7 How PVM Works

In this chapter we describe the implementation of the PVM software and the reasons behind the basic design decisions. The most important goals for PVM 3 are fault tolerance, scalability, heterogeneity, and portability. PVM is able to withstand host and network failures. It doesn't automatically recover an application after a crash, but it does provide polling and notification primitives to allow fault-tolerant applications to be built. The virtual machine is dynamically reconfigurable. This property goes hand in hand with fault tolerance: an application may need to acquire more resources in order to continue running once a host has failed. Management is as decentralized and localized as possible, so virtual machines should be able to scale to hundreds of hosts and run thousands of tasks. PVM can connect computers of different types in a single session. It runs with minimal modification on any flavor of Unix or an operating system with comparable facilities (multitasking, networkable). The programming interface is simple but complete, and any user can install the package without special privileges.

To allow PVM to be highly portable, we avoid the use of operating system and language features that would be be hard to retrofit if unavailable, such as multithreaded processes and asynchronous I/O. These exist in many versions of Unix, but they vary enough from product to product that different versions of PVM might need to be maintained. The generic port is kept as simple as possible, though PVM can always be optimized for any particular machine.

We assume that *sockets* are used for interprocess communication and that each host in a virtual machine group can connect directly to every other host via TCP [13] and UDP [14] protocols. The requirement of full IP connectivity could be removed by specifying message routes and using the pvmds to forward messages. Some multiprocessor machines don't make sockets available on the processing nodes, but do have them on the front-end (where the pvmd runs).

7.1 Components

7.1.1 Task Identifiers

PVM uses a *task identifier* (TID) to address pvmds, tasks, and groups of tasks within a virtual machine. The TID contains four fields, as shown in Figure 7.1. Since the TID is used so heavily, it is made to fit into the largest integer data type (32 bits) available on a wide range of machines.

The fields S, G, and H have global meaning: each pvmd of a virtual machine interprets them in the same way. The H field contains a host number relative to the virtual machine. As it starts up, each pvmd is configured with a unique host number and therefore "owns"

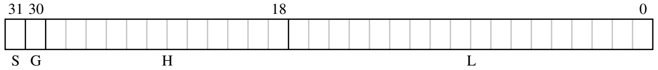

Figure 7.1
Generic task id

part of the TID address space. The maximum number of hosts in a virtual machine is limited to $2^H - 1$ (4095). The mapping between host numbers and hosts is known to each pvmd, synchronized by a global *host table*. Host number zero is used, depending on context, to refer to the local pvmd or a *shadow pvmd*, called *pvmd'* (Section 7.3.7).

The S bit is used to address pvmds, with the H field set to the host number and the L field cleared. This bit is a historical leftover and causes slightly schizoid naming; sometimes pvmds are addressed with the S bit cleared. It should someday be reclaimed to make the H or L space larger.

Each pvmd is allowed to assign private meaning to the L field (with the H field set to its own host number), except that "all bits cleared" is reserved to mean the pvmd itself. The L field is 18 bits wide, so up to $2^{18} - 1$ tasks can exist concurrently on each host. In the generic Unix port, L values are assigned by a counter, and the pvmd maintains a map between L values and Unix process id's. Use of the L field in multiprocessor ports is described in Section 7.10.

The G bit is set to form multicast addresses (GIDs), which refer to groups of tasks. Multicasting is described in Section 7.6.4.

The design of the TID enables the implementation to meet the design goals. Tasks can be assigned TIDs by their local pvmds without off-host communication. Messages can be routed from anywhere in a virtual machine to anywhere else, by hierarchical naming. Portability is enhanced because the L field can be redefined. Finally, space is reserved for error codes. When a function can return a vector of TIDs mixed with error codes, it is useful if the error codes don't correspond to legal TIDs. The TID space is divided up as follows:

Use	S	G	H	L
Task identifier	0	0	$1..H_{max}$	$1..L_{max}$
Pvmd identifier	1	0	$1..H_{max}$	0
Local pvmd (from task)	1	0	0	0
Pvmd' from master pvmd	1	0	0	0
Multicast address	0	1	$1..H_{max}$	$0..L_{max}$
Error code	1	1	(small neg. number)	

Naturally, TIDs are intended to be opaque to the application, and the programmer should not attempt to predict their values or modify them without using functions supplied in the programming library. More symbolic naming can be obtained by using a name server library layered on top of the raw PVM calls, if the convenience is deemed worth the cost of name lookup.

7.1.2 Architecture Classes

PVM assigns an *architecture name* to each kind of machine on which it runs, to distinguish between machines that run different executables, because of hardware or operating system differences. Many standard names are defined, and others can be added.

Sometimes machines with incompatible executables use the same binary data representation. PVM takes advantage of this to avoid data conversion. Architecture names are mapped to *data encoding* numbers, and the encoding numbers are used to determine when it is necessary to convert.

7.1.3 Message Model

PVM daemons and tasks can compose and send messages of arbitrary lengths containing typed data. The data can be converted using XDR [16] when passing between hosts with incompatible data formats. Messages are tagged at send time with a user-defined integer code and can be selected for receipt by source address or tag.

The sender of a message does not wait for an acknowledgment from the receiver, but continues as soon as the message has been handed to the network and the message buffer can be safely deleted or reused. Messages are buffered at the receiving end until received. PVM reliably delivers messages, provided the destination exists. Message order from each sender to each receiver in the system is preserved; if one entity sends several messages to another, they will be received in the same order.

Both blocking and nonblocking receive primitives are provided, so a task can wait for a message without (necessarily) consuming processor time by polling for it. Or, it can poll for a message without hanging. A receive with timeout is also provided, which returns after a specified time if no message has arrived.

No acknowledgments are used between sender and receiver. Messages are reliably delivered and buffered by the system. If we ignore fault recovery, then either an application will run to completion or, if some component goes down, it won't. In order to provide fault recovery, a task (T_A) must be prepared for another task (T_B, from which it wants a message) to crash, and must be able to take corrective action. For example, it might reschedule its request to a different server, or even start a new server. From the viewpoint of T_A, it doesn't matter specifically when T_B crashes relative to messages sent from T_A.

While waiting for T_B, T_A will receive either a message from T_B or notification that T_B has crashed. For the purposes of flow control, a fully blocking send can easily be built using the semi-synchronous send primitive.

7.1.4 Asynchronous Notification

PVM provides *notification* messages as a means to implement fault recovery in an application. A task can request that the system send a message on one of the following three events:

Type	Meaning
`PvmTaskExit`	Task exits or crashes
`PvmHostDelete`	Host is deleted or crashes
`PvmHostAdd`	New hosts are added to the VM

Notify requests are stored in the pvmds, attached to objects they monitor. Requests for remote events (occurring on a different host than the requester) are kept on both hosts. The remote pvmd sends the message if the event occurs, while the local one sends the message if the remote host goes down. The assumption is that a local pvmd can be trusted; if it goes down, tasks running under it won't be able to do anything, so they don't need to be notified.

7.1.5 PVM Daemon and Programming Library

PVM Daemon One pvmd runs on each host of a virtual machine. Pvmds owned by (running as) one user do not interact with those owned by others, in order to reduce security risk, and minimize the impact of one PVM user on another.

The pvmd serves as a message router and controller. It provides a point of contact, authentication, process control, and fault detection. An idle pvmd occasionally checks that its peers are still running. Even if application programs crash, pvmds continue to run, to aid in debugging.

The first pvmd (started by hand) is designated the *master*, while the others (started by the master) are called *slaves*. During normal operation, all are considered equal. But only the master can start new slaves and add them to the configuration. Reconfiguration requests originating on a slave host are forwarded to the master. Likewise, only the master can forcibly delete hosts from the machine.

Programming Library The libpvm library allows a task to interface with the pvmd and other tasks. It contains functions for packing (composing) and unpacking messages, and functions to perform PVM *syscalls* by using the message functions to send service requests to the pvmd. It is made as small and simple as possible. Since it shares an address space with unknown, possibly buggy, code, it can be broken or subverted. Minimal sanity-checking of parameters is performed, leaving further authentication to the pvmd.

The top level of the libpvm library, including most of the programming interface functions, is written in a machine-independent style. The bottom level is kept separate and can be modified or replaced with a new machine-specific file when porting PVM to a new environment.

7.2 Messages

7.2.1 Fragments and Databufs

The pvmd and libpvm manage message buffers, which potentially hold large amounts of dynamic data. Buffers need to be shared efficiently, for example, to attach a multicast message to several send queues (see Section 7.6.4). To avoid copying, all pointers are to a single instance of the data (a *databuf*), which is refcounted by allocating a few extra bytes for an integer at the head of the data. A pointer to the data itself is passed around, and routines subtract from it to access the refcount or free the block. When the refcount of a databuf decrements to zero, it is freed.

PVM messages are composed without declaring a maximum length ahead of time. The pack functions allocate memory in steps, using databufs to store the data, and frag descriptors to chain the databufs together.

A frag descriptor `struct frag` holds a pointer (`fr_dat`) to a block of data and its length (`fr_len`). It also keeps a pointer (`fr_buf`) to the databuf and its total length (`fr_max`); these reserve space to prepend or append data. Frags can also reference static (non-databuf) data. A frag has link pointers so it can be chained into a list. Each frag keeps a count of references to it; when the refcount decrements to zero, the frag is freed and the underlying databuf refcount is decremented. In the case where a frag descriptor is the head of a list, its refcount applies to the entire list. When it reaches zero, every frag in the list is freed. Figure 7.2 shows a list of fragments storing a message.

7.2.2 Messages in Libpvm

Libpvm provides functions to pack all of the primitive data types into a message, in one of several encoding formats. There are five sets of encoders and decoders. Each message buffer has a set associated with it. When creating a new message, the encoder set is determined by the format parameter to `pvm_mkbuf()`. When receiving a message, the decoders are determined by the encoding field of the message header. The two

most commonly used ones pack data in *raw* (host native) and *default* (XDR) formats. *Inplace* encoders pack descriptors of the data (the frags point to static data), so the message is sent without copying the data to a buffer. There are no inplace decoders. *Foo* encoders use a machine-independent format that is simpler than XDR; these encoders are used when communicating with the pvmd. *Alien* decoders are installed when a received message can't be unpacked because its encoding doesn't match the data format of the host. A message in an alien data format can be held or forwarded, but any attempt to read data from it results in an error.

Figure 7.2 shows libpvm message management. To allow the PVM programmer to handle message buffers, they are labeled with integer message id's (MIDs), which are simply indices into the message heap. When a message buffer is freed, its MID is recycled. The heap starts out small and is extended if it becomes full. Generally, only a few messages exist at any time, unless an application explicitly stores them.

A vector of functions for encoding/decoding primitive types (`struct encvec`) is initialized when a message buffer is created. To pack a long integer, the generic pack function `pvm_pklong()` calls `(message_heap[mid].ub_codef->enc_long)()` of the buffer. Encoder vectors were used for speed (as opposed to having a case switch in each pack function). One drawback is that every encoder for every format is touched (by naming it in the code), so the linker must include all the functions in every executable, even when they're not used.

7.2.3 Messages in the Pvmd

By comparison with libpvm, message packing in the pvmd is very simple. Messages are handled using `struct mesg` (shown in Figure 7.3). There are encoders for signed and unsigned integers and strings, which use in the libpvm *foo* format. Integers occupy four bytes each, with bytes in network order (bits 31..24 followed by bits 23..16, ...). Byte strings are packed as an integer length (including the terminating null for ASCII strings), followed by the data and zero to three null bytes to round the total length to a multiple of four.

7.2.4 Pvmd Entry Points

Messages for the pvmd are reassembled from packets in `loclinpkt()` if from a local task, or in `netinpkt()` if from another pvmd or foreign task. Reassembled messages are passed to one of three *entry points*:

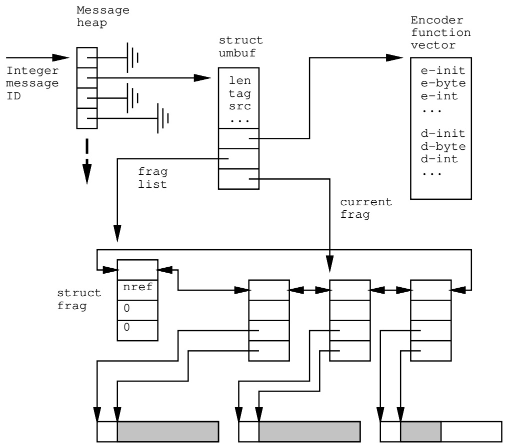

Figure 7.2
Message storage in libpvm

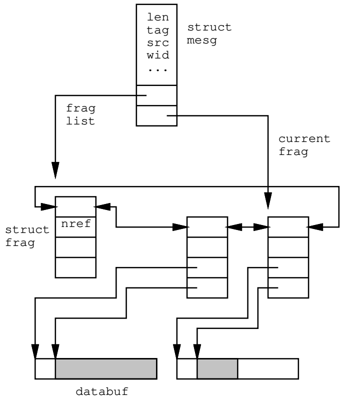

Figure 7.3
Message storage in pvmd

Function	Messages From
`loclentry()`	Local tasks
`netentry()`	Remote pvmds
`schentry()`	Local or remote special tasks (Resource manager, Hoster, Tasker)

If the message tag and contents are valid, a new thread of action is started to handle the request. Invalid messages are discarded.

7.2.5 Control Messages

Control messages are sent to a task like regular messages, but have tags in a reserved space (between `TC_FIRST` and `TC_LAST`). Normally, when a task downloads a message, it queues it for receipt by the program. Control messages are instead passed to `pvmmctl()` and then discarded. Like the entry points in the pvmd, `pvmmctl()` is an entry point in the task, causing it to take some asynchronous action. The main difference is that control messages can't be used to get the task's attention, since it must be in `mxfer()`, sending or receiving, in order to get them.

The following control message tags are defined. The first three are used by the direct routing mechanism (discussed in Section 7.6.3). `TC_OUTPUT` is used to implement `pvm_catchout()` (Section 7.7.2). User-definable control messages may be added in the future as a way of implementing PVM signal handlers.

Tag	Meaning
`TC_CONREQ`	Connection request
`TC_CONACK`	Connection ack
`TC_TASKEXIT`	Task exited/doesn't exist
`TC_NOOP`	Do nothing
`TC_OUTPUT`	Claim child stdout data
`TC_SETTMASK`	Change task trace mask

7.3 PVM Daemon

7.3.1 Startup

At startup, a pvmd configures itself as a master or slave, depending on its command line arguments. It creates and binds sockets to talk to tasks and other pvmds, and it opens an error log file `/tmp/pvml.`*uid*. A master pvmd reads the host file if supplied; otherwise it uses default parameters. A slave pvmd gets its parameters from the master pvmd via the command line and configuration messages.

After configuration, the pvmd enters a loop in function `work()`. At the core of the work loop is a call to `select()` that probes all sources of input for the pvmd (local tasks and the network). Packets are received and routed to send queues. Messages to the pvmd are reassembled and passed to the entry points.

7.3.2 Shutdown

A pvmd shuts down when it is deleted from the virtual machine, killed (signaled), loses contact with the master pvmd, or breaks (e.g., with a bus error). When a pvmd shuts down, it takes two final actions. First, it kills any tasks running under it, with signal `SIGTERM`. Second, it sends a final shutdown message (Section 7.5.2) to every other pvmd in its host table. The other pvmds would eventually discover the missing one by timing out trying to communicate with it, but the shutdown message speeds the process.

7.3.3 Host Table and Machine Configuration

A host table describes the configuration of a virtual machine. It lists the name, address and communication state for each host. Figure 7.4 shows how a host table is built from `struct htab` and `struct hostd` structures.

Host tables are issued by the master pvmd and kept synchronized across the virtual machine. The delete operation is simple: On receiving a `DM_HTDEL` message from the master, a pvmd calls `hostfailentry()` for each host listed in the message, as though the deleted pvmds crashed. Each pvmd can autonomously delete hosts from its own table on finding them unreachable (by timing out during communication). The add operation is done with a three-phase commit, in order to guarantee global availability of new hosts synchronously with completion of the add-host request. This is described in Section 7.3.8.

Each host descriptor has a refcount so it can be shared by multiple host tables. As the configuration of the machine changes, the host descriptors (except those added and deleted, of course) propagate from one host table to the next. This propagation is necessary because they hold various state information.

Host tables also serve other uses: They allow the pvmd to manipulate host sets, for example, when picking candidate hosts on which to spawn a task. Also, the advisory host file supplied to the master pvmd is parsed and stored in a host table.

Host File If the master pvmd is started with a host file, it parses the file into a host table, `filehosts`. If some hosts in the file are to be started automatically, the master sends a `DM_ADD` message to itself. The slave hosts are started just as though they had been added dynamically (Section 7.3.8).

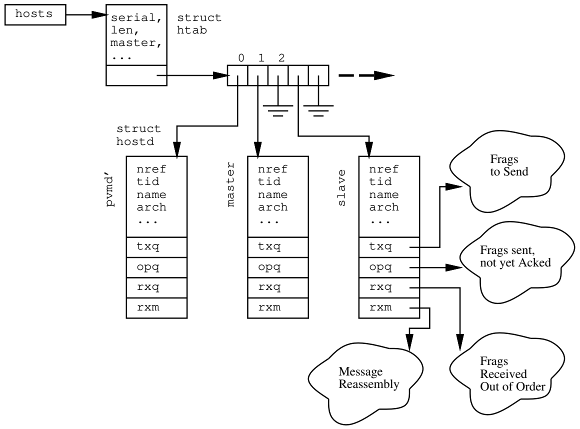

Figure 7.4
Host table

7.3.4 Tasks

Each pvmd maintains a list of all tasks under its management (Figure 7.5). Every task, regardless of state, is a member of a threaded list, sorted by task id. Most tasks are also in a second list, sorted by process id. The head of both lists is **`locltasks`**.

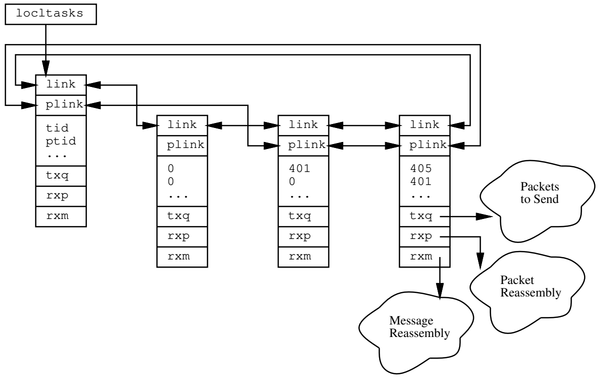

Figure 7.5
Task table

PVM provides a simple debugging system described in Section 7.7.4. More complex debuggers can be built by using a special type of task called a *tasker*, introduced in version 3.3. A tasker starts (*execs*, and is the parent of) other tasks. In general, a debugger is a process that controls the execution of other processes - can read and write their memories and start and stop instruction counters. On many species of Unix, a debugger must be the direct parent of any processes it controls. This is becoming less common with growing availability of the attachable *ptrace* interface.

The function of the tasker interface overlaps with the simple debugger starter, but is fundamentally different for two reasons: First, all tasks running under a pvmd (during the life of the tasker) may be children of a single tasker process. With **`PvmTaskDebug`**, a new debugger is necessarily started for each task. Second, the tasker cannot be enabled or disabled by spawn flags, so it is always in control, though this is not an important difference.

If a tasker is registered (using `pvm_reg_tasker()`) with a pvmd when a `DM_EXEC` message is received to start new tasks, the pvmd sends a `SM_STTASK` message to the tasker instead of calling `execv()`. No *SM_STTASKACK* message is required; closure comes from the task reconnecting to the pvmd as usual. The pvmd doesn't get `SIGCHLD` signals when a tasker is in use, because it's not the parent process of tasks, so the tasker must send notification of exited tasks to the pvmd in a `SM_TASKX` message.

7.3.5 Wait Contexts

The pvmd uses a wait context (waitc) to hold state when a thread of operation must be interrupted. The pvmd is not truly multithreaded but performs operations concurrently. For example, when a pvmd gets a syscall from a task and must interact with another pvmd, it doesn't block while waiting for the other pvmd to respond. It saves state in a waitc and returns immediately to the `work()` loop. When the reply arrives, the pvmd uses the information stashed in the waitc to complete the syscall and reply to the task. Waitcs are serial numbered, and the number is sent in the message header along with the request and returned with the reply.

For many operations, the TIDs and *kind* of wait are the only information saved. The `struct waitc` includes a few extra fields to handle most of the remaining cases, and a pointer, `wa_spec`, to a block of extra data for special cases—the spawn and host startup operations, which need to save `struct waitc_spawn` and `struct waitc_add`.

Sometimes more than one phase of waiting is necessary—in series, parallel, or nested. In the parallel case, a separate waitc is created for each foreign host. The waitcs are *peered* (linked in a list) together to indicate they pertain to the same operation. If a waitc has no peers, its peer links point to itself. Usually, peered waitcs share data, for example, `wa_spec`. All existing parallel operations are conjunctions; a peer group is finished when every waitc in the group is finished. As replies arrive, finished waitcs are collapsed out of the list and deleted. When the finished waitc is the only one left, the operation is complete. Figure 7.6 shows single and peered waitcs stored in `waitlist` (the list of all active waitcs).

When a host fails or a task exits, the pvmd searches `waitlist` for any blocked on this TID and terminates those operations. Waitcs from the dead host or task blocked on something else are not deleted; instead, their `wa_tid` fields are zeroed. This approach prevents the wait id's from being recycled while replies are still pending. Once the defunct waitcs are satisfied, they are silently discarded.

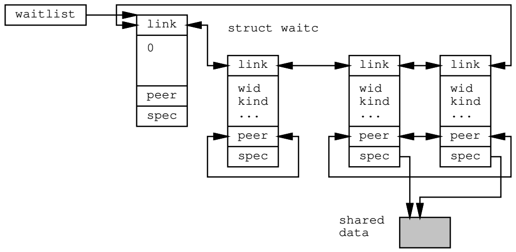

Figure 7.6
Wait context list

7.3.6 Fault Detection and Recovery

Fault detection originates in the pvmd-pvmd protocol (Section 7.5.2). When the pvmd times out while communicating with another, it calls `hostfailentry()`, which scans `waitlist` and terminates any operations waiting on the down host.

A pvmd can recover from the loss of any foreign pvmd except the master. If a slave loses the master, the slave shuts itself down. This algorithm ensures that the virtual machine doesn't become partitioned and run as two partial machines. It does, however, decrease fault tolerance of the virtual machine because the master must never crash. There is currently no way for the master to hand off its status to another pvmd, so it always remains part of the configuration. (This is an improvement over PVM 2, in which the failure of any pvmd would shut down the entire system.)

7.3.7 Pvmd'

The shadow pvmd (pvmd') runs on the master host and is used by the master to start new slave pvmds. Any of several steps in the startup process (for example, starting a shell on the remote machine) can block for seconds or minutes (or hang), and the master pvmd must be able to respond to other messages during this time. It's messy to save all the state involved, so a completely separate process is used.

The pvmd' has host number 0 and communicates with the master through the normal pvmd-pvmd interface, though it never talks to tasks or other pvmds. The normal host failure detection mechanism is used to recover in the event the pvmd' fails. The startup operation has a wait context in the master pvmd. If the pvmd' breaks, the master catches a `SIGCHLD` from it and calls `hostfailentry()`, which cleans up.

7.3.8 Starting Slave Pvmds

Getting a slave pvmd started is a messy task with no good solution. The goal is to get a process running on the new host, with enough identity to let it be fully configured and added as a peer.

Ideally, the mechanism used should be widely available, secure, and fast, while leaving the system easy to install. We'd like to avoid having to type passwords all the time, but don't want to put them in a file from where they can be stolen. No one system meets all of these criteria. Using `inetd` or connecting to an already-running pvmd or pvmd server at a reserved port would allow fast, reliable startup, but would require that a system administrator install PVM on each host. Starting the pvmd via `rlogin` or `telnet` with a *chat* script would allow access even to IP-connected hosts behind firewall machines and would require no special privilege to install; the main drawbacks are speed and the effort needed to get the chat program working reliably.

Two widely available systems are `rsh` and `rexec()`; we use both to cover the cases where a password does and does not need to be typed. A manual startup option allows the user to take the place of a chat program, starting the pvmd by hand and typing in the configuration. `rsh` is a privileged program that can be used to run commands on another host without a password, provided the destination host can be made to trust the source host. This can be done either by making it equivalent (requires a system administrator) or by creating a `.rhosts` file on the destination host (this isn't a great idea). The alternative, `rexec()`, is a function compiled into the pvmd. Unlike `rsh`, which doesn't take a password, `rexec()` requires the user to supply one at run time, either by typing it in or by placing it in a `.netrc` file (this is a really bad idea).

Figure 7.7 shows a host being added to the machine. A task calls `pvm_addhosts()` to send a request to its pvmd, which in turn sends a `DM_ADD` message to the master (possibly itself). The master pvmd creates a new host table entry for each host requested, looks up the IP addresses, and sets the options from host file entries or defaults. The host descriptors are kept in a `waitc_add` structure (attached to a wait context) and not yet added to the host table. The master forks the pvmd' to do the dirty work, passing it a list of hosts and commands to execute (an `SM_STHOST` message). The pvmd' uses `rsh`, `rexec()` or manual startup to start each pvmd, pass it parameters, and get a line of configuration data back. The configuration dialog between pvmd' and a new slave is as follows:

pvmd' → slave:	(exec) `$PVM_ROOT/lib/pvmd -s -d8 -nhonk 1 80a9ca95:0f5a 4096 3 80a95c43:0000`
slave → pvmd':	`ddpro<2312> arch<ALPHA> ip<80a95c43:0b3f> mtu<4096>`
pvmd' → slave:	`EOF`

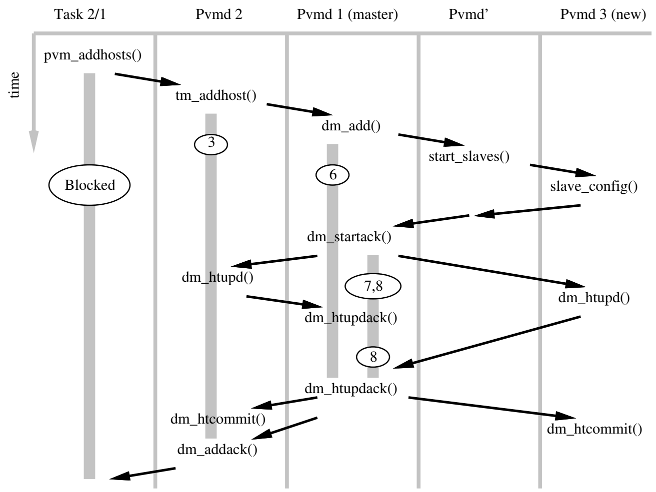

Figure 7.7
Timeline of addhost operation

The addresses of the master and slave pvmds are passed on the command line. The slave writes its configuration on standard output, then waits for an EOF from the pvmd' and disconnects. It runs in probationary status (`runstate = PVMDSTARTUP`) until it receives the rest of its configuration from the master pvmd. If it isn't configured within five minutes (parameter `DDBAILTIME`), it assumes there is some problem with the master and quits. The protocol revision (`DDPROTOCOL`) of the slave pvmd must match that of the master. This number is incremented whenever a change in the protocol makes it incompatible with the previous version. When several hosts are added at once, startup is done in parallel. The pvmd' sends the data (or errors) in a `DM_STARTACK` message to the master pvmd, which completes the host descriptors held in the wait context.

If a special task called a *hoster* is registered with the master pvmd when it receives the `DM_ADD` message, the pvmd' is not used. Instead, the `SM_STHOST` message is sent to the hoster, which starts the remote processes as described above using any mechanism it wants, then sends a `SM_STHOSTACK` message (same format as `DM_STARTACK`) back to the master pvmd. Thus, the method of starting slave pvmds is dynamically replaceable, but the hoster does not have to understand the configuration protocol. If the hoster task fails during an add operation, the pvmd uses the wait context to recover. It assumes none of the slaves were started and sends a `DM_ADDACK` message indicating a system error.

After the slaves are started, the master sends each a `DM_SLCONF` message to set parameters not included in the startup protocol. It then broadcasts a `DM_HTUPD` message to all new and existing slaves. Upon receiving this message, each slave knows the configuration of the new virtual machine. The master waits for an acknowledging `DM_HTUPDACK` message from every slave, then broadcasts an `HT_COMMIT` message, shifting all to the new host table. Two phases are needed so that new hosts are not advertised (e.g., by `pvm_config()`) until all pvmds know the new configuration. Finally, the master sends a `DM_ADDACK` reply to the original request, giving the new host id's.

Note: Recent experience suggests it would be cleaner to manage the pvmd' through the task interface instead of the host interface. This approach would allow multiple starters to run at once (parallel startup is implemented explicitly in a single pvmd' process).

7.3.9 Resource Manager

A *resource manager* (RM) is a PVM task responsible for making task and host scheduling (placement) decisions. The resource manager interface was introduced in version 3.3. The simple schedulers embedded in the pvmd handle many common conditions, but require

the user to explicitly place program components in order to get the maximum efficiency. Using knowledge not available to the pvmds, such as host load averages, a RM can make more informed decisions automatically. For example, when spawning a task, it could pick the host in order to balance the computing load. Or, when reconfiguring the virtual machine, the RM could interact with an external queuing system to allocate a new host.

The number of RMs registered can vary from one for an entire virtual machine to one per pvmd. The RM running on the master host (where the master pvmd runs) manages any slave pvmds that don't have their own RMs. A task connecting anonymously to a virtual machine is assigned the default RM of the pvmd to which it connects. A task spawned from within the system inherits the RM of its parent task.

If a task has a RM assigned to it, service requests from the task to its pvmd are routed to the RM instead. Messages from the following libpvm functions are intercepted:

Libpvm function	Default Message	RM Message
`pvm_addhost()`	TM_ADDHOST	SM_ADDHOST
`pvm_delhost()`	TM_DELHOST	SM_DELHOST
`pvm_spawn()`	TM_SPAWN	SM_SPAWN

Queries also go to the RM, since it presumably knows more about the state of the virtual machine:

Libpvm function	Default Message	RM Message
`pvm_config()`	TM_CONFIG	SM_CONFIG
`pvm_notify()`	TM_NOTIFY	SM_NOTIFY
`pvm_task()`	TM_TASK	SM_TASK

The call to register a task as a RM (`pvm_reg_rm()`) is also redirected if RM is already running. In this way the existing RM learns of the new one, and can grant or refuse the request to register.

Using messages `SM_EXEC` and `SM_ADD`, the RM can directly command the pvmds to start tasks or reconfigure the virtual machine. On receiving acknowledgement for the commands, it replies to the client task. The RM is free to interpret service request parameters in any way it wishes. For example, the architecture class given to `pvm_spawn()` could be used to distinguish hosts by memory size or CPU speed.

7.4 Libpvm Library

7.4.1 Language Support

Libpvm is written in C and directly supports C and C++ applications. The Fortran library, `libfpvm3.a` (also written in C), is a set of *wrapper* functions that conform to the Fortran calling conventions. The Fortran/C linking requirements are portably met by preprocessing the C source code for the Fortran library with `m4` before compilation.

7.4.2 Connecting to the Pvmd

On the first call to a libpvm function, `pvm_beatask()` is called to initialize the library state and connect the task to its pvmd. Connecting (for *anonymous* tasks) is slightly different from reconnecting (for spawned tasks).

The pvmd publishes the address of the socket on which it listens in `/tmp/pvmd.`*uid*, where *uid* is the numeric user id under which the pvmd runs. This file contains a line of the form

`7f000001:06f7` or `/tmp/aaa014138`

This is the IP address and port number (in hexadecimal) of the socket, or the path if a Unix-domain socket. To avoid the need to read the address file, the same information is passed to spawned tasks in environment variable `PVMSOCK`.

To reconnect, a spawned task also needs its expected process id. When a task is spawned by the pvmd, a task descriptor is created for it during the *exec* phase. The descriptor must exist so it can stash any messages that arrive for the task before it reconnects and can receive them. During reconnection, the task identifies itself to the pvmd by its PID. If the task is always the child of the pvmd (i.e., the exact process *exec*'d by it), then it could use the value returned by `getpid()`. To allow for intervening processes, such as debuggers, the pvmd passes the expected PID in environment variable `PVMEPID`, and the task uses that value in preference to its real PID. The task also passes its real PID so it can be controlled normally by the pvmd.

`pvm_beatask()` creates a TCP socket and does a proper connection dance with the pvmd. Each must prove its identity to the other, to prevent a different user from spoofing the system. It does this by creating a file in `/tmp` writable only by the owner, and challenging the other to write in the file. If successful, the identity of the other is proven. Note that this authentication is only as strong as the filesystem and the authority of root on each machine.

A protocol serial number (`TDPROTOCOL`) is compared whenever a task connects to a pvmd or another task. This number is incremented whenever a change in the protocol makes it incompatible with the previous version.

Disconnecting is much simpler. It can be done forcibly by a *close* from either end, for example, by exiting the task process. The function `pvm_exit()` performs a clean shutdown, such that the process can be connected again later (it would get a different TID).

7.5 Protocols

PVM communication is based on TCP , UDP, and Unix-domain sockets. While more appropriate protocols exist, they aren't as generally available.

VMTP [4] is one example of a protocol built for this purpose. Although intended for RPC-style interaction (request-response), it could support PVM messages. It is packet oriented and efficiently sends short blocks of data (such as most pvmd-pvmd management messages) but also handles streaming (necessary for task-task communication). It supports multicasting and priority data (something PVM doesn't need yet). Connections don't need to be established before use; the first communication initializes the protocol drivers at each end. VMTP was rejected, however. because it is not widely available (using it requires modifying the kernel).

This section explains the PVM protocols. There are three connections to consider: Between pvmds, between pvmd and task, and between tasks.

7.5.1 Messages

The pvmd and libpvm use the same message header, shown in Figure 7.8. *Code* contains an integer tag (message type). Libpvm uses *Encoding* to pass the encoding style of the message, as it can pack in different formats. The pvmd always sets Encoding (and requires that it be set) to 1 (*foo*), Pvmds use the *Wait Context* field to pass the wait id's (if any, zero if none) of the *waitc* associated with the message. Certain tasks (resource manager, tasker, hoster) also use wait id's. The *Checksum* field is reserved for future use. Messages are sent in one or more fragments, each with its own fragment header (described below). The message header is at the beginning of the first fragment.

7.5.2 Pvmd-Pvmd

PVM daemons communicate with one another through UDP sockets. UDP is an unreliable delivery service which can lose, duplicate or reorder packets, so an acknowledgment and retry mechanism is used. UDP also limits packet length, so PVM fragments long messages.

We considered TCP, but three factors make it inappropriate. First is scalability. In

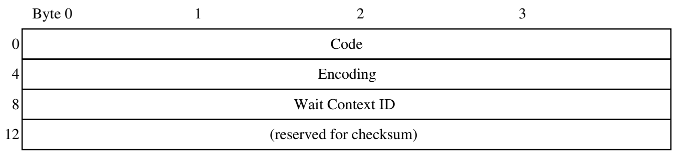

Figure 7.8
Message header

a virtual machine of N hosts, each pvmd must have connections to the other $N-1$. Each open TCP connection consumes a file descriptor in the pvmd, and some operating systems limit the number of open files to as few as 32, whereas a single UDP socket can communicate with any number of remote UDP sockets. Second is overhead. N pvmds need $N(N-1)/2$ TCP connections, which would be expensive to set up. The PVM/UDP protocol is initialized with no communication. Third is fault tolerance. The communication system detects when foreign pvmds have crashed or the network has gone down, so we need to set timeouts in the protocol layer. The TCP keepalive option might work, but it's not always possible to get adequate control over the parameters.

The packet header is shown in Figure 7.9. Multibyte values are sent in (Internet) *network byte order* (most significant byte first).

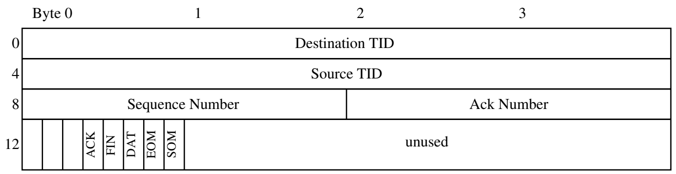

Figure 7.9
Pvmd-pvmd packet header

The source and destination fields hold the TIDs of the true source and final destination of the packet, regardless of the route it takes. Sequence and acknowledgment numbers start at 1 and increment to 65535, then wrap to zero.

SOM (EOM) – Set for the first (last) fragment of a message. Intervening fragments have both bits cleared. They are used by tasks and pvmds to delimit message boundaries.

DAT – If set, data is contained in the packet, and the sequence number is valid. The

packet, even if zero length, must be delivered.

ACK – If set, the acknowledgment number field is valid. This bit may be combined with the DAT bit to piggyback an acknowledgment on a data packet.[1]

FIN – The pvmd is closing down the connection. A packet with FIN bit set (and DAT cleared) begins an orderly shutdown. When an acknowledgement arrives (ACK bit set and ack number matching the sequence number from the FIN packet), a final packet is sent with both FIN and ACK set. If the pvmd panics, (for example on a trapped segment violation) it tries to send a packet with FIN and ACK set to every peer before it exits.

The state of a connection to another pvmd is kept in its host table entry. The protocol driver uses the following fields of `struct hostd`:

Field	Meaning
`hd_hostpart`	TID of pvmd
`hd_mtu`	Max UDP packet length to host
`hd_sad`	IP address and UDP port number
`hd_rxseq`	Expected next packet number from host
`hd_txseq`	Next packet number to send to host
`hd_txq`	Queue of packets to send
`hd_opq`	Queue of packets sent, awaiting ack
`hd_nop`	Number of packets in `hd_opq`
`hd_rxq`	List of out-of-order received packets
`hd_rxm`	Buffer for message reassembly
`hd_rtt`	Estimated smoothed round-trip time

Figure 7.10 shows the host send and outstanding-packet queues. Packets waiting to be sent to a host are queued in FIFO `hd_txq`. Packets are appended to this queue by the routing code, described in Section 7.6.1. No receive queues are used; incoming packets are passed immediately through to other send queues or reassembled into messages (or discarded). Incoming messages are delivered to a pvmd entry point as described in Section 7.2.4.

The protocol allows multiple outstanding packets to improve performance over high-latency networks, so two more queues are required. `hd_opq` holds a per-host list of unacknowledged packets, and global `opq` lists all unacknowledged packets, ordered by time to retransmit. `hd_rxq` holds packets received out of sequence until they can be accepted.

The difference in time between sending a packet and getting the acknowledgement is used to estimate the round-trip time to the foreign host. Each update is filtered into the

[1] Currently, the pvmd generates an acknowledgement packet for each data packet.

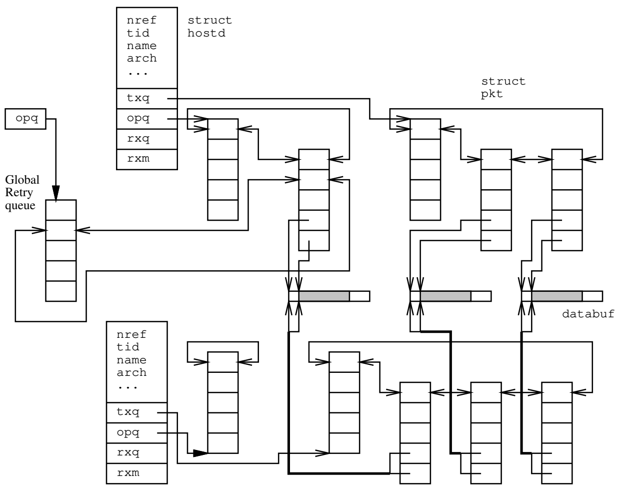

Figure 7.10
Host descriptors with send queues

estimate according to the formula

$$hd\_rtt_n = 0.75 * hd\_rtt_{n-1} + 0.25 * \Delta t.$$

When the acknowledgment for a packet arrives, the packet is removed from `hd_opq` and `opq` and discarded. Each packet has a retry timer and count, and each is resent until acknowledged by the foreign pvmd. The timer starts at $3 * hd\_rtt$, and doubles for each retry up to 18 seconds. `hd_rtt` is limited to nine seconds, and backoff is bounded in order to allow at least 10 packets to be sent to a host before giving up. After three minutes of resending with no acknowledgment, a packet expires.

If a packet expires as a result of timeout, the foreign pvmd is assumed to be down or unreachable, and the local pvmd gives up on it, calling `hostfailentry()`

7.5.3 Pvmd-Task and Task-Task

A task talks to its pvmd and other tasks through TCP sockets. TCP is used because it delivers data reliably. UDP can lose packets even within a host. Unreliable delivery requires retry (with timers) at both ends: since tasks can't be interrupted while computing to perform I/O, we can't use UDP.

Implementing a packet service over TCP is simple because of its reliable delivery. The packet header is shown in Figure 7.11. No sequence numbers are needed, and only flags *SOM* and *EOM* (these have the same meaning as in Section 7.5.2). Since TCP provides no record marks to distinguish back-to-back packets from one another, the length is sent in the header. Each side maintains a FIFO of packets to send, and switches between reading the socket when data is available and writing when there is space.

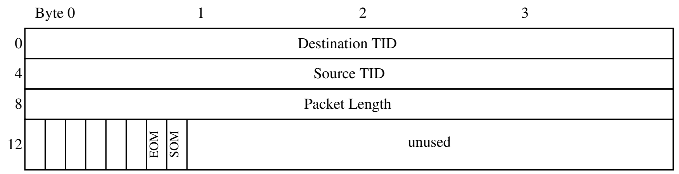

Figure 7.11
Pvmd-task packet header

The main drawback to TCP (as opposed to UDP) is that more system calls are needed to transfer each packet. With UDP, a single `sendto()` and single `recvfrom()` are required. With TCP, a packet can be sent by a single `write()` call, but must be received by two `read()` calls, the first to get the header and the second to get the data.

When traffic on the connection is heavy, a simple optimization reduces the average number of reads back to about one per packet. If, when reading the packet body, the requested length is increased by the size of a packet header, the read may succeed in getting both the packet body and header of the next packet at once. We have the header for the next packet for free and can repeat this process.[2]

Version 3.3 introduced the use of Unix-domain stream sockets as an alternative to TCP for local communication, to improve latency and transfer rate (typically by a factor of two). If enabled (the system is built without the `NOUNIXDOM` option), stream sockets are used between the pvmd and tasks as well as between tasks on the same host.

7.6 Message Routing

7.6.1 Pvmd

Packet Buffers Packet descriptors (`struct pkt`) track message fragments through the pvmd. Fields `pk_buf`, `pk_max`, `pk_dat` and `pk_len` are used in the same ways as similarly named fields of a frag, described in Section 7.2.1. Besides data, pkts contain state to operate the pvmd-pvmd protocol.

Message Routing Messages are sent by calling `sendmessage()`, which routes by destination address. Messages for other pvmds or tasks are linked to packet descriptors and attached to a send queue. If the pvmd addresses a message to itself, `sendmessage()` passes the whole message descriptor to `netentry()`, avoiding the packet layer entirely. This loopback interface is used often by the pvmd. During a complex operation, `netentry()` may be reentered several times as the pvmd sends itself messages.

Messages to the pvmd are reassembled from packets in message reassembly buffers, one for each local task and remote pvmd. Completed messages are passed to entry points (Section 7.2.4).

Packet Routing A graph of packet and message routing inside the pvmd is shown in Figure 7.12. Packets are received from the network by `netinput()` directly into buffers long enough to hold the largest packet the pvmd will receive (its MTU in the host table). Packets from local tasks are read by `loclinput()`, which creates a buffer large enough for each packet after it reads the header. To route a packet, the pvmd chains it onto the queue for its destination. If a packet is multicast (see Section 7.6.4), the descriptor is replicated, counting extra references on the underlying databuf. One copy is placed in each send queue. After the last copy of the packet is sent, the databuf is freed.

[2] This was once implemented, but was removed while the code was updated and hasn't been reintroduced.

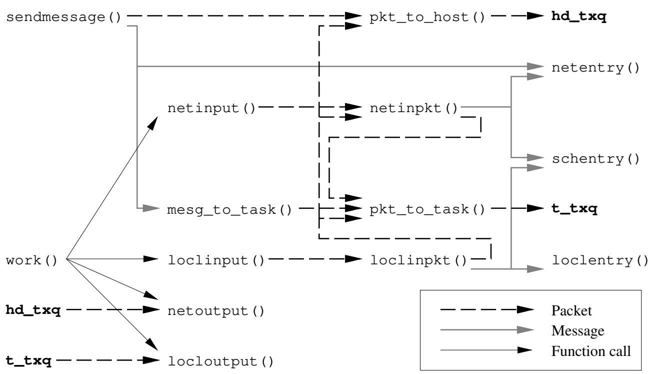

Figure 7.12
Packet and message routing in pvmd

Refragmentation Messages are generally built with fragment length equal to the MTU of the host's pvmd, allowing them to be forwarded without refragmentation. In some cases, the pvmd can receive a packet (from a task) too long to be sent to another pvmd. The pvmd refragments the packet by replicating its descriptor as many times as necessary. A single databuf is shared between the descriptors. The `pk_dat` and `pk_len` fields of the descriptors cover successive chunks of the original packet, each chunk small enough to send. The `SOM` and `EOM` flags are adjusted (if the original packet is the start or end of a message). At send time, `netoutput()` saves the data under where it writes the packet header, sends the packet, and then restores the data.

7.6.2 Pvmd and Foreign Tasks

Pvmds usually don't communicate with foreign tasks (those on other hosts). The pvmd has message reassembly buffers for each foreign pvmd and each task it manages. What it doesn't want is to have reassembly buffers for foreign tasks. To free up the reassembly buffer for a foreign task (if the task dies), the pvmd would have to request notification from the task's pvmd, causing extra communication.

For the sake of simplicity the pvmd local to the sending task serves as a message repeater. The message is reassembled by the task's local pvmd as if it were the receiver, then forwarded all at once to the destination pvmd, which reassembles the message again. The source address is preserved, so the sender can be identified.

Libpvm maintains dynamic reassembly buffers, so messages from pvmd to task do not cause a problem.

7.6.3 Libpvm

Four functions handle all packet traffic into and out of libpvm. `mroute()` is called by higher-level functions such as `pvm_send()` and `pvm_recv()` to copy messages into and out of the task. It establishes any necessary routes before calling `mxfer()`. `mxfer()` polls for messages, optionally blocking until one is received or until a specified timeout. It calls `mxinput()` to copy fragments into the task and reassemble messages. In the generic version of PVM, `mxfer()` uses `select()` to poll all routes (sockets) in order to find those ready for input or output. `pvmmctl()` is called by `mxinput()` when a control message (Section 7.2.5) is received.

Direct Message Routing Direct routing allows one task to send messages to another through a TCP link, avoiding the overhead of forwarding through the pvmds. It is implemented entirely in libpvm, using the notify and control message facilities. By default, a task routes messages to its pvmd, which forwards them on. If direct routing is enabled (`PvmRouteDirect`) when a message (addressed to a task) is passed to `mroute()`, it attempts to create a direct route if one doesn't already exist. The route may be granted or refused by the destination task, or fail (if the task doesn't exist). The message is then passed to `mxfer()`.

Libpvm maintains a protocol control block (`struct ttpcb`) for each active or denied connection, in list `ttlist`. The state diagram for a ttpcb is shown in Figure 7.13. To request a connection, `mroute()` makes a ttpcb and socket, then sends a `TC_CONREQ` control message to the destination via the default route. At the same time, it sends a `TM_NOTIFY` message to the pvmd, to be notified if the destination task exits, with closure (message tag) `TC_TASKEXIT`. Then it puts the ttpcb in state `TTCONWAIT`, and calls `mxfer()` in blocking mode repeatedly until the state changes.

When the destination task enters `mxfer()` (for example, to receive a message), it receives the `TC_CONREQ` message. The request is granted if its routing policy (`pvmrouteopt != PvmDontRoute`) and implementation allow a direct connection, it has resources available, and the protocol version (`TDPROTOCOL`) in the request matches its own. It makes a ttpcb with state `TTGRNWAIT`, creates and listens on a socket, and then replies with a `TC_CONACK` message. If the destination denies the connection, it nacks, also with a `TC_CONACK` message. The originator receives the `TC_CONACK` message, and either opens the connection (`state = TTOPEN`) or marks the route denied (`state = TTDENY`). Then,

`mroute()` passes the original message to `mxfer()`, which sends it. Denied connections are cached in order to prevent repeated negotiation.

If the destination doesn't exist, the `TC_CONACK` message never arrives because the `TC_CONREQ` message is silently dropped. However, the `TC_TASKEXIT` message generated by the notify system arrives in its place, and the ttpcb state is set to `TTDENY`.

This connect scheme also works if both ends try to establish a connection at the same time. They both enter `TTCONWAIT`, and when they receive each other's `TC_CONREQ` messages, they go directly to the `TTOPEN` state.

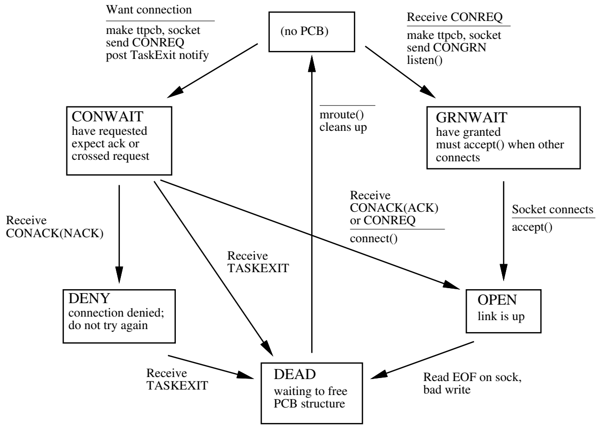

Figure 7.13
Task-task connection state diagram

7.6.4 Multicasting

The libpvm function `pvm_mcast()` sends a message to multiple destinations simultaneously. The current implementation only routes multicast messages through the pvmds. It uses a 1:N fanout to ensure that failure of a host doesn't cause the loss of any messages (other than ones to that host). The packet routing layer of the pvmd cooperates with the libpvm to multicast a message.

To form a multicast address TID (GID), the G bit is set (refer to Figure 7.1). The L field is assigned by a counter that is incremented for each multicast, so a new multicast address is used for each message, then recycled.

To initiate a multicast, the task sends a `TM_MCA` message to its pvmd, containing a list of recipient TIDs. The pvmd creates a multicast descriptor (`struct mca`) and GID. It sorts the addresses, removes bogus ones, and duplicates and caches them in the mca. To each destination pvmd (ones with destination tasks), it sends a `DM_MCA` message with the GID and destinations on that host. The GID is sent back to the task in the `TM_MCA` reply message.

The task sends the multicast message to the pvmd, addressed to the GID. As each packet arrives, the routing layer copies it to each local task and foreign pvmd. When a multicast packet arrives at a destination pvmd, it is copied to each destination task. Packet order is preserved, so the multicast address and data packets arrive in order at each destination. As it forwards multicast packets, each pvmd eavesdrops on the header flags. When it sees a packet with `EOM` flag set, it flushes the mca.

7.7 Task Environment

7.7.1 Environment Variables

Experience seems to indicate that inherited environment (Unix `environ`) is useful to an application. For example, environment variables can be used to distinguish a group of related tasks or to set debugging variables.

PVM makes increasing use of environment, and may eventually support it even on machines where the concept is not native. For now, it allows a task to export any part of `environ` to tasks spawned by it. Setting variable `PVM_EXPORT` to the names of other variables causes them to be exported through spawn. For example, setting

```
PVM_EXPORT=DISPLAY:SHELL
```

exports the variables `DISPLAY` and `SHELL` to children tasks (and `PVM_EXPORT` too).

The following environment variables are used by PVM. The user may set these:

`PVM_ROOT`	Root installation directory
`PVM_EXPORT`	Names of environment variables to inherit through spawn
`PVM_DPATH`	Default slave pvmd install path
`PVM_DEBUGGER`	Path of debugger script used by spawn

The following variables are set by PVM and should not be modified:

`PVM_ARCH`	PVM architecture name
`PVMSOCK`	Address of the pvmd local socket; see Section 7.4.2
`PVMEPID`	Expected PID of a spawned task
`PVMTMASK`	Libpvm Trace mask

7.7.2 Standard Input and Output

Each task spawned through PVM has `/dev/null` opened for *stdin*. From its parent, it inherits a *stdout sink*, which is a *(TID, code)* pair. Output on *stdout* or *stderr* is read by the pvmd through a pipe, packed into PVM messages and sent to the TID, with message tag equal to the code. If the output TID is set to zero (the default for a task with no parent), the messages go to the master pvmd, where they are written on its error log.

Children spawned by a task inherit its stdout sink. Before the spawn, the parent can use `pvm_setopt()` to alter the output TID or code. This doesn't affect where the output of the parent task itself goes. A task may set output TID to one of three settings: the value inherited from its parent, its own TID, or zero. It can set output code only if output TID is set to its own TID. This means that output can't be assigned to an arbitrary task.

Four types of messages are sent to an stdout sink. The message body formats for each type are as follows:

Spawn:	`(code) {`		Task has been spawned
		`int tid,`	Task id
		`int -1,`	Signals spawn
		`int ptid`	TID of parent
	`}`		
Begin:	`(code) {`		First output from task
		`int tid,`	Task id
		`int -2,`	Signals task creation
		`int ptid`	TID of parent
	`}`		
Output:	`(code) {`		Output from a task
		`int tid,`	Task id
		`int count,`	Length of output fragment
		`char data[count]`	Output fragment
	`}`		
End:	`(code) {`		Last output from a task
		`int tid,`	Task id
		`int 0`	Signals EOF
	`}`		

The first two items in the message body are always the task id and output count, which allow the receiver to distinguish between different tasks and the four message types. For each task, one message each of types *Spawn*, *Begin*, and *End* is sent, along with zero or more messages of class *Output*, (`count > 0`). Classes *Begin*, *Output* and *End* will be received in order, as they originate from the same source (the pvmd of the target task). Class *Spawn* originates at the (possibly different) pvmd of the parent task, so it can be received in any order relative to the others. The output sink is expected to understand the different types of messages and use them to know when to stop listening for output from a task (EOF) or group of tasks (global EOF).

The messages are designed so as to prevent race conditions when a task spawns another task, then immediately exits. The output sink might get the *End* message from the parent task and decide the group is finished, only to receive more output later from the child task. According to these rules, the *Spawn* message for the second task must arrive before the *End* message from the first task. The *Begin* message itself is necessary because the *Spawn* message for a task may arrive after the *End* message for the same task. The state transitions of a task as observed by the receiver of the output messages are shown in Figure 7.14.

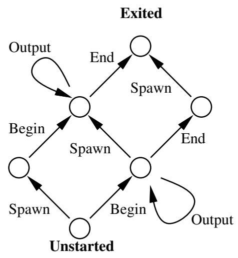

Figure 7.14
Output states of a task

The libpvm function `pvm_catchout()` uses this output collection feature to put the output from children of a task into a file (for example, its own stdout). It sets output TID to its own task id, and the output code to control message `TC_OUTPUT`. Output from children and grandchildren tasks is collected by the pvmds and sent to the task, where it is received by `pvmmctl()` and printed by `pvmclaimo()`.

7.7.3 Tracing

The libpvm library has a tracing system that can record the parameters and results of all calls to interface functions. Trace data is sent as messages to a trace sink task just as output is sent to an stdout sink (Section 7.7.2). If the trace output TID is set to zero (the default), tracing is disabled.

Besides the trace sink, tasks also inherit a trace mask, used to enable tracing function-by-function. The mask is passed as a (printable) string in environment variable `PVMTMASK`. A task can manipulate its own trace mask or the one to be inherited from it. A task's trace mask can also be set asynchronously with a `TC_SETTMASK` control message.

Constants related to trace messages are defined in public header file `pvmtev.h`. Trace data from a task is collected in a manner similar to the output redirection discussed above. Like the type *Spawn*, *Begin*, and *End* messages which bracket output from a task, `TEV_SPNTASK`, `TEV_NEWTASK` and `TEV_ENDTASK` trace messages are generated by the pvmds to bracket trace messages.

The tracing system was introduced in version 3.3 and is still expected to change somewhat.

7.7.4 Debugging

PVM provides a simple but extensible debugging facility. Tasks started by hand could just as easily be run under a debugger, but this procedure is cumbersome for those spawned by an application, since it requires the user to comment out the calls to `pvm_spawn()` and start tasks manually. If `PvmTaskDebug` is added to the flags passed to `pvm_spawn()`, the task is started through a debugger script (a normal shell script), `$PVM_ROOT/lib/debugger`.

The pvmd passes the name and parameters of the task to the debugger script, which is free to start any sort of debugger. The script provided is very simple. In an *xterm* window, it runs the correct debugger according to the architecture type of the host. The script can be customized or replaced by the user. The pvmd can be made to execute a different debugger via the `bx=` host file option or the `PVM_DEBUGGER` environment variable.

7.8 Console Program

The PVM console is used to manage the virtual machine—to reconfigure it or start and stop processes. In addition, it's an example program that makes use of most of the libpvm functions.

`pvm_getfds()` and `select()` are used to check for input from the keyboard and messages from the pvmd simultaneously. Keyboard input is passed to the command interpreter, while messages contain notification (for example, HostAdd) or output from a task.

The console can collect output or trace messages from spawned tasks, using the redirection mechanisms described in Section 7.7.2 and Section 7.7.3, and write them to the screen or a file. It uses the *begin* and *end* messages from child tasks to maintain groups of tasks (or *jobs*), related by common ancestors. Using the `PvmHostAdd` notify event, it informs the user when the virtual machine is reconfigured.

7.9 Resource Limitations

Resource limits imposed by the operating system and available hardware are in turn passed to PVM applications. Whenever possible, PVM avoids setting explicit limits; instead, it returns an error when resources are exhausted. Competition between users on the same host or network affects some limits dynamically.

7.9.1 In the PVM Daemon

How many tasks each pvmd can manage is limited by two factors: the number of processes allowed a user by the operating system, and the number of file descriptors available to the pvmd. The limit on processes is generally not an issue, since it doesn't make sense to have a huge number of tasks running on a uniprocessor machine.

Each task consumes one file descriptor in the pvmd, for the pvmd-task TCP stream. Each spawned task (not ones connected anonymously) consumes an extra descriptor, since its output is read through a pipe by the pvmd (closing stdout and stderr in the task would reclaim this slot). A few more file descriptors are always in use by the pvmd for the local and network sockets and error log file. For example, with a limit of 64 open files, a user should be able to have up to 30 tasks running per host.

The pvmd may become a bottleneck if all these tasks try to talk to one another through it.

The pvmd uses dynamically allocated memory to store message packets en route between tasks. Until the receiving task accepts the packets, they accumulate in the pvmd in an FIFO procedure. No flow control is imposed by the pvmd: it will happily store all the packets given to it, until it can't get any more memory. If an application is designed so that tasks can keep sending even when the receiving end is off doing something else and not receiving, the system will eventually run out of memory.

7.9.2 In the Task

As with the pvmd, a task may have a limit on the number of others it can connect to directly. Each direct route to a task has a separate TCP connection (which is bidirectional), and so consumes a file descriptor. Thus, with a limit of 64 open files, a task can establish direct routes to about 60 other tasks. Note that this limit is in effect only when using task-task direct routing. Messages routed via the pvmds use only the default pvmd-task connection.

The maximum size of a PVM message is limited by the amount of memory available to the task. Because messages are generally packed using data existing elsewhere in memory, and they must be reside in memory between being packed and sent, the largest possible message a task can send should be somewhat less than half the available memory. Note that as a message is sent, memory for packet buffers is allocated by the pvmd, aggravating the situation. In-place message encoding alleviates this problem somewhat, because the data is not copied into message buffers in the sender. However, on the receiving end, the entire message is downloaded into the task before the receive call accepts it, possibly leaving no room to unpack it.

In a similar vein, if many tasks send to a single destination all at once, the destination task or pvmd may be overloaded as it tries to store the messages. Keeping messages from being freed when new ones are received by using `pvm_setrbuf()` also uses up memory.

These problems can sometimes be avoided by rearranging the application code, for example, to use smaller messages, eliminate bottlenecks, and process messages in the order in which they are generated.

7.10 Multiprocessor Systems

Developed initially as a parallel programming environment for Unix workstations, PVM has gained wide acceptance and become a de facto standard for message-passing programming. Users want the same programming environment on multiprocessor computers so they can move their applications onto these systems. A common interface would also allow users to write vendor-independent programs for parallel computers and to do part or most of the development work on workstations, freeing up the multiprocessor supercomputers for production runs.

With PVM, multiprocessor systems can be included in the same configuration with workstations. For example, a PVM task running on a graphics workstation can display the results of computations carried out on a massively parallel processing supercomputer. Shared-memory computers with a small number of processors can be linked to deliver supercomputer performance.

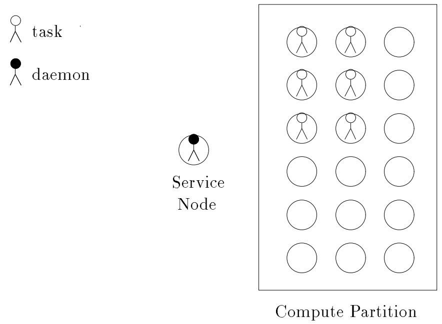

Figure 7.15
PVM daemon and tasks on MPP host

The virtual machine hides the configuration details from the programmer. The physical processors can be a network of workstations, or they can be the nodes of a multicomputer. The programmer doesn't have to know how the tasks are created or where they are running; it is the responsibility of PVM to schedule user's tasks onto individual processors. The user can, however, tune the program for a specific configuration to achieve maximum performance, at the expense of its portability.

Multiprocessor systems can be divided into two main categories: message passing and shared memory. In the first category, PVM is now supported on Intel's iPSC/860 and Paragon, as well as Thinking Machine's CM-5. Porting PVM to these platforms is straightforward, because the message-passing functions in PVM map quite naturally onto the native system calls. The difficult part is the loading and management of tasks. In the second category, message passing can be done by placing the message buffers in shared memory. Access to these buffers must be synchronized with mutual exclusion locks. PVM 3.3 shared memory ports include SGI multiprocessor machines running IRIX 5.x and Sun Microsystems, Inc., multiprocessor machines running Solaris 2.3 (This port also runs on the Cray Research, Inc., CS6400). In addition, CRAY and DEC have created PVM ports for their T3D and DEC 2100 shared memory multiprocessors, respectively.

7.10.1 Message-Passing Architectures

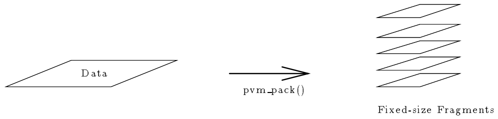

Figure 7.16
Packing: breaking data into fixed-size fragments

A typical MPP system has one or more service nodes for user logins and a large number of compute nodes for number crunching. The PVM daemon runs on one of the service nodes and serves as the gateway to the outside world. A task can be started on any one of the service nodes as a Unix process and enrolls in PVM by establishing a TCP socket connection to the daemon. The only way to start PVM tasks on the compute nodes is via pvm_spawn(). When the daemon receives a request to spawn new tasks, it will allocate a set of nodes if necessary, and load the executable onto the specified number of nodes.

The way PVM allocates nodes is system dependent. On the CM-5, the entire partition is allocated to the user. On the iPSC/860, PVM will get a subcube big enough to accommodate all the tasks to be spawned. Tasks created with two separate calls to pvm_spawn() will reside in different subcubes, although they can exchange messages directly by using the physical node address. The NX operating system limits the number of active subcubes system-wide to 10. Pvm_spawn will fail when this limit is reached or when there are not enough nodes available. In the case of the Paragon, PVM uses the default partition unless a different one is specified when pvmd is invoked. Pvmd and the spawned tasks form one giant parallel application. The user can set the appropriate NX environment variables such as NX_DFLT_SIZE before starting PVM, or he can specify the equivalent command-line arguments to pvmd (i.e., pvmd -sz 32).

PVM message-passing functions are implemented in terms of the native send and receive system calls. The "address" of a task is encoded in the task id, as illustrated in Figure 7.17. This enables the messages to be sent directly to the target task, without any

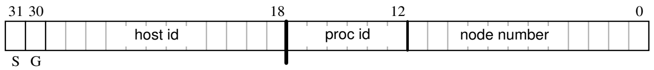

Figure 7.17
How TID is used to distinguish tasks on MPP

help from the daemon. The node number is normally the logical node number, but the

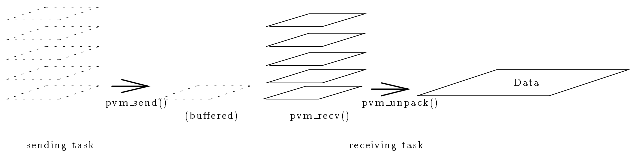

Figure 7.18
Buffering: buffering one fragment by receiving task until pvm_recv() is called

physical address is used on the iPSC/860 to allow for direct intercube communication. The instance number is used to distinguish tasks running on the same node.

PVM normally uses asynchronous send primitives to send messages. The operating system can run out of message handles very quickly if a lot of small messages or several large messages are sent at once. PVM will be forced to switch to synchronous send when there are no more message handles left or when the system buffer gets filled up. To improve performance, a task should call pvm_send() as soon as the data becomes available, so (one hopes) when the other task calls pvm_recv(), the message will already be in its buffer. PVM buffers one incoming packet between calls to pvm_send()/pvm_recv(). A large message, however, is broken up into many fixed-size fragments during packing, and each piece is sent separately. Buffering one of these fragments is not sufficient unless pvm_send() and pvm_recv() are synchronized. Figures 7.16 and 7.18 illustrate this process.

The front end of an MPP system is treated as a regular workstation. Programs to be run there should be linked with the regular PVM library, which relies on Unix sockets to transmit messages. Normally one should avoid running processes on the front end, because communication between those processes and the node processes must go through the PVM daemon and a TCP socket link. Most of the computation and communication should take place on the compute nodes in order to take advantage of the processing power of these nodes and the fast interconnects between them.

Since the PVM library for the front end is different from the one for the nodes, the executable for the front end must be different from the one compiled for the nodes. An SPMD program, for example, has only one source file, but the object code must be linked with the front end and node PVM libraries separately to produce two executables if it is to be started from the front end. An alternative would be a "hostless" SPMD program, which could be spawned from the PVM console.

Table 7.1 shows the native system calls used by the corresponding PVM functions on

Table 7.1
Implementation of PVM system calls

PVM	iPSC860	Paragon	CM-5
pvm_spawn	getcube/load	nx_loadve	fork
pvm_send	isend/csend	isend/csend	CMMD_send_async/_noblock
pvm_recv	irecv	irecv	CMMD_receive_async
pvm_mcast	gsendx	gsendx	CMMD_send_async/_noblock

various platforms.

The CM-5 is somewhat different from the Intel systems because it requires a special host process for each group of tasks spawned. This process enrolls in PVM and relays messages between pvmd and the node programs. This, needless to say, adds even more overhead to daemon-task communications.

Another restrictive feature of the CM-5 is that all nodes in the same partition are scheduled as a single unit. The partitions are normally configured by the system manager and each partition must contain at least 16 processors. User programs are run on the entire partition by default. Although it is possible to idle some of the processors in a partition, as PVM does when fewer nodes are called for, there is no easy way to harness the power of the idle processors. Thus, if PVM spawns two groups of tasks, they will time-share the partition, and any intergroup traffic must go through pvmd.

Additionally, CMMD has no support for multicasting. Thus, pvm_mcast() is implemented with a loop of CMMD_async_send().

7.10.2 Shared-Memory Architectures

The shared-memory architecture provides a very efficient medium for processes to exchange data. In our implementation, each task owns a shared buffer created with the shmget() system call. The task id is used as the "key" to the shared segment. If the key is being used by another user, PVM will assign a different id to the task. A task communicates with other tasks by mapping their message buffers into its own memory space.

To enroll in PVM, the task first writes its Unix process id into pvmd's incoming box. It then looks for the assigned task id in pvmd's pid→TID table.

The message buffer is divided into pages, each of which holds one fragment (Figure 7.19). PVM's page size can be a multiple of the system page size. Each page has a header, which contains the lock and the reference count. The first few pages are used as the incoming box, while the rest of the pages hold outgoing fragments (Figure 7.20).

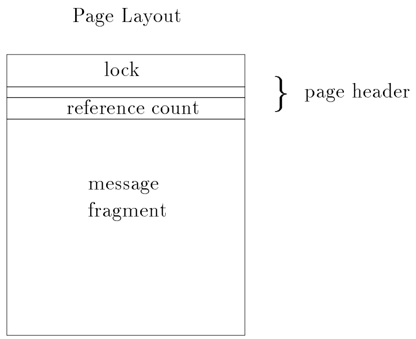

Figure 7.19
Structure of a PVM page

To send a message, the task first packs the message body into its buffer, then delivers the message header (which contains the sender's TID and the location of the data) to the incoming box of the intended recipient. When pvm_recv() is called, PVM checks the incoming box, locates and unpacks the messages (if any), and decreases the reference count so the space can be reused. If a task is not able to deliver the header directly because the receiving box is full, it will block until the other task is ready.

Inevitably some overhead will be incurred when a message is packed into and unpacked from the buffer, as is the case with all other PVM implementations. If the buffer is full, then the data must first be copied into a temporary buffer in the process's private space and later transferred to the shared buffer.

Memory contention is usually not a problem. Each process has its own buffer, and each page of the buffer has its own lock. Only the page being written to is locked, and no process should be trying to read from this page because the header has not been sent out. Different processes can read from the same page without interfering with each other, so multicasting will be efficient (they do have to decrease the counter afterwards, resulting in some contention). The only time contention occurs is when two or more processes trying to deliver the message header to the same process at the same time. But since the header is very short (16 bytes), such contention should not cause any significant delay.

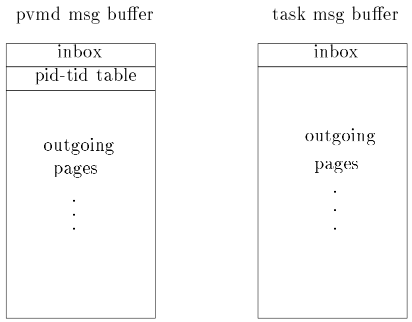

Figure 7.20
Structures of shared message buffers

To minimize the possibility of page faults, PVM attempts to use only a small number of pages in the message buffer and recycle them as soon as they have been read by all intended recipients.

Once a task's buffer has been mapped, it will not be unmapped unless the system limits the number of mapped segments. This strategy saves time for any subsequent message exchanges with the same process.

7.10.3 Optimized Send and Receive on MPP

In the original implementation, all user messages are buffered by PVM. The user must pack the data into a PVM buffer before sending it, and unpack the data after it has been received into an internal buffer. This approach works well on systems with relatively high communication latency, such as the Ethernet. On MPP systems the packing and unpacking introduce substantial overhead. To solve this problem we added two new PVM functions, namely pvm_psend() and pvm_precv(). These functions combine packing/unpacking and sending/receiving into one single step. They could be mapped directly into the native message passing primitives available on the system, doing away with internal buffers altogether. On the Paragon these new functions give almost the same performance as the native ones.

Although the user can use both pvm_psend() and pvm_send() in the same program, on MPP the pvm_psend() must be matched with pvm_precv(), and pvm_send() with pvm_recv().