THE PPM ENVIRONMENT MANAGER

Stuart Sechrest
Computer Science Division
University of California at Berkeley
Berkeley, California 94720

An environment is a set of name-value bindings, maintained for a particular user, that are provided to a process at runtime. This can provide a great deal of flexibility in tailoring the behavior of programs to a particular user's preferences. It can also serve as a simple means of interprocess communication. To be useful, however, the environment must take into account the natural structure of a user's work. Groups of cooperating processes form jobs and should share certain bindings. At the same time. processes in different jobs running on the same machine should share other bindings. A flat name space for bindings does not provide sufficient structure to handle these overlapping sets of shared bindings. The PPM Environment Manager provides an environment with a variety of contexts, allowing bindings to be, for example, machine-specific or application-specific. The environment, however, remains simple to use. The PPM Environment Manager was designed to support multiprocess programs, distributed programs, and programs offloaded in a network of machines. A prototype has been implemented on top of UNIX 4.3BSD.

1. Environments

Information that is useful to or necessary for a running program may not be determinable when the program is written, 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 address of a service
- the user's preferred font size for displayed output

One approach to providing this information at runtime is to bind a *value* to a *logical name* known to a program and to have the program call a routine to evaluate this binding. Following this approach, we would like to give each user fine-grained control over exactly the information provided to each program that he runs. We would like, at the same time, to make it easy for the user

This work was sponsored in part 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. Additional support was provided by IBM. The views and conclusions contained in this document are those of the author and should not be interpreted as representing official policies, either expressed or implied, of the Defense Research Projects Agency, of the US Goverment, or of IBM.

to provide this information, and for application writers to take advantage of the flexibility that it provides.

We call such a set of name-value bindings an *environment.* (We adopt the term from the UNIX operating system.) An environment can serve three functions:

Information repository -- An environment provides a uniform mechanism for supplying information to a program 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.

Point of indirection -- An 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. A user might bind the address of his preferred print server to an environment name. A printing program could evaluate this name to find the actual address to send the output to. This flexibility can be useful both in testing new services and in tailoring the services used by each user to an appropriate set of resources.

Means of communication -- If the bindings in the environment are *shared,* processes without shared address spaces 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, without requiring every application writer to include apecial provisions for such updates.

The user should not, of course, have to set a binding for each process to which it is relevant. He should be able to bind a value to a name once for evaluation by a set of processes. The user should have the ability, however, to set a binding to different values for different groups of processes. Some information, for example, may be machine specific, and should be seen only by processes on a particular machine. Some information is job specific, and should be seen only by processes in a particular job. Some information is application specific, and should be seen only

by processes running particular applications. The problem in constructing an environment suitable for distributed computing is to provide this flexibility, without sacrificing simplicity and convenience.

In this paper we describe the PPM Environment Manager, part of the *Personal Program Manager*, or *PPM* [13] (an earlier version of PPM is described in [4]). PPM provides a layer of distributed program management services between applications and the underlying operating system (or systems). These services are used by application programs including such tools as command interpreters supporting distributed computations and supporting login sessions where jobs are shifted between machines to balance their loads. The PPM Environment Manager provides a shared pool of name-value bindings for the processes of a particular user. The structure of the environment's name space and the semantics for binding evaluation provides the flexibility necessary for distributed programming, without making the environment difficult to use. Section 2 of this paper describes related work. Section 3 describes the design of the PPM Environment, its name space and values. Section 4 discusses a prototype implementation, and Section 5 concludes.

2. Related Work

The term "environment," we have said, is adopted from the UNIX operating system. The standard UNIX command interpreters, or *shells*, that is the Bourne Shell [3], the C shell [7], and the Korn Shell [8], support environments copied into each process's address space. These environments can be used as information repositories, but the simple copying semantics make it difficult to store information that is application specific or, in a distributed system, location specific. Because each process holds separate copies of bindings, they cannot be used for communication.

An *environment manager* has been proposed for MACH [14] which allows bindings to be shared. The environment manager is a separate server process that provides its clients with access to a pool of bindings. Each process may share access to its parent's environment, be given a copy of its parents environment, or be given an empty environment. The choice is the parent process's. This does not allow a child process to change a binding for its parent unless the parent allows for this possibility. There can be problems with this. Suppose that a shell process holds its current working directory as an environment binding. (This can be useful if the shell is running on a different machine from the one where it is starting jobs.) Rather than building a "change working directory" command into the shell, the working directory could be set in a shared environment by a program run by the shell. The shell, however, would have to treat the process running this command differently from other programs, which would have copies of the shell's environment to insulated them from this very change.

It is possible for a process to acquire access to more than one scope in the MACH scheme. 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 the MACH scheme are simple character 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 access to machine-specific or application-specific bindings. Distributed applications might wish to share access to global information. However, a binding for something like host machine name 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 copied across a machine boundary.

Some of the functionality desired for environments has been offered by *system name servers*, such as Grapevine [1] [12] and the Cambridge Name Server [10], but without providing the individual user control over the precise bindings seen be each process. System name servers

allow values to be bound to logical names, but the same bindings are generally seen by large sets of processes. This is useful, for example, for publicizing a server's address to a large number of client's. It is not as useful, however, for allowing multiple instances of a client each to contact a separate instance of a server. Such name servers are not useful for passing user specific information, such as a user's full name.

Cheriton and Mann's [5] name interpretation model for the V system has some of the features of an environment and some of a system name server. In the V system, at the lowest level, process identifiers are used as interprocess communication addresses. Because these identifiers are not known until runtime, the kernel provides a mechanism for binding thread identifiers to logical identifiers. A server, with a kernel call, can establish a binding between a logical identifier and its process identifier. Clients wishing to find a server address evaluate the binding with a kernel call. One of the parameters used in establishing 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, recognizing the importance in a distributed system of providing bindings that are specific to a particular site.

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 and are generally 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. By starting all interpretations at a per-user context server, this mechanism 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.

3. The Design of the PPM Environment

3.1. Basic Concepts

A binding is a linkage in some *context* between a *simple name* and a *value*. A *simple name* is a character string that is unambiguous within a context. Ideally, a simple name used in a program would be a straightforward and meaningful string such as *TIMEOUT* or *SERVER_ADDRESS*. The context in which a simple name is evaluated may be identified implicitly, as, for example, when there is only a single context for each user, or explicitly. Explicit context references may use a separate naming scheme from the scheme for simple names, or the two references may be integrated, as when the context is identified by a prefix added to the simple name (for example, */A/B/C* would refer to the simple name *C* in the context */A/B*).

Bindings must somehow be set for a process to evaluate. Some bindings might be set specifically for one process, but most will be provided as default settings to a number of processes. There are, in general, two ways that this could come about. The bindings might lie in a context *shared* among a number of processes, or bindings might be copied from an existing context when new contexts are created. The latter possibility we will call *inheritance*. It is also possible for these two schemes to be combined if a new context can inherit a *reference* to a binding in a different context. For now we will focus on pure inheritance.

Copying bindings each time a new context is created is not the only way of implementing inheritance. It can also be implemented by looking for a simple name in each of a chain of contexts when evaluating a binding. The choice between these possibilities is not important at an abstract level, but since the latter implementation is simpler to illustrate, we will assume that implementation for now.

Inheritance allows the *specialization* of bindings without name conflicts. This is illustrated in Figure 1. Processes *P1* and *P2* see different values when they evaluate the simple name *TIMEOUT*. If only a single shared context were available to the two processes, they could use different values only by using different names. Process *P1* inherits its value, 30 sec, from the superior context, while process *P2* sees the value 20 sec set in the inferior context. Inheritance allows changed bindings to be *cleared* rather than *reset*. By eliminating a context, all the bindings it contains are eliminated. With a single shared context, temporary changes have to be explicitly undone.

If a new context is created for each process, every process will be able to have its own specialized bindings. But from what context ought a process's context inherit the bulk of its bindings? One approach, taken by UNIX, is to copy bindings for a child process from the parent process's context. This approach, however, is limiting, because the value set for the parent may not be the appropriate default for the child. In Figure 1, for example, suppose that the superior context belongs to a process P0, running on a machine called Gemini, that is (logically) parent to

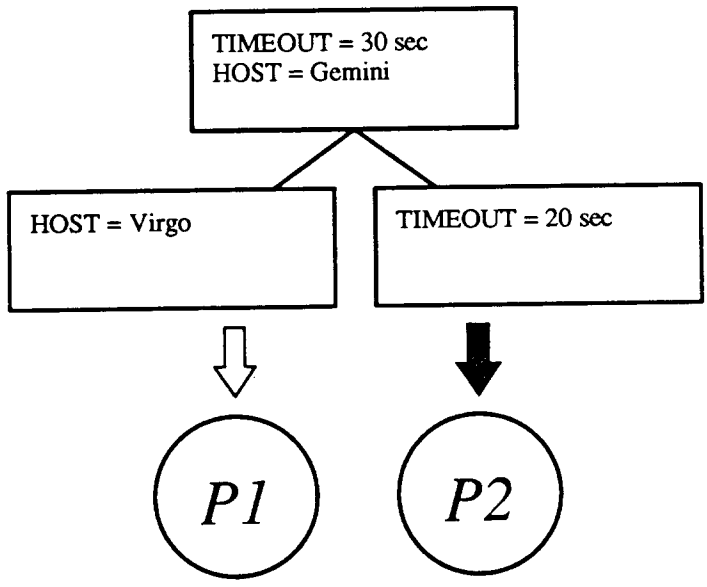


Figure 1 Inheritance allows bindings to be specialized without name conflicts.

both processes P1 and P2. Process P0's context contains the binding *HOST = Gemini*. Process P2, running on Gemini, can inherit this binding. If process P1 is running on the machine Virgo, rather than Gemini, the binding for *HOST* must be reset, rather than inherited. If P1 were to start a child process on Gemini, the binding would have to be explicitly reset again, even though its grandparent has the correct binding. This is unfortunate, because it requires that the parent process (or some other authority) know exactly which bindings need to be reset under various circumstances.

There are other bases for inheritance than parentage; bindings might be inherited on the basis of the machine on which they run, the application program they are running, the job to which they belong. Rather than choosing a single basis for inheritance it is possible for a context to have multiple 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. The need for multiple inheritance arises in other areas, such as in the design of object-oriented languages. Several of these languages, including Flavors [9], Trellis/Owl [11] and CommonLoops [2] have introduced multiple inheritance into their method (or function) inheritance scheme. 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. These same alternatives are available for resolving ambiguities in the inheritance of bindings in an environment.

3.2. Inheritance in PPM

The PPM Environment allows a process to inherit bindings through multiple lines of inheritance (see Figure 2). The inheritance line on the left might contain machine-independent bindings, and the line on the right machine-dependent bindings. Each process has a precedence ordering among its inheritance lines called an *evaluation path*. The contexts in Figure 2 are labeled in precedence order for process P1. PPM imposes a four-level hierarchical structure on a user's work. The user's processes are part of a *login session*. The login session contains *command sessions*, the command sessions contain *jobs* and the jobs contain processes. Each inheritance line contains four contexts, reflecting this structure. If two contexts within the same line of inheritance contain a binding for the same name, the process inherits the binding in the lower (more specific) context.

The four-level context hierarchy allows bindings to be set for four different scopes within a particular line of inheritance. The broadest scope, a login session, applies to all of the user's processes. The next level, a command session, applies to processes that are started from the same command interpreter; since a user may run jobs from several different windows, each with its own state. The third level is a job, and the fourth a process. Facilities outside the PPM Environment manager keep track of which process belongs to which job and which job belongs to which command session, as well as assigning to jobs and command sessions unique (within the login session) identifiers. The decision to limit jobs and command sessions to a single level each was made for the sake of simplicity. The alternative would be to allow a job to contain layers of subjobs and command sessions to contain layers of subcommand sessions. Restricting jobs to a simple structure has not proven to be a great limitation. Restricting command sessions to a single level has shown drawbacks. In particular, it makes it more difficult to allow to be undone the effects on the environment of some group of commands.

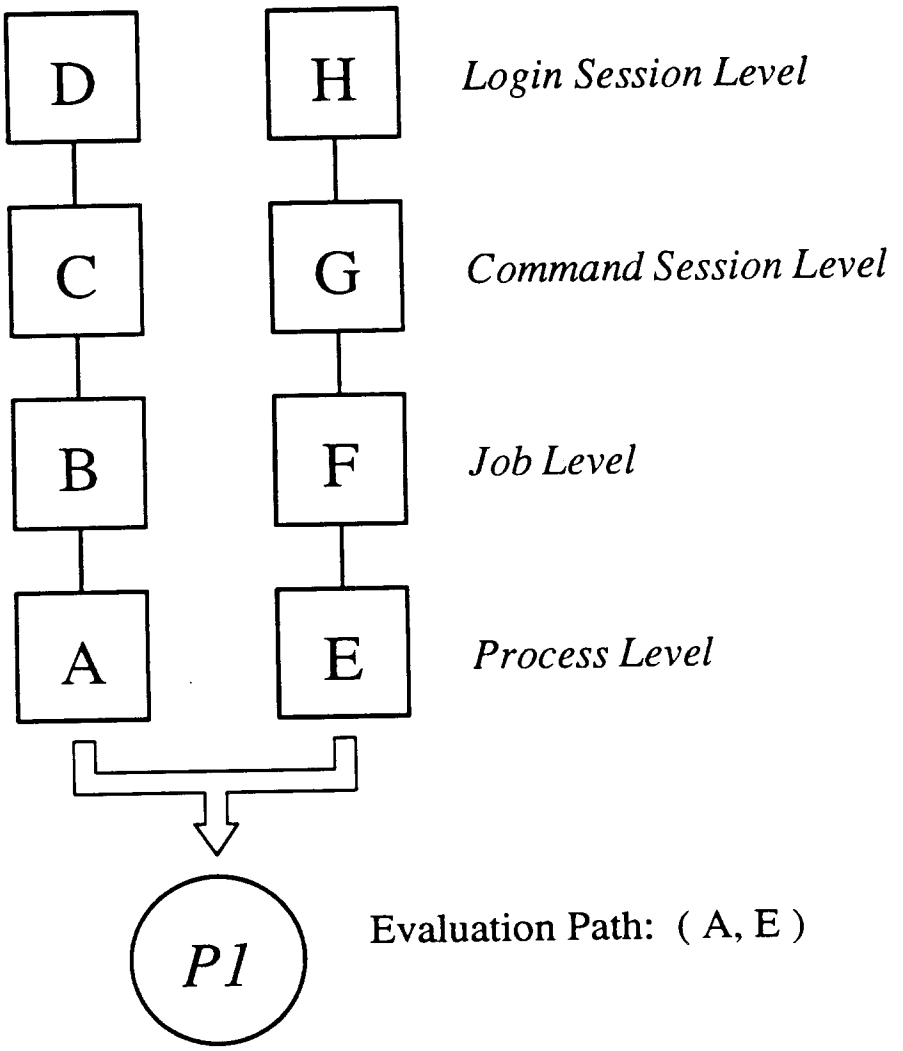


Figure 2 Multiple lines of inheritance

Different processes belonging to a single user will inherit bindings from overlapping sets of contexts. The extent of the overlap is determined by their evaluation paths and their closeness in the overall logical structure of the login session. For example, if processes P1 and P2 were components of he same job lying on different machines, they might inherit bindings from contexts in the pattern shown in Figure 3. Because they are on different machines, they will inherit independent sets of machine-dependent bindings. However, because they are in the same job, and hence the same command session and login session, they will inherit bindings from overlapping sets of machine-independent contexts.

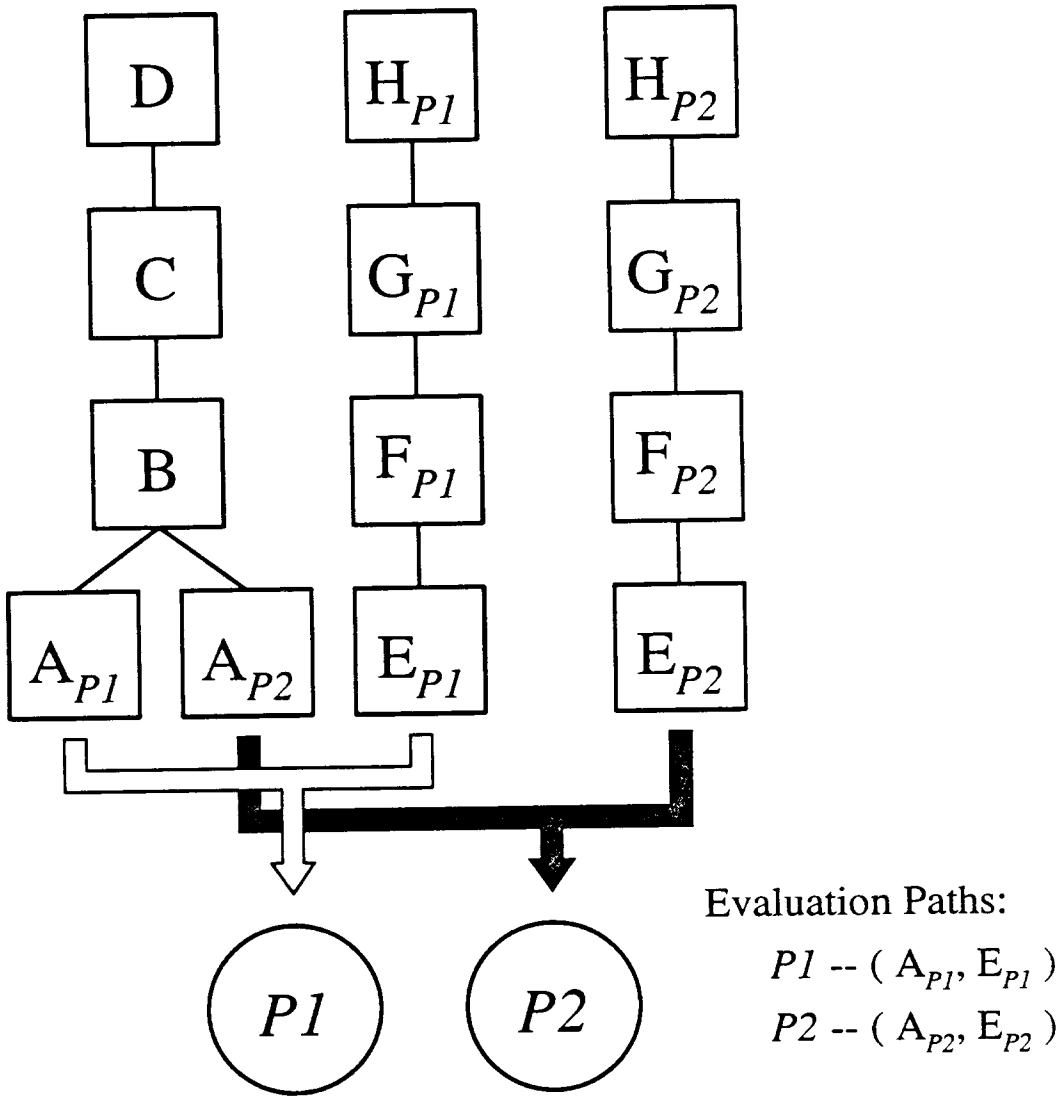


Figure 3 Shared lines of inheritance

Note that a line of inheritance for a process is a path upwards through a tree of contexts. A single root can be used to unite these trees, and so we will call them *inheritance subtrees*. We have experimented with three kinds of inheritance subtrees:

- *Default subtrees* contain bindings that can be applicable to all processes. Typically an evaluation path will include references to a high-precedence default tree and a low-precedence default tree.

- *Machine subtrees* contain bindings that are applicable to processes on a particular machine.
- *Application subtrees* contain bindings that are applicable to processes that are running a particular application.

We have treated inheritance by copying bindings into new contexts as equivalent with inheritance by looking up the binding in a series of contexts. In doing so, we have skipped over one point: What if a bindings value has changed since the process was created? Pure inheritance suggests that the old value of the binding should be returned, but this limits the utility of the environment for communication. We therefore make a basic distinction between the semantics of bindings inherited from the upper two levels and from the lower two levels. For bindings set in the upper levels, a process inherits the *value* the binding held when the process was created. For bindings set in the lower two levels, a process inherits a *reference* to the binding, rather than the value, and will see the binding's most recently assigned value. This has the effect of insulating running processes from changes at the upper levels, while allowing interprocess communication through bindings set at the lower levels.

3.3. Naming

Thus far, we have considered only the evaluation of bindings specified by simple names. These are looked up in contexts following the rules of inheritance. To set bindings in one of these contexts, however, it is necessary to be able to name the context explicitly. Contexts are given hierarchical names, resembling the file names in a hierarchical file system. An empty root context, named */PPMEnv*, is added to the context hierarchy above the login-session-level roots of the inheritance subtrees. A complete binding name includes a path from this root to a context and a simple name within that context. For example, to set a binding for *SimpleName* in a job context, we use a name of the form */PPMEnv/IS/CS/J/SimpleName*, where *S* identifies a inheritance subtree, and where *CS* and *J* are identifiers for the command session and job.

Figure 4 shows an example of the names used for the contexts in two lines of inheritance. The line on the left is in the high-precedence default subtree. This subtree's root context is called *AllH* by convention. The line on the right is in a machine subtree. It's root context's name, by convention, has the prefix *HOST* followed by a machine identifier, shown here as the number *123*. The other name components are unique identifiers.

The full names of contexts are unwieldy and difficult to construct in programs. For this reason, contexts are often referred to by *nicknames*. Nicknames are strings, delimited by angle brackets, inserted into binding names. Any string can be defined in the environment and then used as a nickname, but certain contexts have standard nicknames. Figure 4 shows nicknames for the contexts as well as their explicit names. For example, the process context in the high-precedence default subtree can be referred to as *<PROC>*. Context nicknames in the machine

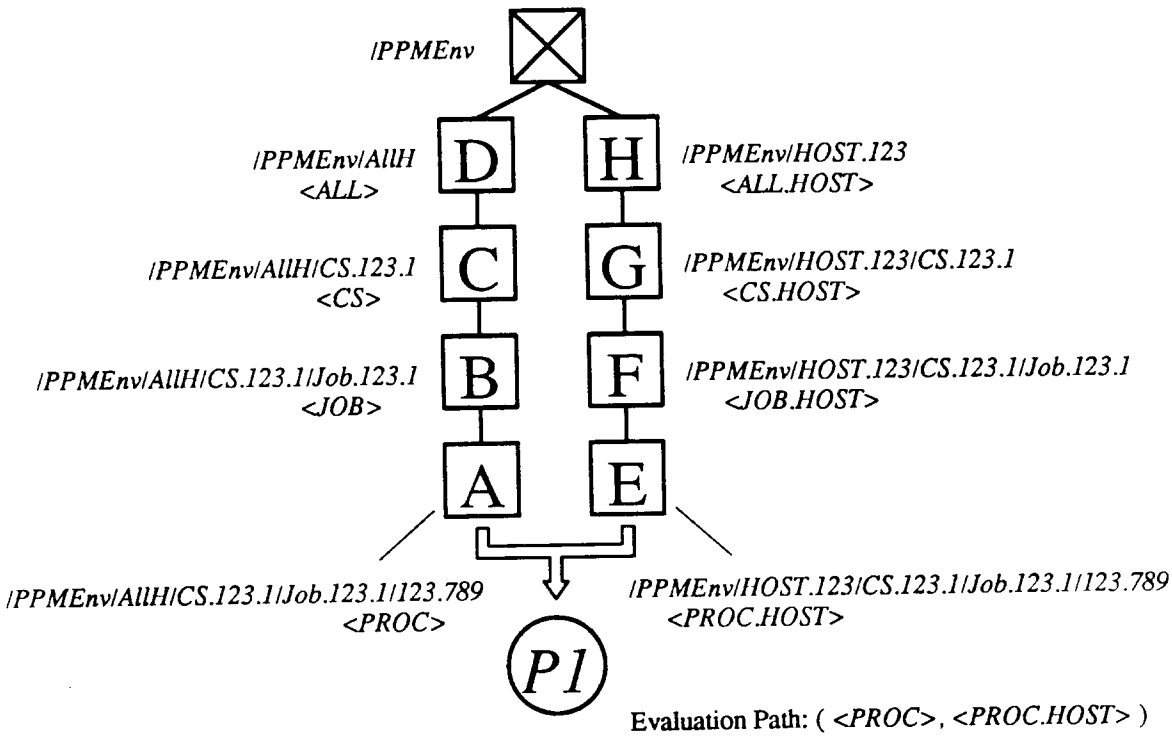


Figure 4 Names for contexts in the environment naming hierarchy.

subtree of the process's local machine have the suffix *HOST*. Contexts in the low-precedence default subtree have the suffix *LOW*, and contexts in an application subtree have the suffix *APPL*.

By using full names, any process can name (and hence set or a evaluate a binding in) any context. Most applications, however, evaluate bindings primarily through simple names and set bindings primarily using nicknames. Nicknames offer a form of relative naming. By using nicknames, all the contexts affecting a process can be referred to by fixed strings. This makes it much easier for programs to refer to bindings and to express indirections. (One use of this is to use nicknames to express a process's evaluation path, as shown in Figure 4.) It is more complex than the relative naming mechanisms used in hierarchical file systems because a process does not have a single "current context," instead having several.

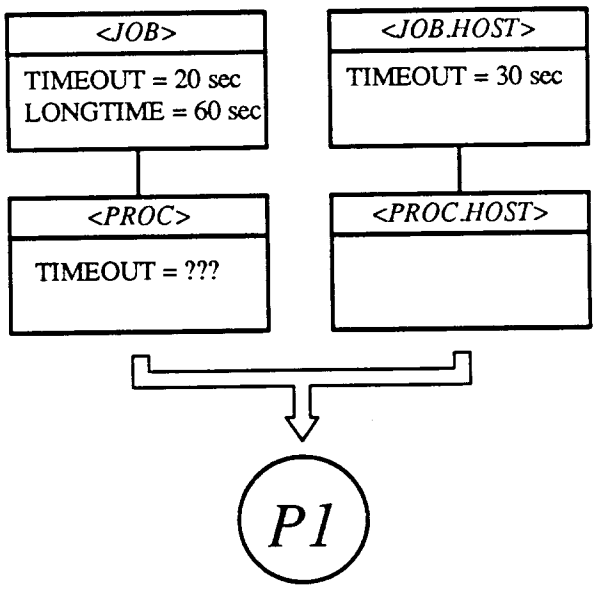
3.4. Bindings

The PPM Environment holds three types of bindings, *actual values*, *special values*, and *indirections*. An *actual value* is a set of uninterpreted bytes. A *special value* allows a specific binding to be unset (using the value *UNDEFINED*) or masked (using the value *ILLDEFINED*). When a binding is unset values may be inherited from higher contexts in the same line of inheritance. When a binding is masked, values in the same line of inheritance will not be inherited. This is used primarily for debugging. If a binding holds an *indirection*, that is a reference to another binding, this new binding will be evaluated, and its value returned.

Figure 5 shows an example of how these possibilities work. If the value of *TIMEOUT* is set to an actual value in the context *<PROC>*, that actual value will be seen by P1. If the special value UNDEFINED is used, the value 20 sec will be inherited from *<JOB>*. If ILLDEFINED is used, the value in *<JOB>* will be masked and the value 30 sec will be inherited from *<JOB.HOST>*. If *<PROC>/TIMEOUT* is set to be an indirection through the binding *<PROC.HOST>/TIMEOUT*, the latter will be evaluated, and its value, inherited from *<JOB.HOST>* will be returned. Finally, if *<PROC>/TIMEOUT* is set to an indirection through

LONGTIME, this name will be evaluated and its value returned. If *<PROC>/TIMEOUT* is set to an indirection, such as *@LONGTIME*, and a call is made to set the value of *<PROC>/TIMEOUT* to the value 45 sec, it is the value of *<JOB>/LONGTIME* that will change. This is usually, but not always, what is wanted. Therefore, we also need calls to set and evaluate bindings that treat special values and indirections the same as actual values.

Actual values are not interpreted by the PPM Environment. This means that the environment does not use type information in evaluating bindings and that ensuring correct type matching between binding settings and the expectations of their evaluators is left to the applications. In a network of heterogeneous machines, values may have to be stored in a system independent format, such as ASCII strings, or a more elaborate external data representation, such as CCITT



Value of *<PROC>/TIMEOUT*	Value of *TIMEOUT* for P1
45 sec	45 sec
UNDEFINED	20 sec
ILLDEFINED	30 sec
@<PROC.HOST>/TIMEOUT	30 sec
@LONGTIME	60 sec

Figure 5 The values bound to names can be *actual values*, *special values*, or *indirections*.

X.409 (DeSchon [6] presents a useful comparison of this and other data representation standards).

The first time that an environment subtree is accessed, the subtree is initialized. Initial values for bindings are read from some file or set of files. The files used is dependent upon the type of information held by the subtree. The current implementation allows default subtrees, machine subtrees, and application subtrees, each with its own naming conventions for initialization files. The initialization files may contain bindings to be placed in any subtree, not just the newly accessed one. This may be done to allow the initialization, say, to place in a machine subtree an indirection to a value placed in a default subtree.

4. Implementing the PPM Environment

The PPM Environment has been implemented on top of UNIX 4.3BSD as part of PPM, a set of services and tools for supporting distributed login sessions. Since one of the goals of PPM was to assist distributed programs in handling machine failure, the PPM Environment Manager was implemented as a distributed program holding replicated data. This approach ensures that when a machine fails, the bindings for processes that had been running on that machine can still be evaluated.

Clients of the PPM Environment Manager make calls to an interface library. The library routines set and evaluate bindings through a remote procedure call (RPC) to a local server processes. Setting a binding requires that all of the servers update their copy of the binding value. Lookups can then be performed entirely locally.

The time taken to perform operations on the environment is dominated by the time taken for RPCs, which can vary considerably with the system load. Our implementation relied upon UDP datagrams. The dominant factor here is the rescheduling delay incurred by the process making RPCs, which will be long in moderately loaded machines. Thus, the time taken for a local RPC varies from 10 to 30 ms on a lightly to moderately loaded VAX 785 and from 4 to 15 ms on a VAX 8600. The time taken for an RPC between such machines varies from 12 to 35 ms.

The time to evaluate a binding is essentially determined by the time to perform a local RPC. Setting a binding requires a two-phase protocol where the binding is first locked and then updated. The local server acts as coordinator for this protocol. Again, the major cost is the rescheduling delay that can occur. The calls to the remote participants can be overlapped, however, which minimizes the cost of adding new participants.

The cost of evaluating a binding can be reduced if the binding's value is stored within the address space of the client process. The PPM Environment Manager interface library includes a call to request that a specific binding be cached by the client. The library routines manage these cached values. When a cached binding is updated the local server process sends a message to the client. Caching reduces the cost of repeatedly evaluating a binding to essentially the cost of the call to the library call and the cost of checking for an update message.

The PPM Environment Manager provides operations to set and evaluate bindings normally, that is, respecting indirections and special values, and absolutely, that is treating both indirections and special values as if they were actual values. The Manager also provides a call that allows a client to find the most recently assigned value of a binding, without regard to rules of inheritance, and calls to lock or unlock a binding. The latter calls allow bindings (or groups of bindings) to be evaluated and reset atomically.

The current implementation of the PPM Environment Manager implements inheritance by lookup. Contexts are created only as they are needed, that is empty contexts need not be explicitly created. Contexts at the job and process level are created when bindings are set for at this level. These contexts are eliminated after a job has terminated. Thus, it is possible to evaluate bindings in the context of a process which has terminated if the job of which it is part has not terminated. Job termination is decided by other components of PPM, and the PPM Environment Manager is then informed. At the command session and login session levels, binding values are timestamped. Each binding value has a valid time, between the time that it was set and the

present or the time that it was reset. A value applies to any process that was created during its valid time. The time at which each process was created is available from other components of PPM. Values which are not applicable to any current process can be disposed of.

5. Conclusions

The PPM Environment Manager offers a shared pool of bindings to processes belonging to a particular user that may be scattered across a number of machines. The name space of the environment and the semantics of binding evaluation provide a rich structure that allows a wide variety of information to be passed through the environment without complicated programming. This environment has been quite important in implementing other portions of PPM, a system designed to support distributed applications by providing the user with a coherent distributed login session [13]. We have used the PPM Environment to implement a command interpreter able to run distributed jobs. The environment is used to hold items such as the default working directory for each machine, the machine on which to create the next process, the sources and destinations of each process's inputs and outputs. The environment is also used in configuring byte-stream I/O and in implementing a library for message I/O, which hides the details of addressing from the application program. We use the environment to hold dependency relations among the processes of a distributed job. These are evaluated and acted upon by components of PPM when a process fails, to simplify the handling of failure for the application program.

We hope that future versions of the PPM Environment Manager will integrate access to the environment with access to a system name server. With larger networks, a greater variety of resources, and more flexible displays, adaptability to the needs of particular users is of growing importance, and the diverse mechanisms for passing information should be replaced by a single, consistent approach. We believe that the flexibility to change bindings seen by individual jobs and processes is a useful aid to program development, that the environment scoping model is important in the development of distributed software, and that the use of the environment will

encourage the development of more flexible applications programs.

References

1. A. D. Birrell, R. Levin, R. M. Needham and M. D. Schroeder, "Grapevine: An exercise in distributed computing", *Communications of the ACM 25*, 4 (April 1982), 260-274.
2. 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*, October 1986, 17-29.
3. 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.
4. 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*, May 1986, 389-397.
5. 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*, May 1984, 290-297.
6. A. L. DeSchon, "A survey of data representation standards", RFC 971, USC ISI, January 1986.
7. 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.
8. D. Korn, "Introduction to KSH", *1983 Summer USENIX Conference Proceedings*, July 1983, 191-202.
9. D. A. Moon, "Object-oriented programming with Flavors", *Proc. of Object-Oriented Programming Systems, Languages and Applications '86*, October 1986, 1-8.
10. R. M. Needham and A. J. Herbert, *The Cambridge Distributed Computing System*, Addison-Wesley, Reading, Massachusetts, 1982.
11. 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*, October 1986, 9-16.
12. M. D. Schroeder, A. D. Birrell and R. M. Needham, "Experience with Grapevine: The growth of a distributed system", *ACM Transactions on Computer Systems 2*, 1 (February 1984), 3-23.
13. S. Sechrest, "A client-server shell architecture for distributed programming", Technical Report No. UCB/CSD 88/457, Computer Science Division, University of California at Berkeley, October 1988.
14. M. R. Thompson, "MACH environment manager", Internal memo of the Department of Computer Science, Carnegie-Mellon University, February 1987.