment of PPM

10.3.1. Accomplishments

Interactive program execution facilities began as simple command interpreters that could load new program code segments and start their execution [MMM72]. Later systems allowed the command interpreter to create new processes [BBM72] [RiT74]. These command interpreters were able to provide some services internally and to rely upon the underlying kernel for others.

This arrangement, however, is outdated. It has shown weaknesses for three important reasons. First, network operating systems make it possible to run jobs on multiple systems, but the structure of the shell may make this difficult, and may impose distinctions in the way local and remote jobs are managed. Second, bitmapped displays make it possible for a user to engage in multiple interactive sessions simultaneously and to use nontyping input techniques. The structure of the shell, however, limits the sharing of information between sessions, and limits the use of nontyping interfaces. Third, the use of diverse resources can require a greater flexibility in the identities used for programs. The shell, however, may not allow identity to be changed within a single session. Dividing a login session on the basis of location, user interface, or identity imposes unnecessary constraints.

PPM allows the owner to operate within a single login session by basing that session on distributed personal services, rather than upon the services provided by a single process, or directly upon the services provided by the underlying operating system. The use of personal services has advantages over using system services, both in flexibility in designing the services, ease of implementation, and in the degree to which authentication issues can be hidden from clients.

PPM identifies a set of services that are required to support login sessions, and makes them available to all user programs. This removes the distinction between the services available through the shell to the interactive user and to command scripts and those that can be used by application programs. This has the dual effect of, on the one hand, making it easier to write distributed applications, while, on the other, making it easier for the user interface to evolve.

The PPM Services support a job model that is appropriate to distributed systems, in that jobs may cross the boundaries between machines or even between operating systems. This is a necessary step for the sensible management of distributed programs. The support integrates remote execution with bookkeeping. The bookkeeping, moreover, is devised to provide the user with information about all parts of a job, even those that are no longer running. This is frequently important if multiprocess jobs are to be debugged. System bookkeeping in systems like

UNIX, which are oriented toward single process jobs, does not preserve this sort of information.

The PPM Environment makes it easy to share information with a PPM Session. Moreover, the rich structure of the environment makes it possible to restrict the scope of sharing. This makes the environment a much more useful form of information repository. The environment is a convenient means of tailoring programs to user preferences. It is therefore of importance for all sorts of programs having nothing to do with distribution. Thus, the PPM Environment could change not only the way multiprocess and distributed programs are written, but many single process programs as well.

PPM provides a framework for using a distributed system. This framework is important for load redistribution and for accessing resources offered by heterogeneous systems, as well as for distributed programming. The "personal" approach taken by PPM has proven to have benefits that were initially unforeseen, but which argue for the use of services offered by personal agents in the design of future systems.

10.3.2. Areas for Future Work

Work on the support of distributed and multiprocess computation must continue, both with the development of services like those of PPM, and with the integration of these services with other tools, such as program monitors and debuggers. Some of the areas to be considered are the following:

- *Load sharing:* A distributed shell, such as that described in Chapter 7, is a useful tool for running distributed applications. Distributed applications, however, are at present quite rare. One of the important motivations for developing PPM has, therefore, been the desire to run single-process applications remotely, so that the load on a set of available machines is kept in rough balance. Load sharing strategies through remote job placement and through process migration have been extensively studied [The86] [Zho87] [KrL88]. Implementations have often involved the submission of jobs to a special server or subsystem that gathers load information and places the job appropriately [HCG82] [Ber85] [Hag86] [Ezz86] [LLM88]. Unfortunately, single-site shells are not well suited to managing offloaded work.

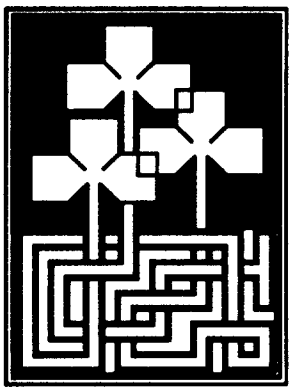
  PPM is designed to allow such load sharing to take place without placing offloaded work outside the shell session. Just as the user can request that the PPM Creation Service place a job on a remote machine, the user could request that the Creation Service choose an appropriate machine. The Creation Service agent could consult a system load information server, such as that described in [Zat85], for assistance in choosing an appropriate machine. PPM is thus compatible with load sharing mechanisms, but work is required on some of the practical aspects of sharing. Using new machines requires the extension of the PPM Session, which could be more expensive than running the job locally, and the extension may require the user to be prompted for a password or other information. Appropriate policies and mechanisms must therefore be developed for load sharing using the PPM Services.
- *Process migration:* We have not considered the possibility that the underlying system might allow processes to migrate, but PPM does not prevent migration and in some ways provides support for it. The interface to the PPM Creation Service could be expanded to include a call requesting that a process be moved from one machine to another. If the underlying systems cannot accommodate this, the call would fail. The Bookkeeping Service could keep track of the current location of the process, so that process and job control actions would occur correctly, and so that correct information would be given to clients.

  The PPM environment can accommodate mobility, but some care must be taken to ensure the correct mix of semantics. The PPM Environment Service uses path lookup in evaluating

bindings to provide a form of multiple inheritance. One of the paths of inheritance is dependent upon the process's location. If the location changes, the machine dependent bindings will automatically change to those appropriate to the new machine (cached values would, of course, have to be flushed). It is not clear, however, that this change is desirable in all cases. The process migration facilities in the Sprite operating system [DoO87], for example, allow a process to move while remaining, logically, at the original "home" site. In this system, migration should probably be kept invisible to both the PPM Environment Service and the PPM Bookkeeping Service. The pains taken by the underlying migration facility to present logical immobility should not be undermined by services lying on top of the kernel. In the V system [TLC85] [The86], processes are attached to logical hosts, which, upon migration, are rebound to new physical hosts. In this case, however, the intent is to provide a flexible indirection, rather than a fixed indirection as in Sprite. Therefore, it would be appropriate to make the migration visible through the PPM Services.

- *New command languages:* Chapter 7 outlined some modest extensions to the command language used by the UNIX shells to accommodate distribution and nonpipeline job organizations. More powerful command languages can be used that provide graphical interfaces. This may prove to be a more natural means of expressing configurations.
- *Extended job models:* Job configuration under PPM is limited to the configuration of jobs whose processes have nonoverlapping address spaces. PPM does not provide facilities for configuring overlapping address spaces. There are two reasons for this. First, many systems do not allow shared writable memory segments. In particular, the implementation of PPM components was done on Berkeley UNIX and QuickSilver, which do not allow shared segments. Second, it is not clear whether the management of shared address space ought to be exposed to the user, or always hidden by the compiler and runtime libraries. PPM also provides no facilities for configuring communication outside of one-way byte streams and message ports. For example, it provides no support for two-way byte streams, such as those available through TCP in Berkeley UNIX. The sufficiency of a job model not supporting these options cannot be proved. Only experience can help us here.
- *More powerful environments:* The PPM Environment Service can be extended in several ways. One goal should be the integration of the PPM Environment binding lookup mechanisms with a system-wide name server. Bindings should be able to derive values set through a name server for all users. There are several problems involved in this integration. One is that a call to a system-wide name server that fails to find a binding is wasted effort. This means that all bindings for which no value has been set become expensive to evaluate. A second goal might be the use of a more powerful indirection mechanism in the environment. Currently, indirections simply name other bindings to evaluate. A more powerful mechanism might allow the insertion of more elaborate instructions, such as a sequence of names to evaluate, or directions to consult a system-wide name server.

Ultimately, practical experience in everyday programming will be the proof or disproof of the value of the ideas and mechanisms discussed in this dissertation. Thus, success will be achieved when distributed programming is an everyday reality, and when we no longer think of logging into a system as logging into a particular machine.



BIBLIOGRAPHY

[AgE85]
R. Agrawal and A. K. Ezzat.
Processor sharing in NEST: A network of computer workstations.
Proc. First Int'l Conference on Computer Workstations, pages 198-208, San Jose, California, November 1985.

[Alm86]
G.T. Almes.
The impact of language and system on remote procedure call design.
Proc. of the Sixth International Conference on Distributed Computing Systems, pages 414-421, Cambridge, Massachusetts, May 1986.

[AFR87]
D. P. Anderson, D. Ferrari, P. V. Rangan and S. Y. Tzou.
The DASH project: Issues in the design of very large distributed systems.
Technical Report No. UCB/CSD 87/338, Computer Science Division, Universtiy of California at Berkeley, January 1987.

[AnF88]
D. P. Anderson and D. Ferrari.
The DASH project: An overview.
Technical Report No. UCB/CSD 88/405, Computer Science Division, Universtiy of California at Berkeley, February 1988.

[BRS86]
R. V. Baron, R. F. Rashid, E. H. Siegel, A. Tevanian, Jr. and M. W. Young.
MACH-1: A multiprocessor oriented operating system and environment.
pp. 80-99 in *New Computing Environments: Parallel, Vector and Systolic*.
SIAM, 1986.

[BBB88]
R. V. Baron, D. Black, W. Bolosky, J. Chew, D. B. Golub, R. F. Rashid, A. Tevanian, Jr. and M. W. Young.
MACH Kernel Interface Manual.
Dept of Computer Science, Carnegie-Mellon University, February 1988.

[BHM77]
F. Baskett, J. H. Howard and J. T. Montague.
Task communcations in DEMOS.
Proc. of the Sixth Symposium on Operating Systems Principles, pages 23-31, Purdue University, November 1977.

[Ber85]
B. Bershad.
Load balancing with Maitre d'.
Technical Report No. UCB/CSD 85/276, Computer Science Division, University of California at Berkeley, December 1985.

[BCM88]
M. Bhattacharyya, D. Cohrs and B. Miller.
A visual process connector for Unix.

IEEE Software, 5(4), July 1988.

[BiJ87]
K. P. Birman and T. A. Joseph.
Reliable communication in the presence of failures.
ACM Transactions on Computer Systems, 5(1):47-76, February 1987.

[BLN82]
A. D. Birrell, R. Levin, R. M. Needham and M. D. Schroeder.
Grapevine: An exercise in distributed computing.
Communications of the ACM, 25(4):260-274, April 1982.

[BiN84]
A. D. Birrell and B. J. Nelson.
Implementing remote procedure calls.
ACM Transactions on Computer Systems, 2(1):39-59, February 1984.

[Bir85]
A. D. Birrell.
Secure communications using remote procedure call.
ACM Transactions on Computer Systems, 3(1):1-14, February 1985.

[BLN86]
A. D. Birrell, B. W. Lampson, R. M. Needham and M. D. Schroeder.
A global authentication service without global trust.
Proc. 1986 IEEE Symposium on Security and Privacy, pages 223-230, April 1986.

[BBM72]
D. G. Bobrow, J. D. Burchfiel, D. L. Murphy and R. S. Tomlinson.
TENEX, a paged time sharing system for the PDP-10.
Communications of the ACM, 15(3):135-143, March 1972.

[BKK86]
D. G. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik and F Zdybel.
CommonLoops: Merging Lisp and object-oriented programming.
Proc. of Object-Oriented Programming Systems, Languages and Applications '86, pages 17-29, Portland, Oregon, October 1986.

[Bou86]
S. R. Bourne.
An introduction to the UNIX shell.
in *UNIX User's Supplementary Documents.*
Computer Systems Research Group, Univerisity of California at Berkeley, April, 1986.

[BMR82]
D. R. Brownbridge, L. F. Marshall and B. Randell.
The Newcastle Connection or UNIXes of the world unite.
Software--Practice and Experience, pages 1147-1162, 1982.

[BuM85]
D. A. Butterfield and R. M. Mathews.
Remote tasking.
in *The LOCUS Distributed System Architecture*, G. Popek and B. Walker, editors. .
MIT Press, Cambridge, Mass., 1985.

[CSC86]
L. F. Cabrera, S. Sechrest and R. Caceres.

The administration of distributed computations in a networked environment: An interim report.
Proc. of the Sixth International Conference on Distributed Computing Systems, pages 389-397, Cambridge, Massachusetts, May 1986.

[ChL85]
K. M. Chandy and L. Lamport.
Distributed snapshots: Detection global states of distributed systems.
ACM Transactions on Computer Systems, 3(1):63-75, February 1985.

[ChM84a]
J-M Chang and N. F. Maxemchuk.
Reliable broadcast protocols.
ACM Transactions on Computer Systems, 2(3):251-273, August 1984.

[ChM84b]
D. R. Cheriton and T. P. Mann.
Uniform access to distributed name interpretation in the V-system.
The Fourth International Conference on Distributed Computing Systems, pages 290-297, San Francisco, California, May 1984.

[ChZ85]
D. R. Cheriton and W. Zwaenepoel.
Distributed process groups in the V kernel.
ACM Transactions on Computer Systems, 3(2):77-107, May 1985.

[Che87]
D. R. Cheriton.
UIO: A uniform I/O system for distributed systems.
ACM Transactions on Computer Systems, 5(1):12-46, February 1987.

[Che88]
D. R. Cheriton.
The V distributed system.
Communications of the ACM , 31(3):314-333, March 1988.

[CMC88]
D. L. Cohrs, B. P. Miller and L. A. Call.
Distributed upcalls: A mechanism for layering asynchronous abstractions.
Proc. of the Eighth International Conference on Distributed Computing Systems, pages 55-62, San Jose, California, June 1988.

[CCC84]
UNIX Programmer's Manual.
Computer Systems Research Group, Univerisity of California at Berkeley, March 1984.

[CuW84]
R. Curtis and L. Wittie.
Naming in distributed language systems.
Proc. of the Fourth International Conference on Distributed Computing Systems, pages 298-302, San Francisco, California, May 1984.

[Dan82]
R. B. Dannenberg.
Resource sharing in a network of personal computers.
Technical Report CMU-CS-82-152, Carnegie-Mellon University , December 1982.

[DaH85]
R. B. Dannenberg and P. G. Hibbard.
A butler process for resource sharing on Spice machines.
ACM Transactions on Office Information Systems, 3(3):234-252, July 1985.

[DJA88]
P. Dasgupta, R. L. LeBlanc, Jr. and W. F. Appelbe.
The Clouds distributed operating system.
Proc. of the Eighth International Conference on Distributed Computing Systems, pages 2-9, San Jose, California, June 1988.

[DeS86]
A. L. DeSchon.
A survey of data representation standards.
RFC 971, USC ISI, January 1986.

[DoO87]
F. Douglis and J. K. Ousterhout.
Process migration in the Sprite operating system.
Proc. of the Seventh Int'l Conference on Distributed Computing Systems, pages 18-25, Berlin, West Germany, September 1987.

[Eri82]
L. W. Ericson.
DPL-82: A language for distributed processing.
Proc. of the Third Int'l Conference on Distributed Computing Systems, pages 526-531, Miami/Ft. Lauderdale, Florida, October 1982.

[Ezz86]
A. Ezzat.
Load balancing in NEST: A network of workstations.
Proc. 1986 Fall Joint Computer Conference, pages 1138-1149, Dallas, Texas, November 1986.

[GJK84]
H. Garcia-Molina, F. Germano, Jr. and W. H. Kohler.
Debugging a distributed computing system.
IEEE Transactions on Software Engineering, SE-10(2):210-219, March 1984.

[GrZ83]
T. Gross and W. Zwaenepoel.
Systems support for multi-process debugging.
Proc. of the ACM Software Engineering Notes/SIGPLAN Notices Symposium on High-Level Debugging, pages 192-196, Asilomar, California, March 1983.

[Hag86]
R. Hagmann.
Process Server: Sharing processing power in a workstation environment.
Proc. of the 6th Int'l Conference on Distributed Computing Systems, pages 260-267, Cambridge, Mass., May 1986.

[HMS88]
R. Haskin, Y. Malachi, W. Sawdon and G. Chan.
Recovery management in QuickSilver.
ACM Transactions on Computer Systems, 6(1):82-108, February 1988.

[HCG82]
K. Hwang, W. Croft, G. Goble, B. Wah, F. Briggs, W. Simmons and C. Coates.
A UNIX based local computer network for load balancing.
IEEE Computer, 15(4):55-66, April 1982.

[JoS79]
A. K. Jones and K. Schwans .
TASK forces: Distributed software for solving problems of substantial size.
Proc. of the Fourth Int'l Conference on Software engineering, pages 315-330, September 1979.

[JCD79]
A. K. Jones, R. J. Chansler, I. Durham, K. Schwans and S. R. Vegdahl.
StarOS, a multiprocessor operating system for the support of task forces.
Proc. of the Seventh Symposium on Operating Systems Principles, Pacific Grove, California, December 1979.

[Joy86]
W. Joy.
An introduction to the C shell.
in *UNIX User's Supplementary Documents.*
Computer Systems Research Group, Univerisity of California at Berkeley, April, 1986.

[JLH88]
E. Jul, H. Levy, N. Hutchinson and A. Black.
Fine-grained mobility in the Emerald system.
ACM Transactions on Computer Systems, 6(1):109-133, February 1988.

[KeR78]
B. W. Kernighan and D. M. Ritchie.
The C Programming Language.
Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[Kil84]
T. J. Killian.
Processes as files.
1984 Summer USENIX Proceedings, pages 203-207, Salt Lake City, Utah, June 1984.

[Kor83]
D. Korn.
Introduction to KSH.
1983 Summer USENIX Conference Proceedings, pages 191-202, Toronto, Ontario, Canada, July 1983.

[KrM85]
J. Kramer and J. Magee.
Dynamic configuration for distributed systems.
IEEE Transactions on Software Engineering, SE-11(4), April 1985.

[Kra83]
A. Kratzer.
A programming environment for distributed programming.
COMPCON 83, pages 35-41, Silver Springs, Maryland, September 1983.

[KrL88]
P. Krueger and M. Livny.

A comparison of preemptive and non-preemptive load distribution.
Proc. of the Eighth Int'l Conference on Distributed Computing Systems , pages 123-130, San Jose, California, June 1988.

[Lam86]
B. W. Lampson.
Designing a global name service.
Principles of Distributed Computing, pages 1-10, 1986.

[Lan80]
K. A. Lantz.
Command interaction in distributed systems.
COMPCON 80, pages 25-32, Washington, D. C., September 1980.

[LeM82]
R. J. LeBlanc and A. B. Maccabe.
The design of a programming language based on connectivity networks.
Proc. of the Third Int'l Conference on Distributed Computing Systems, pages 532-541, Miami/Ft. Lauderdale, Florida, October 1982.

[LeF85]
T. J. LeBlanc and S. A. Friedberg.
Hierarchical Process Composition in Distributed Operating Systems.
Proc. of the Fifth Int'l Conference on Distributed Computing Systems, pages 26-34, Denver, Colorado, May 1985.

[LFJ86]
S. J. Leffler, R. S. Fabry, W. N. Joy and P Lapsley.
An advanced 4.3BSD interprocess communication tutorial.
UNIX Programmer's Supplementary Documents, Vol. 1, 4.3 Berkeley Software Distribution, April 1986.

[LSB79]
V. Lesser, D. Serrain and J. Bonar.
PCL: A process-oriented job control language.
Proc. of the First Int'l Conference on Distributed Computing Systems, pages 315-329, Huntsville, Alabama, October 1979.

[LCJ87]
B. Liskov, D. Curtis, P. Johnson and R. Scheifler.
Implementation of Argus.
Proc. of the Eleventh Symposium on Operating Systems Principles, pages 111-122, Austin, Texas, November 1987.

[LLM88]
M. J. Litzkow, M. Livny and M. W. Mutka.
Condor -- a hunter of idle workstations.
Proc. of the Eighth Int'l Conference on Distributed Computing Systems, pages 104-111, San Jose, California, June 1988.

[MMM72]
The Multiplexed Information and Computing Service: Programmer's Manual.
Massachussetts Institute of Technology, Cambridge, Massachussetts, November 30, 1972.

[MMS86]
B. P. Miller, C. Macrander and S. Sechrest.

A distributed programs monitor for Berkeley UNIX.
Software--Practice and Experience, 16(2):183-200, February 1986.

[MNS87]
S. P. Miller, B. C. Neuman, J. I. Schiller and J. H. Saltzer.
Kerberos authentication and authorization system.
Project Athena Technical Plan, Section E.2.1, Massachusetts Institute of Technology, July 1987.

[Moo86]
D. A. Moon.
Object-oriented programming with Flavors.
Proc. of Object-Oriented Programming Systems, Languages and Applications '86, pages 1-8, Portland, Oregon, October 1986.

[Mos85]
J. E. B. Moss.
Nested Transactions: An Approach to Reliable Distributed Computing.
MIT Press, Cambridge, Mass., 1985.

[NeH82]
R. M. Needham and A. J. Herbert.
The Cambridge Distributed Computing System.
Addison-Wesley, Reading, Massachusetts, 1982.

[OSS80]
J. K. Ousterhout, D. A. Scelza and P. S. Sindhu.
Medusa: An experiment in distributed operating system structure.
Communications of the ACM, 23(2):92-105, February 1980.

[OCD88]
J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson and B. B. Welch.
The Sprite network operating system.
IEEE Computer, 21(2):23-36, February 1988.

[PoW85]
G. Popek and B. Walker, editors.
The LOCUS Distributed System Architecture.
MIT Press, Cambridge, Mass., 1985.

[Ras86]
R. F. Rashid.
Threads of a new system.
UNIX Review, pages 37-49, August 1986.

[RiT74]
D. M. Ritchie and K. Thompson.
The UNIX time-sharing system.
Communications of the ACM, 17(7):365-375, July 1974.

[RiT78]
D. M. Ritchie and K. Thompson.
The UNIX time-sharing system.
in *7th Edition UNIX Programmer's Manual.*
AT&T Bell Laboratories, 1978.

[Rit84]
D. M. Ritchie.
A stream input-output system.
AT&T Bell Laboratories Technical Journal, 63(8):1897-1910, October 1984.

[Ros88]
M. Rosenblum.
The performance of Sun's remote procedure call.
Technical Report to appear, Lawrence Berkeley Laboratories, 1988.

[Sal74]
J. H. Saltzer.
Protection and the control of information sharing in MULTICS.
Communications of the ACM, 17(7):388-402, July 1974.

[SJR86]
R. D. Sansom, D. P. Julin and R. F. Rashid.
Extending a capability based system into a network environment.
Technical Report CMU-CS-86-115, Carnegie-Mellon University, April 1986.

[SCB86]
C. Schaffert, T. Cooper, B. Bullis, M. Killian and C. Wilpot.
An introduction to Trellis/Owl.
Proc. of Object-Oriented Programming Systems, Languages and Applications '86, pages 9-16, Portland, Oregon, October 1986.

[ScS72]
M. D. Schroeder and J. H. Saltzer.
A hardware architecture for implementing protection rings.
Communications of the ACM, 15(3A):157-170, March 1972.

[Sco84]
M. L. Scott.
A framework for the evaluation of high-level languages for distributed computing.
Technical Report #563, University of Wisconsin, October 1984.

[SoM86]
R. J. Souza and S. P. Miller.
UNIX and remote procedure calls: A peaceful coexistence?.
Proc. of the Sixth International Conference on Distributed Computing Systems, pages 268-277, Cambridge, Massachusetts , May 1986.

[SpK86]
M. Spezialetti and P. Kearns.
Efficient distributed snapshots.
Proc. of the Sixth International Conference on Distributed Computing Systems, pages 382-388, Cambridge, Massachusetts, May 1986.

[SSS86]
Remote Procedure Call Programming Guide.
Sun Microsystems, Inc., February 1986.

[TLC85]
M. M. Theimer, K. A. Lantz and D. R. Cheriton.
Preemptable remote execution facilities for the V-system.
Proc. of the Tenth Symposium on Operating Systems Principles, pages 2-12, Orcas Island,

Washington, December 1985.

[The86]
M. M. Theimer.
Preemptable remote execution facilities for loosely-coupled distributed systems.
Report No. STAN-CS-86-1128, Stanford University, June 1986.

[Tho87]
M. R. Thompson.
MACH environment manager.
Internal memo of the Department of Computer Science, Carnegie-Mellon University, February 1987.

[WeO88]
B. B. Welch and J. K. Ousterhout.
Pseudo devices: User-level extensions to the Sprite file system.
1988 Summer USENIX Proceedings, pages 37-49, San Francisco, California, June 20-24, 1988.

[Zat85]
S. Zatti.
A multivariable information scheme to balance the load in a distributed system.
Technical Report No. UCB/CSD 85/234, Computer Science Division, University of California at Berkeley, May 1985.

[Zho87]
S. Zhou.
Performance studies of dynamic load balancing in distributed systems.
Technical Report No. UCB/CSD 87/376, Computer Science Division, University of California at Berkeley, October 1987.