PVM on the RHODOS Distributed Operating System

J. Rough, A. Goscinski, D. De Paoli

{ruffy, ang, ddp}@deakin.edu.au

School of Computing and Mathematics
Deakin University
Geelong, Victoria, 3217

Abstract

This report describes the design and implementation of a syntax-compatible version of the PVM parallel processing tool for the RHODOS distributed operating system. The implementation of the Unix version of the tool is examined in detail, and the design of the tool for RHODOS is presented with a discussion of issues raised by the introduction of a distributed operating system.

Contents

1 Introduction 1
2 The Structure of PVM 2
3 PVM on Unix 4
3.1 Task Management 4
3.2 Interprocess Communication 5
3.2.1 Message Passing 5
3.2.2 Buffer Management 7
3.2.3 Group Communication 8
3.3 The Task Identification Number 9
3.4 Event Notification 10
4 PVM on RHODOS 10
4.1 Task Management 10
4.2 Interprocess Communication 12
4.2.1 Message Passing 12
4.2.2 Buffer Management 13
4.2.3 Group Communication 13
4.3 The Task Identification Number 14
4.4 Event Notification 16
5 Implementation and Testing 16
6 Conclusions 17
7 Bibliography 18

1 Introduction

Parallel Virtual Machine (PVM) is a programming tool for parallel processing that executes on a (possibly heterogeneous) network of workstations running Unix and presents an unified virtual machine to the programmer. PVM was developed under the Heterogeneous Network Computing research project which is a joint venture between Oak Ridge National Laboratory, the University of Tennesse, Emory University, and Carnegie Mellon University [Geist et al 1994]. PVM has been implemented on various architectures including Sun workstations, IBM PCs, Macintoshes, Intel Hypercubes, and Cray supercomputers [PVM 1996].

PVM provides transparent local and remote interprocess communication (message passing), and “task” management. Messages under PVM are strongly typed and data conversion between incompatible architectures is automatic. Tasks are defined as “the unit of parallelism in PVM” and “often but not always a Unix process”. In the case of PVM programs written in the C language, a task is always a process. Process groups are also implemented with group send, barrier synchronisation and global reduce operations. Global reduce operations involve the application of non-associative mathematical operations to user-specified data sets across an entire process group. The final result of the operation is stored in the data structure of one of the tasks in the group (specified by the user). PVM supplies four such mathematical operations: global maximum, minimum, sum, and product [Geist et al 1994].

The Unix operating system provides a complicated interface to interprocess communication (sockets), and provides only two protocols in TCP and UDP. In terms of interprocess communication for parallel processing, TCP does not scale very well (there is too much overhead from maintaining connections between every pair of communicating processes) and UDP would require the implementation of a guaranteed delivery service by the user in order to be useful. PVM counters this by using TCP connections to a local server reducing the impact of connection maintenance to only one connection per process, without adding the overhead of guaranteeing delivery. The UDP protocol is then used for communication between each of the servers, with the servers providing a guaranteed delivery service. The provision of these features has enabled PVM to gain popularity among research institutions that cannot afford the funds for the purchase of traditional massively parallel computers.

The ResearcH Oriented Distributed Operating System (RHODOS) is a modern distributed operating system composed of a microkernel and kernel servers supported by the client-server model. RHODOS provides transparent interprocess communication and process management as does PVM, but at the operating system level. Porting PVM to RHODOS would allow the majority of Unix PVM’s functionality to be replaced with operating system mechanisms, thus affording increases in efficiency and speed.

The major issues that are faced by moving PVM to RHODOS are the heterogeneous nature of the Unix systems supported by PVM versus the homogeneity of RHODOS; differences in interprocess communication architecture; and differences in process management.

The introduction of PVM to the RHODOS operating system will provide a recognisable set of interprocess communication primitives on the RHODOS system to any programmer that is familiar with PVM but is new to the RHODOS operating system. Existing

PVM programs will be able to be compiled for the RHODOS system without modification, thus expanding the range of software that is available for testing and measuring performance. PVM programs will also benefit from the addition of features of the distributed operating system such as transparent dynamic load balancing.

This report outlines the design, implementation and testing of the PVM tool for the RHODOS operating system. A discussion of the structure of the Unix PVM environment and the effects the RHODOS environment has on the structure is presented in Section 2. A review of the major features of the PVM package is presented in Section 3. The design of the PVM tool for the RHODOS operating system is described in Section 4. Section 5 presents a discussion of the implementation and testing carried out on the RHODOS version of PVM. Section 6 concludes this report.

2 The Structure of PVM

The PVM package is composed of two parts: a server that contains the mechanisms required for PVM functionality, and a programming library that provides the programmer with an interface to the server. A ‘console’ program is also included in the package that allows the user to manipulate the virtual machine through a command prompt interface [Geist et al 1994]. Any commands entered at this command prompt are mapped directly to standard PVM primitives, hence this is only a utility.

A server is run on each workstation that is part of the virtual machine. The servers cooperate to provide an abstract view of the network of workstations to the user in the form of a unified virtual machine (Figure 1), with each server responsible for the support of the PVM tasks running on the same workstation (Figure 2). This support involves the transparent routing of messages to a destination process’ local server, manipulation of the tasks’ message queues, and process management. The servers also function as a trusted entity for authentication and supply fault detection (in the form of event notification) for building fault tolerant applications [Manchek 1994].

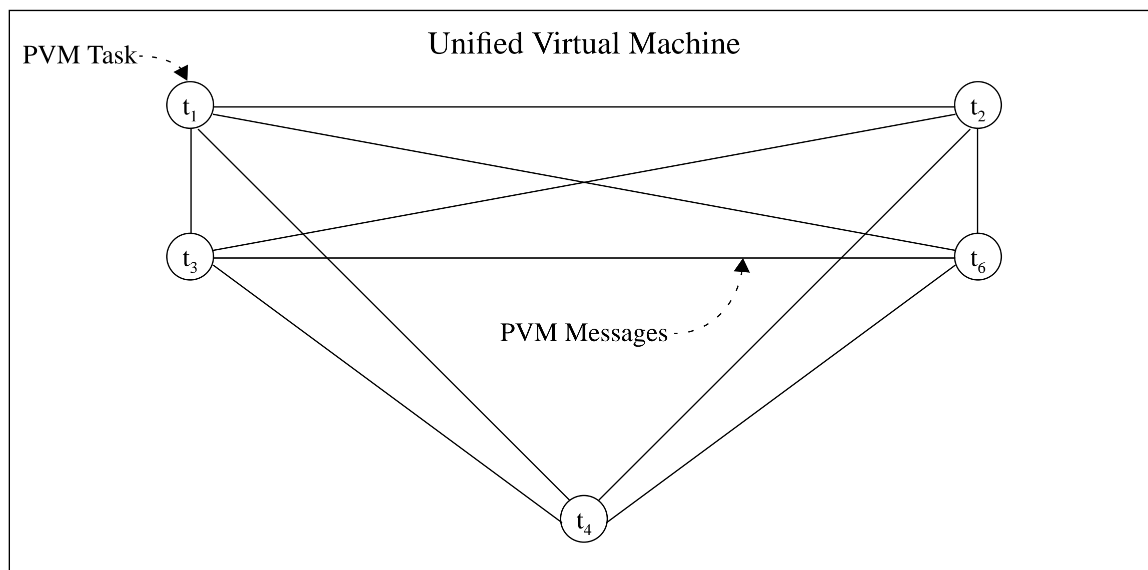

Figure 1: Programmer View of PVM Task Interaction

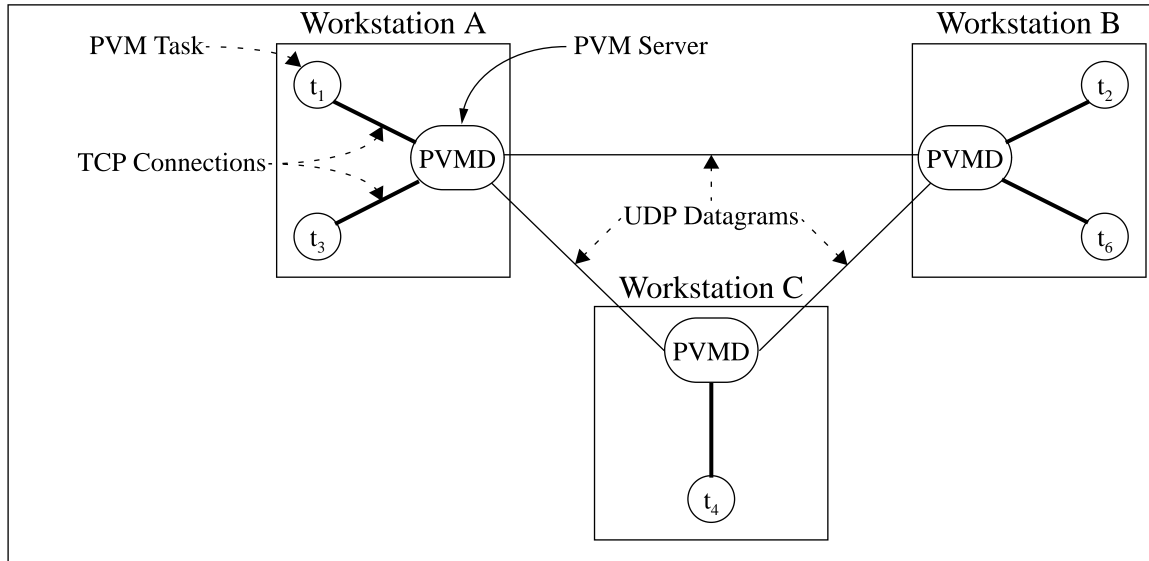

Figure 2: Architecture of PVM on Unix

The first server is manually started by the user, and is known as the master server. The master server in addition to the responsibilities of a slave server also coordinates the addition or removal of workstations to and from the virtual machine. Any requests to manipulate the virtual machine that are received by slave servers are immediately forwarded to the master server for processing [Manchek 1994].

Distributed Operating Systems by their very nature provide a virtual machine, presenting a unified processing environment, instead of a collection of individual workstations connected by a communication subsystem [Goscinski 1991]. For the RHODOS distributed operating system, this has involved the provision of transparency between local and remote operations for interprocess communication [Goscinski et al 1994] and process management [De Paoli et al 1995].

The provision of transparent local and remote interprocess communication and process management by the RHODOS operating system means that it is no longer necessary for the PVM server to provide this functionality. The elimination of this functionality reduces the requirements of the PVM server, so that it now only needs to provide a trusted entity for authentication, and to provide event notification.

The first of these requirements, to provide authentication, can also be eliminated if we examine the semantics of the RHODOS interprocess communication facility. The RHODOS receive operation provides a description of the IPC port where a message was sent, and from this description the owner of that port can be determined. RHODOS also includes support for user authentication ([Wang and Goscinski 1992a] and [Wang and Goscinski 1992b]), hence the owner's name obtained from the description of the sending port can be trusted to authenticate other PVM tasks. The last requirement, to provide event notification, will be addressed in Section 4.4, and is also eliminated.

In summary, the requirements of the server were to provide transparent local and remote interprocess communication and task management, a trusted entity for authentication, and event notification. Each of these requirements has been fulfilled by using functionality

already provided by the RHODOS operating system, thus the server can be safely removed from the model.

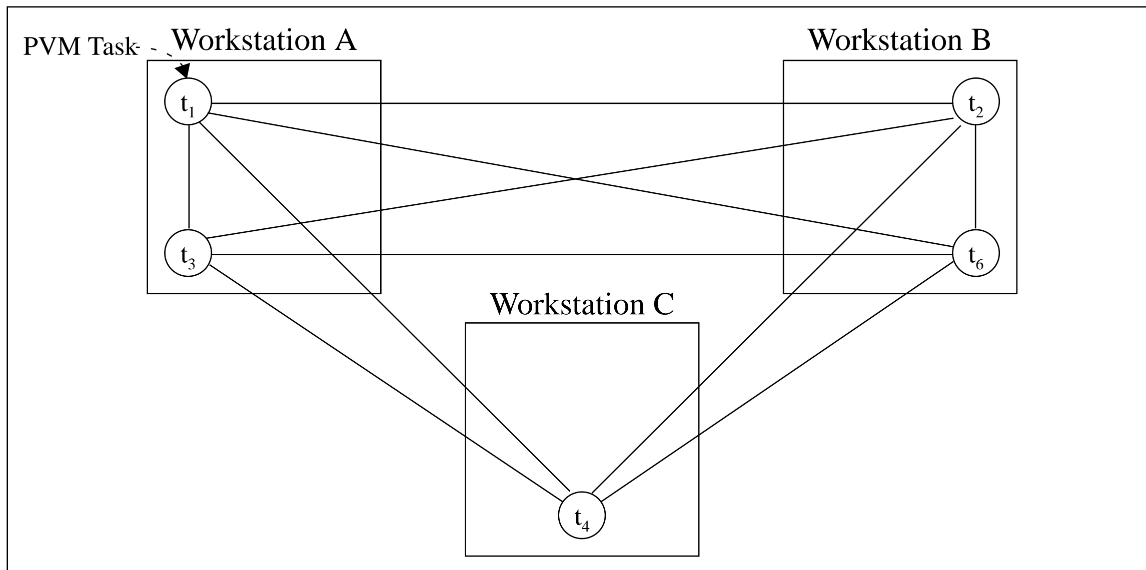

Figure 3: Architecture of PVM on RHODOS

The removal of the server from the PVM model provides additional benefits. Under the Unix PVM, each task depends on the local server for all interaction with other tasks. If the server crashes, the tasks do not have the ability to continue running. With the tasks now communicating directly with each other (Figure 3) this dependency has been removed. This is extended further by the fact that a slave server shuts down when contact is lost with the master server, first terminating all the tasks under its control. This can not longer happen. Thus the reliability of the tasks running under the RHODOS PVM model has been improved. The elimination of the server from the communications model should also decrease the time it takes to send a message from one task to another.

3 PVM on Unix

The major issues that need to be addressed in order to implement PVM on RHODOS are: task management, interprocess communication, task identification, and fault tolerance. It is the goal of this section to present a review of how these issues have been addressed in the Unix version of PVM, so that a functionally equivalent design for RHODOS can be determined.

The design of task management in the Unix PVM is the subject of Section 3.1. A review of the design of interprocess communication in the Unix PVM is presented in Section 3.2. The model used for identifying tasks for task management and interprocess communication is presented in Section 3.3. Finally, the support PVM has for fault tolerant applications is presented in Section 3.4.

3.1 Task Management

PVM provides transparent local and remote task creation (the *pvm_spawn()* primitive) and termination (the *pvm_kill()* primitive) mechanisms for the management of tasks. For task creation, the user also has the option of specifying either a particular architecture or a

particular workstation as the target for the new task. The task creation and termination primitives are implemented by forwarding the creation or termination request to the server that is local to the target workstation/task which then carries out the requested operation. Primitives are also provided for signalling the termination of a task (*pvm_exit()*) and for determining which workstation a particular task is located upon (*pvm_tidtohost()*).

The selection of a target workstation for task creation is the responsibility of the server that is local to the task that requested the operation. The following algorithm is used to select a target workstation [Sunderam 1990]:

(1) a list of candidate workstations is computed;
(2) the next workstation in the pool is selected in a round-robin manner;
(3) a measurement of the load is obtained from the selected workstation;
(4) if the load measurement is below a specific threshold, then the selected workstation is deemed to be the target workstation;
(5) otherwise, the algorithm is repeated from (2).

If none of the workstations in the list of candidates has a load measurement below the threshold, then the workstation with the overall lowest value is deemed to be the target workstation.

Once a target workstation has been selected, a message is dispatched to that workstation to request the creation of the new task. The target workstation attempts to create the task, and returns a message to the caller indicating the success or failure of the operation. Task termination is achieved by sending the task to be terminated the Unix SIGTERM signal using the *pvm_sendsig()* primitive (discussed in Section 3.4).

3.2 Interprocess Communication

Interprocess communication forms an essential part of any parallel processing task. PVM provides the user with message passing for the transmission of PVM managed buffers, as well as support for group communication. A detailed review of these features of the Unix version of PVM is the subject of this Section, in order to allow the design for the RHODOS version to be determined (see Section 4.2).

3.2.1 Message Passing

On the first call to a PVM library function, any state information in the library is initialised and a TCP connection (using Unix domain sockets) is opened to the PVM server. This connection is used for all communication between the PVM task and the server including control messages (such as modification of the virtual machine) and messages for transmission to other tasks. Any messages that are sent to a task that has not yet registered itself with the PVM server are buffered by the server in order to prevent the loss of any messages. The servers communicate with each other using UDP datagrams. These datagrams are used for forwarding any messages between servers [Manchek 1994].

The TCP and UDP protocols were selected because of their wide availability. The TCP protocol is used for task-server communication because using UDP (unreliable quality of service) would require the task to be interrupted periodically to manage the re-transmission or acknowledgement of packets and also the splitting and reconstruction of messages into packets. The use of UDP would also require additional coding by the programmer using PVM

to handle any interruptions that may occur. The UDP protocol was selected for communication between servers in order to eliminate the overhead of TCP connection maintenance and the requirement of a file descriptor for each connection, thus improving scalability [Manchek 1994].

Two paths are available for communicating between tasks. The first of these is known as *PvmRouteDefault*. The task transmits the message to its local server (via TCP) that forwards the message via UDP to the server local to the destination task. The server that is local to the destination task then places the message on the task's message queue [Manchek 1994].

The second path known as *PvmRouteDirect* involves the establishment of a TCP connection directly between the source and destination tasks, followed by the transmission of the message across that connection. The TCP connection is established by the source task first sending a connection request message to the destination task through the default message route discussed above. The destination task creates a TCP port if the direct connection is to be allowed, and returns a positive or negative acknowledgement indicating whether or not the connection is allowed. The source task upon receiving a positive acknowledgement connects to the new TCP port and sends the message, or upon receiving a negative acknowledgement returns an error message indicating the denial of direct routing [Manchek 1994].

Each message that is transmitted using PVM is first encapsulated by a 16 byte header. This header contains the tag that was placed on the message, the type of encoding used on the data in the message, a "wait context ID" (used internally by servers to determine any tasks waiting for the completion of an operation), and 4 bytes that are reserved for future use as a checksum [Manchek 1994].

Messages transmitted between a task and a server are encapsulated by another 16 byte header which is required because of the way the TCP protocol treats any transmitted data (as a continuous stream). This header is composed of the destination TID (see Section 3.3), the source TID, the length of the packet, and flags for indicating the start and/or end of a message [Manchek 1994].

The UDP protocol limits the size of a packet, and thus the fragmentation and reconstruction of messages has been implemented to allow messages of any size to be transmitted. For this feature and to provide a reliable delivery service, a header was required for communication between servers. This header is composed of the destination TID, the source TID, the sequence number of the packet, the number of the packet being acknowledged (if the packet is an acknowledgement message), and flags to indicate whether the packet is an acknowledgement, a server is shutting down, there is data contained within the packet, the packet is the start of a message, and the packet is the end of a message [Manchek 1994].

The message passing primitives that are provided to the user are an asynchronous blocking send (the *pvm_send()* primitive) and receive (the *pvm_recv()* primitive), a non-blocking receive (the *pvm_nrecv()* primitive) and a receive with timeout (the *pvm_trecv()* primitive). The *pvm_probe()* primitive has also been made available for checking for the arrival of a message, along with primitives for multicasting a message to several destinations (*pvm_mcast()*), and combined pack and send (*pvm_psend()*) and receive and unpack primitives (*pvm_precv()*) [Geist et al 1994].

3.2.2 Buffer Management

Message passing is provided through the use of PVM managed buffers. Two buffers are automatically created during initialisation: the default send buffer, and the default receive buffer. Data is 'packed' in to the default send buffer in order to construct a message for transmission. The default receive buffer is used to store the message that is retrieved with a call to a receive primitive. The message content can then be 'unpacked' from this buffer [Geist et al 1994].

The default send buffer must first be initialised using the *pvm_initsend()* primitive before any data can be packed in to it. This operation clears any data that may already be stored in the buffer and also sets the type of encoding that will be performed on data that is stored in the buffer. Three methods of encoding are provided:

(1) *PvmDataDefault* - this method of encoding uses Sun Microsystems' External Data Representation (XDR) standard [Sun 1987] which allows data to be transmitted correctly between tasks that are executing on heterogeneous architectures;

(2) *PvmDataRaw* - with this setting no encoding is performed on the data and hence should only be used for communication across homogeneous architectures (this is not enforced though);

(3) and *PvmDataInPlace* - this encoding is the same as *PvmDataRaw* except that instead of copying the data into the buffer only the location of the data is stored, and then the data is read directly from the user's buffer during the send operation. This has not yet been implemented in the Unix version (as of v3.3.11) [Geist et al 1994] and [PVM 1996].

Unlike the default send buffer, the default receive buffer is automatically reset to empty each time a call is made to a receive operation [Geist et al 1994].

The *pvm_pkbyte()*, *pvm_pkcplx()*, *pvm_pkdcplx()*, *pvm_pkdouble()*, *pvm_pkfloat()*, *pvm_pkint()*, *pvm_pkuint()*, *pvm_pkulong()*, *pvm_pklong()*, and *pvm_pkshort()* primitives are provided for packing an array of variables of their associated type in to the default send buffer. Additional primitives have been provided for packing null terminated strings (the *pvm_pkstr()* primitive) and for packing data in to the default send buffer using a printf-like format (the *pvm_packf()* primitive) [Geist et al 1994].

Similarly, the *pvm_upkbyte()*, *pvm_upkcplx()*, *pvm_upkdcplx()*, *pvm_upkdouble()*, *pvm_upkfloat()*, *pvm_upkint()*, *pvm_upkuint()*, *pvm_upkulong()*, *pvm_upklong()*, and *pvm_upkshort()* primitives are provided for unpacking an array of variables of their associated type from the default receive buffer. Additional primitives have been provided for unpacking null terminated strings (the *pvm_upkstr()* primitive) and for unpacking data from the default receive buffer using a scanf-like format (the *pvm_unpackf()* primitive) [Geist et al 1994].

PVM buffers are identified by a number that is unique to each particular buffer. They can be manipulated by using the *pvm_mkbuf()* primitive for creating buffers, *pvm_freebuf()* primitive for deleting buffers, *pvm_getsbuf()* and *pvm_setsbuf()* primitives for getting and setting default send buffer respectively, *pvm_getrbuf()* and *pvm_setrbuf()* primitives for getting and setting the default receive buffer, and the *pvm_bufinfo()* primitive for obtaining the statistics (message size, tag, and source task) on a particular buffer [Geist et al 1994].

3.2.3 Group Communication

For some parallel processing applications, it is more natural to think of the tasks that are being used for the application as a group of tasks, instead of individual tasks. There are also tasks where it is more appropriate to refer to the tasks used for the application as a number between *0* and *p - 1* (where *p* is the number of tasks involved in the application). For these reasons, PVM provides the user with process groups and group communication facilities.

Groups within PVM are identified by a null terminated string. Any task can join or leave a group at any time without having to inform the other members of the group. There is no limit imposed on the number of tasks that may be in a group, nor is there a limit on the number of groups that a task may be a member of. Upon joining a group, a task is allocated an instance number for that particular group. The instance numbers are allocated from zero for the first task enrolled in the group, and count up. The instance number a task has been allocated which will not change during the period of membership. However if the task were to leave and then re-join the group, there is no guarantee that the same instance number will be allocated [Geist et al 1994].

The mechanisms for task groups have been built on top of the core PVM primitives and are supported by the addition of another server known as the “group server”. The group server is responsible for:

(1) managing the list of tasks in each group, ensuring that no task is enrolled in a group twice;
(2) coordinating the *pvm_barrier()* operation;
(3) mapping any *pvm_bcast()* calls to the appropriate *pvm_mcast()* call (Section 3.2.1);
(4) the selection of coordinators for the *pvm_reduce()* operation.

The primitives that are provided for group management are for joining a group (*pvm_joingroup()*), leaving a group (*pvm_lvgroup()*), finding out how many tasks are registered in the group (*pvm_gsize()*), to match a TID and group name pair to that task’s instance number (*pvm_getinst()*), and to match an instance number and group name pair to that tasks TID (*pvm_tid()*). Each of these operations is performed by the group server. Groups are created automatically upon the first call to the *pvm_joingroup()* primitive with a new group name. The data entries for a group are not destroyed by the server after the last task has left the group, but the first task to enrol in the group would receive an instance number of zero, and thus it will be functionally equivalent to a newly created group.

For group based communication, PVM provides a send operation (the *pvm_bcast()* primitive) that is mapped to the *pvm_mcast()* multicast send primitive, barrier synchronisation (the *pvm_barrier()* primitive), and a reduce operation (the *pvm_reduce()* primitive) with built-in support for global minimum, maximum, sum and product operations and also user-defined operations. The group send operation has been implemented by requesting a list of TIDs of the tasks that are in the group, and then using this list as the list of TIDs parameter of the *pvm_mcast()* primitive (see Section 3.2.1) to multicast the message to those tasks.

The barrier synchronisation operation is coordinated by the group server. Upon receipt of the *pvm_barrier()* call, the tasks send a message to the group server specifying the count of tasks that are to participate in the barrier. If the call is the start of a new barrier, the task count for the barrier is stored with the other group details, the calling task’s TID is added

to a queue of tasks waiting on the barrier, and the group server returns to waiting for messages to arrive. The calling task then calls a blocking receive to wait for the server to return a message indicating that the barrier has cleared. If it is not a new barrier, the count is checked against the count the barrier was initialised with (they must be equal or an error is returned), and the calling task's TID is added to the queue of tasks waiting on the barrier. Once the number of tasks in the queue waiting for the barrier is equal to the required count, the barrier count is cleared and a message is sent to each of the tasks waiting for the barrier to allow them to continue.

The reduce operation is used for the application of a non-associative mathematical function to sets of data spread across each of the tasks in the group, with the result placed in the memory of the "root task" (specified by the user). The operation is performed by the tasks themselves, with coordination provided by the group server. The group server selects a task on each workstation with group members to act as a coordinator for the reduce operation. The coordinators are responsible for the collection of data from the other group members on the workstation, applying the reduce operation on the data, and sending summary information to the root task.

The reduce operation is implemented by the calling tasks first requesting from the group server the coordinator for this workstation, the number of tasks on the current workstation (if the calling task is to be the coordinator for this workstation), and the number of workstations in the virtual machine that currently have tasks that are group members (if the calling tasks is the root task). If a task is a coordinating task, it first receives the data from the other tasks in the group on the workstation and applies the specified mathematical function to the data as it is received. Once all the data from other tasks on the workstation has been received and processed, the result is sent to the root task. The root task may or may not be a coordinator for the workstation it is located upon, and if so first receives and processes the data from the other tasks in the group on the workstation. The root task then receives the rest of the data from the other coordinator tasks, again applying the reduce operation to the data as it arrives.

Section 5.7 of [Geist et al 1994] claims that the *pvm_reduce()* operation is a non-blocking call. However it can be seen that in the case of the tasks that are deemed to be "coordinators" or the "root" task, the tasks must block until all the data has been received from all the other tasks/coordinators. Thus the *pvm_reduce()* call is not always a non-blocking call.

3.3 The Task Identification Number

PVM tasks are identified by a 32-bit integer known as the task identification number (TID). The TID identifies tasks as targets for task management and as a communication end point [Geist et al 1994]. The TID is composed of a one bit flag to indicate whether a server is being addressed; a one bit flag to indicate whether a group is being addressed; a 12-bit serial number to indicate the location (workstation) of a task or where a group is being maintained; and the remaining 18-bits are used by the individual PVM servers to identify individual tasks or groups under their control.

The PVM server is responsible for the allocation of TIDs to new tasks during the process creation operation, and the mapping of TIDs to processes during process management operations, and to a task's PVM communications socket during communication operations [Manchek 1994].

3.4 Event Notification

PVM tasks using the *pvm_notify()* primitive are able to request the notification of the addition of workstations to the virtual machine, the removal of a specific workstation(s) from the virtual machine, or the termination of a specific task(s). This notification mechanism is provided by the PVM servers. After the addition of a workstation to the virtual machine, a message is sent to the PVM servers to inform them of the change. Any tasks that have requested the notification of an addition to the virtual machine are then sent a message indicating the TIDs of the new server. The notification of the removal of a workstation from the virtual machine is performed both when the server receives a message indicating that a server is shutting down, and when contact is lost with the server. The notification of the exiting of a task is accomplished by forwarding the notification request to the server local to the task that is to be monitored. A notification message is also sent by the server local to the requesting task if the remote server has been forcefully removed from the virtual machine after the master server had lost contact with it.

PVM also provides the *pvm_sendsig()* primitive which allows for the sending of signals (Unix signals) to other tasks. This has been achieved by requesting the server local to the target task to send the specified signal.

4 PVM on RHODOS

This section presents the design of the PVM tool for the RHODOS distributed operating system. The layout of this section is the same as for Section 3 (PVM on Unix), in order to allow the reader to easily relate back to the design of the Unix version.

The design of task management for the RHODOS PVM is presented in Section 4.1. The design of PVM interprocess communication for RHODOS is presented in Section 4.2. The new model for identifying tasks in RHODOS is presented in Section 4.3. Finally, the design of PVM fault detection mechanisms for the RHODOS system is presented in Section 4.4.

4.1 Task Management

The RHODOS operating system supplies transparent process creation and termination supported by a Global Scheduler employing static allocation (supported by remote process creation [Hobbs and Goscinski 1996]) and dynamic load balancing (supported by process migration [De Paoli and Goscinski 1997]) [Hobbs 1995].

RHODOS provides the *process_create()* primitive for creating processes from a program stored on disk, and the *process_twin()* primitive for duplicating a process already executing. Process termination can be accomplished by the process itself by using the *process_exit()* primitive (functionally equivalent to the traditional C language *exit()* primitive), or can be forcefully terminated by another process using the *process_kill()* primitive [Hobbs and Goscinski 1996].

PVM provides transparent task creation and termination to the user, and uses static allocation for load balancing (Section 3.1). This functionality is already a feature of the RHODOS operating system, hence there is no need for any additional functionality to be introduced. Only a simple mapping of the parameters of the PVM primitives to the equivalent parameters for the RHODOS primitives will be required.

The PVM *pvm_spawn()* (Block 2) primitive requires six parameters:

- *file* - the location of the program on disk to be run;
- *argv* - the command line arguments to be passed to the new child processes;
- *flags* - for specifying a specific workstation or architecture for the new process, to start the process using a debugger, to make the child process output trace information, to start the process on a massively parallel computer front end, or to use the complement of group of candidate workstations (see Section 3.1);
- *where* - used to specify the workstation or architecture used in *flags*;
- *count* - the number of instances of the process that should be created;
- and *tids* - a pointer to an array of integers to be populated with the task identification numbers of the new tasks.

```
int     pvm_spawn(
                char        *file,
                char        **argv,
                int         flags,
                char        *where,
                int         count,
                int         *tids
        );
```

Block 1: The PVM *pvm_spawn()* primitive

The RHODOS *process_create()* (Block 2) primitive requires five parameters:

- *proc_name* - the location of the program on disk to be run;
- *proc_num* - the number of instances of the process that should be created;
- *proc_snames* - a pointer to an array of SNames (discussed in Section 4.2.3) to be populated with the process SNames of the new child processes;
- *proc_arg* - the command line arguments to be passed to the new child processes;
- and *proc_env* - the environment for the child processes.

```
int32_t     process_create(
                    uint8_t     *proc_name,
                    int32_t     proc_num,
                    SNAME       proc_snames[],
                    uint8_t     *proc_arg[],
                    uint8_t     *proc_env[]
            );
```

Block 2: The RHODOS *process_create()* primitive

It can be seen that the *pvm_spawn* parameters *file*, *argv*, and *count* map directly to the *proc_name*, *proc_arg*, and *proc_num* parameters. The requirements of the *flags* parameter is reduced because in RHODOS there is no addressing of individual workstations, there are only homogeneous architectures, and there are no massively parallel computers. This reduces the

flags parameter to starting the process using a debugger, and having the child process output trace information, also eliminating the requirement of the *where* parameter. Due to RHODOS' relative youth, no debugging tools are available. This leaves the tracing flag, which can be passed to the child process as additional configuration information (see Section 4.3).

The *tids* parameter cannot be mapped directly to the RHODOS SName's parameter for reasons discussed in Section 4.3. However, new child tasks in RHODOS will return their identification numbers to the parent process during their initialisation (and before the *pvm_spawn()* primitive returns), and thus the array of integers will be able to be filled then.

4.2 Interprocess Communication

The RHODOS operating system is based on the concept of microkernel and the client-server model. A good interprocess communication facility is thus an essential component of the operating system. A review of PVM support for interprocess communication under Unix was presented in Section 3.2. The impact of the RHODOS interprocess communication facility on PVM, and the design of the PVM interprocess communication support for RHODOS is presented in this section.

4.2.1 Message Passing

The goals during the selection of communication protocols for PVM on Unix were to provide a guaranteed delivery to minimise the workload of the tasks to deliver messages, and to use a datagram service wherever possible to reduce the demands on system and network resources and to increase scalability. For these reasons, the TCP protocol was selected for communication between tasks and servers to minimise the workload placed on tasks, and the UDP protocol was selected for communication between servers to increase scalability.

These protocols cannot be used for the RHODOS operating system because it does not support either of the TCP or UDP protocols. However RHODOS does provide a high performance reliable datagram protocol known as the RHODOS Reliable Datagram Protocol (RRDP) [Joyce et al 1995]. This protocol places no limits on the size of a message, and any necessary fragmentation is handled automatically.

The default path for communication under the Unix PVM was to route messages through the server processes. The elimination of the server from the RHODOS PVM model in Section 2 means that this is no longer possible. However the use of the RRDP protocol under RHODOS also means that it is no longer necessary. Messages will therefore be routed directly between tasks, and this will not introduce any extra overhead to the tasks because the RRDP protocol provides a reliable service, and as a connectionless based service it will scale well.

The removal of the server and new routing for messages also allows the headers that were used in Unix versions of PVM to be modified somewhat. Three headers were used in the Unix PVM: a message header; a header for task-server communication; and a header for server-server communication. The removal of the server eliminates the need for the second and third headers, leaving only the message header.

Of the four fields in the message header (message tag, encoding type, wait context ID and checksum), only the message tag will be required. The encoding type will not be required because only the raw encoding type is now used (see Section 4.2.2). The wait context ID is

used by servers to keep track of tasks blocking on operations so that if a workstation were to be removed from the virtual machine, an error message could then be returned to the calling processes. The removal of the server as an intermediary means that this is no longer possible. The checksum field in the Unix version of PVM was not being used (only reserved for future use) and with RRDP providing a reliable service no use can be seen for such a field.

4.2.2 Buffer Management

As discussed in Section 3.2.2, PVM provides buffers for the storage and manipulation of data that is to be passed between tasks. The data is stored in the PVM buffers based on the level of encoding selected (see Section 3.2.2 for more information). RHODOS is composed only of homogeneous workstations and thus the use of XDR would be an unwarranted overhead. For these reasons, XDR encoding will not be used, and raw encoding will now be the default encoding in order to keep the syntax of the primitives equal to the Unix version of PVM.

The *PvmDataInPlace* encoding level presents the problem of not having an existing model to refer to for its equivalent implementation under RHODOS. However, the RHODOS interprocess communication primitives support reading from multiple data buffers during a send operation. Using this feature eliminates the additional copy operation which is the goal of this encoding level and thus is an appropriate solution.

4.2.3 Group Communication

The naming of groups under RHODOS, as with other objects in RHODOS, is split into two levels: the User Name and the System Name (SName). The User Name is an attributed name, designed to be user friendly; and the System Name, a data structure (shown in Block 3) that is more appropriate for system use. The primitives that RHODOS provides for group communication include (among others): creation of a group (*create_group()*), joining a group (*join_group()*), leaving a group (*leave_group()*), destroying a group (*destroy_group()*), and for obtaining information about a group (*group_info()*).

```
typedef struct sname {
    uint16_t    sn_signature;       /* The user's id */
    uint8_t     sn_type;            /* The object's type */
    uint8_t     sn_copy;            /* Which object copy */
    uint32_t    sn_origin;          /* The object's origin */
    uint32_t    sn_object;          /* The object's name */
    uint32_t    sn_access_rights;   /* The access rights */
    uint32_t    sn_checksum;        /* Checksum */
    } SNAME;
```

Block 3: The RHODOS System Name (SName)

The group server used in the Unix PVM manages the list of tasks in each group, coordinates the barrier operation, maps *pvm_bcast()* calls to the appropriate *pvm_mcast()* call, and selects the coordinators for the *pvm_reduce()* operation (Section 3.2.3). RHODOS provides process groups, thus the first of these requirements is no longer present. The second

of these requirements, to coordinate the barrier operation (see Section 3.2.3), can be implemented by using the RHODOS barrier synchronisation facility [Silcock et al 1997]. The third of these operations, the mapping of *pvm_bcast()* calls to *pvm_mcast()* calls, is trivial with the *pvm_mcast()* primitive supported by the RHODOS group communication service. The last of these requirements, to select the coordinators for the *pvm_reduce()* operation, is made redundant by the nature of a distributed operating system (no specific workstations), and thus the only coordinator that is required is the root instance (nominated by the programmer).

To be able to implement PVM group communications without using a server, certain information needs to be accessible by all of the tasks, specifically: which instance number of a a task in a group is mapped to which task ID (and vice-versa), and which instance/task is coordinating the current barrier synchronisation operation (if any). Under RHODOS, any object can be given multiple (attributed) names [Goscinski and Haddock 1994]. This information can thus be made available by modifying the name of the group object, storing this information as one of the attributes.

4.3 The Task Identification Number

PVM uses task identification numbers (TIDs) for identifying tasks for task management, and as destinations for interprocess communication. RHODOS uses SNames to identify processes for process management and to identify ports as destinations for interprocess communication. The simplest solution would be to simply swap the TIDs with SNames. However there are differences between the two that cause problems with this solution. The differences between TIDs and SNames are:

- the TID is a 32-bit integer and the SName is a data structure (see Block 3);
- the TID identifies the task for both task management and for interprocess communication whereas the SName cannot identify both a process and a communications port.

The first of these differences would mean that swapping TIDs with SNames would cause the RHODOS PVM primitives to be syntactically incompatible with the primitives of the Unix PVM, which is to be avoided. In order to achieve both syntax compatibility and be able to locate SNames for both the PVM task's process SName and port SName, some way must be found to map a 32-bit integer to the two different SNames.

A solution to the problem of how to map a 32-bit integer to two SNames can be found upon examination of the RHODOS naming facility. RHODOS uses attributed names to provide a user-friendly means of addressing objects within the RHODOS system (processes, ports, peripherals, etc). Attributed naming involves using a set of properties (attributes) to describe an object. These attributes of an object can be divided into three categories:

- Naming oriented attributes - these attributes are used to support user-oriented naming or its associated operations, and can be added, updated or deleted at any time by the owner or supervisor;
- System oriented attributes - these attributes are used for the management of the objects (e.g. the size of a particular file) and are maintained by the managing system servers;
- Dynamic attributes - these are used to describe the status of a particular object (e.g. print jobs waiting for a printer).

Through the use of naming oriented attributes, a task's TID can be added as an attribute to the process SName and the port SName. However the problem still remains of how to allocate a unique TID.

The simplest solution would be to allocate the first available TID to a task. However to determine the first available TID, in a worst-case scenario, a task would need to query the name server as many times as there are tasks. A better solution is to use the information in the SName (which is already unique) to build the TID. The fields of the SName that uniquely identify an object are: the object's type (`sn_type`, 8 bits), origin (`sn_origin`, 32 bits) and the object's number (`sn_object`, 32 bits). It can be seen that a single 32-bit number cannot represent every possible SName. However the values of the SNAME can be used to generate a starting value for the search for a free TID.

Before determining how to create the starting value, we first need to determine whether to use the SName of the process, or the SName of the port used for PVM communications. The object number of an SName is incremented each time a new object is created. Using the process SName to generate the starting value would cause the distribution of used TID values to be heavily grouped. In a worst case scenario this would again require the same number of queries to the name server as there are processes in existence. However, each process in RHODOS is automatically allocated a "hidden port" for remote procedure calls (RPCs), and a "unique port" which was once used for identifying processes (a process SName is used now). Thus if the SName of the port used for PVM communication is used, at least two unused TID values will initially exist for every used TID. The selection of the port SName also makes the type of object to be a constant. A starting value can now be constructed from the origin and object number fields.

The origin and object number fields are both 32-bit integers. The origin field represents the network address, each 8 bits representing one number of the IP address. RHODOS currently exists only within a single subnet, thus the IP address can be divided into a 24 bit static part and 8 bit dynamic part. Using the dynamic 8 bits of the object's origin in the construction of the TID leaving 24 bits which can be used to represent the 24 least significant bits of the object number. Hence we now have a starting value.

Using the SName of the port used for PVM communication as the basis for a task's TID presents an additional problem in that the SName port is not known until the task has created the port. This will require the *pvm_spawn()* task creation primitive to wait until the port is created. Under Unix the PVM server generates and returns the TID for each task during the process creation operation. Under RHODOS the TID will need to be generated by a task during initialisation and passed back to the parent task (if it exists).

Under Unix, the port used for PVM communication (a Unix socket) is not created until the first PVM primitive is invoked. This event may occur early, late or even never in the lifetime of a task, hence this model cannot be used for the RHODOS PVM. The solution to this problem lies with the header that is attached to each executable file. For the C language this header typically performs various initialisation operations (such as the queuing of command line arguments), and then calls the *main()* function. This header can be modified so that a special initialisation function for PVM is called before the *main()* function, which will then create the port for PVM communication, register the required attributes with the name server, return the TID to the parent task and receive any special configuration information from the parent task. This guarantees that the TID will be returned to the parent task immediately after the task is created, thus allowing the parent process to continue its execution.

4.4 Event Notification

As discussed in Section 3.4, PVM tasks can request notification for the addition of workstations to the virtual machine, the removal of workstations from the virtual machine, and the termination of another task. Due to the nature of a distributed operating system, the first two of these are eliminated. However the notification of the termination of a task is yet to be addressed.

RHODOS currently has support for the detection of a child process exiting in the *process_wait()* call, but there is currently no support for notifying any process that another process has terminated. However the RHODOS Remote Execution (REX) Manager is responsible for coordinating all process management operations including the termination of processes [Hobbs and Goscinski 1996] and it is not difficult to extend this facility to provide such a notification service.

The *pvm_sendsig()* primitive cannot be supported by the RHODOS operating system at this time because it does not support Unix signals. However the *pvm_kill()* primitive on RHODOS will not use the *pvm_sendsig()* primitive (as in Unix, see Section 3.1), and the use of Unix signals for communication can be accomplished using the usual PVM interprocess communication primitives. Thus, the *pvm_sendsig()* primitive is not required.

5 Implementation and Testing

The RHODOS is characterised by a layered architecture consisting of the microkernel (nucleus), kernel servers, system servers, and user processes [De Paoli et al 1995] (the layered architecture and relevant kernel and system servers are shown in Figure 4). The microkernel provides the basic services required in order to support processes, and any remaining services are handled by the kernel and system servers. The kernel and system servers are privileged processes, with kernel servers capable of modifying the structures maintained within the nucleus. The RHODOS implementation of PVM is located completely within the user processes layer, as no PVM specific support is required from the operating system.

The testing of this and other implementations of PVM involves the application of a test suite [Casanova et al 1995] which consists of a graphical Tcl/Tk front-end to a series of functions that test the functionality of each of the PVM primitives on any particular implementation of PVM. This cannot be used directly on RHODOS because it currently does not have support for Tcl/Tk. However the functions were easy to extract from the program and run on the RHODOS system.

Each of the tests have been run, and completed successfully, except for:

- tests that involve the manipulation or querying of the configuration of the virtual machine (redundant, see Section 2);
- tests of the *pvm_sendsig()* primitive (unsupported, see Section 4.4);

The application of these tests indicates that the implementation of the individual primitives is correct, and that the interface is consistent with the Unix version of PVM.

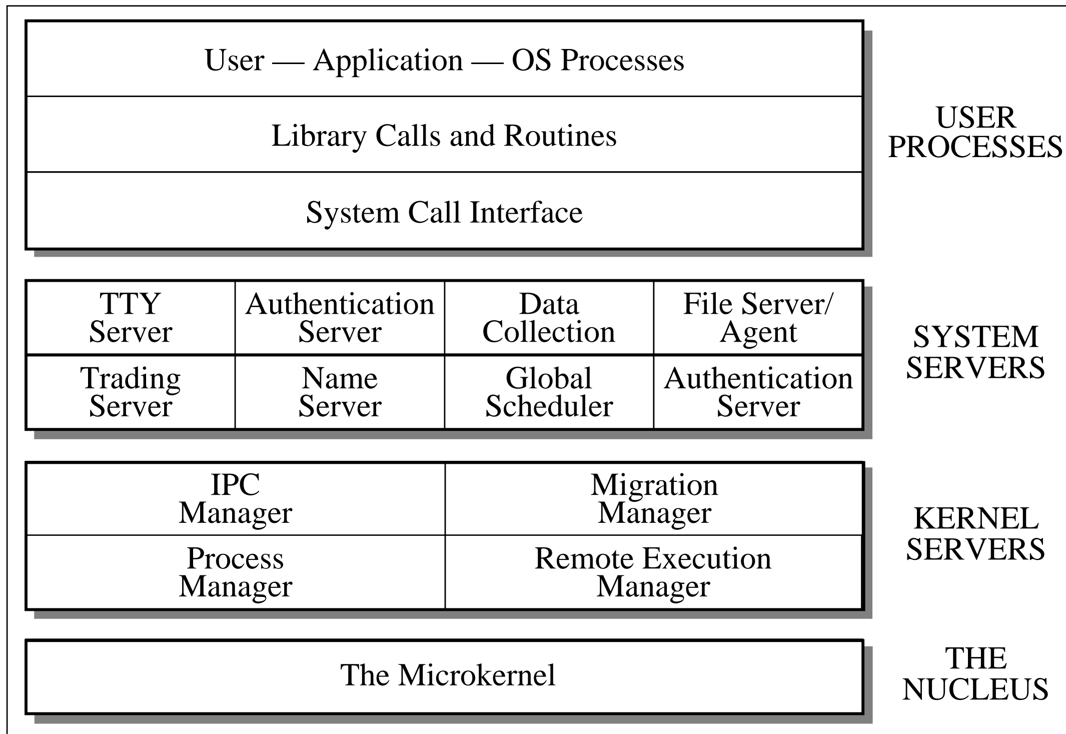

Figure 4: Process Layers in RHODOS

6 Conclusions

This report has described the design and implementation of the PVM tool for the RHODOS distributed operating system which provides an identical syntax to that provided by the Unix version (apart from the loss of workstation- and Unix-specific primitives). A recognisable set of interprocess communication primitives is now available for programmers that are familiar with PVM. Any existing PVM programs are now be able to be compiled for the RHODOS system without modification. Hence the range of software available for testing and performance measurement of the RHODOS system has been expanded.

The replacement of the Unix environment with RHODOS as the base for the PVM package has allowed the functionality provided by the package to be greatly simplified. Furthermore, the addition of RHODOS specific features such as transparent dynamic load balancing will also help to improve the performance of PVM applications.

Acknowledgements

The author would like to acknowledge the assistance of Philip Joyce for his assistance in helping me better understand the RHODOS Group Communication and Naming services and Greg Wickham for his contribution of the model used for PVM task initialisation.

7 Bibliography

[Beguelin et al 1995]
A. Beguelin, J. Dongarra, G. Geist, W. Jiang, R. Manchek, B. Moore, and V. Sunderam, *"PVM - Parallel Virtual Machine System Version 3"*, *pvm_intro(1)* manual page, PVM v3.3.11 distribution, 1995.

[Casanova et al 1995]
H. Casanova, J. Dongarra, P. Mucci, *"A Test Suite for PVM"*, Technical Report UT-CS-95-276, University of Tennessee, March 1995.

[De Paoli and Goscinski 1997]
D. De Paoli and A. Goscinski, *"The RHODOS Migration Facility"*, to appear in the *Journal of Systems and Software*, late 1997.

[De Paoli et al 1995]
D. De Paoli, A. Goscinski, M. Hobbs, and G. Wickham, *"The RHODOS Microkernel, Kernel Servers and Their Cooperation"*, in Proceedings of the IEEE First ICA^{3}PP, Brisbane, Australia, 19-21 April 1995.

[Geist et al 1994]
G. Geist, A. Beguelin, J. Dongarra, W. Jiang, and R. Manchek, *"PVM: Parallel Virtual Machine - A User's Guide and Tutorial for Networked Parallel Computing"*, The MIT Press, 1994.

[Gerrity et al 1990]
G. Gerrity, A. Goscinski, J. Indulska, W. Toomey, and W. Zhu, *"The RHODOS Distributed Operating System"*, Technical Report CS90/4, Department of Computer Science, University College, Australian Defence Force Academy, University of New South Wales, February 1990.

[Goscinski 1991]
A. Goscinski, *"Distributed Operating Systems: The Logical Design"*, Addison-Wesley, 1991.

[Goscinski et al 1994]
A. Goscinski, M. Hobbs, P. Joyce, and G. Wickham, *"Message Passing and RPC-based Interprocess Communication Mechanisms in the RHODOS Microkernel*"*, Technical Report TR C94/09, School of Computing and Mathematics, Deakin University, May 1994.

[Goscinski and Haddock 1994]
A. Goscinski and A. Haddock, *"A Naming and Trading Facility for a Distributed System"*, The Australian Computer Journal, Volume 26(2), pp50-65, May 1994.

[Hobbs 1995]
M. Hobbs, *"Global Scheduling on Distributed Systems: the RHODOS Case"*, Technical Report TR C95/29, School of Computing and Mathematics, Deakin University, August 1995.

[Hobbs and Goscinski 1996]
M. Hobbs and A. Goscinski, *"A Remote Process Creation and Execution Facility Supporting Parallel Execution on Distributed Systems"*, in Proceedings of the IEEE Second ICA^{3}PP, Singapore, June 11-13 1996.

[Joyce et al 1995]
P. Joyce, M. Hobbs, A. Goscinski, and D. De Paoli, *"Implementation and Performance of the Interprocess Communications Facility in RHODOS"*, in Proceedings of the IEEE SICON/ICE, Singapore, July 3-7 1995.

[Manchek 1994]
R. Manchek, *"Design and Implementation of PVM Version 3"*, a thesis presented for the degree of Master of Science, also available as Technical Report UT-CS-94-232, Department of Computer Science, University of Tennessee, May 1994.

[PVM 1996]
PVM v3.3.11 source code, released May 1996.

[Rough 1996]
J. Rough, *"The Development of a Parallel Programming Environment for RHODOS based on PVM"*, a thesis presented for the degree of Bachelor of Science (Honours), School of Computing and Mathematics, Deakin University, November 1996.

[Silcock et al 1997]
J. Silcock, A. Goscinski, *"Invalidation-Based Distributed Shared Memory as a Integral Part of the RHODOS Distributed Operating System"*, submitted to Euro-PDS97, 1997.

[Sun 1987]
Sun Microsystems, *"XDR: External Data Representation Standard"*, Internet RFC 1014, June 1987.

[Sunderam 1990]
V. Sunderam, *"PVM: A Framework for Parallel Distributed Computing*"*, appeared in *Concurrency: Practice and Experience*, Volume 2(4), pp315-339, December 1990.

[Wang and Goscinski 1992a]
M. Wang and A Goscinski, *"The Logical Design of an Authentication Service for RHODOS*"*, Technical Report TR C92/6, School of Computing and Mathematics, Deakin University, September 1992.

[Wang and Goscinski 1992b]
M. Wang and A. Goscinski, *"The Development and Testing of an Authentication Service for RHODOS*"*, Technical Report TR C92/7, School of Computing and Mathematics, Deakin University, September 1992.