DYNAMICPVM:
TASK MIGRATION in PVM

by

Leen Dikken

Report no. ICS/155.1

Shell Research
Amsterdam

November 1993

CONTENTS

PREFACE

This report contains the results of the graduation project for the Computer Science course at the University of Amsterdam. The work for the project was done within the Information & Communication Service department at Shell Research, Amsterdam (Koninklijke/Shell-Laboratorium, Amsterdam), where I have been employed since 1985. Professor Dr. L.O. Hertzberger, Dr. P.M.A. Sloot, Dr. P. van Emde Boas (University of Amsterdam) and Dr. P.A.J. van Deurzen (Shell Research, Amsterdam) were the supervisors.

The initial goal of the project, defined in the spring of 1992, was to explore to what extent distributed systems could contribute to a more efficient use of the networked workstations installed at KSLA (Koninklijke/Shell-Laboratorium, Amsterdam). Follows some initial problems in defining the scope of the project, the idea was born to use PVM and Condor - two available packages supporting distributed computing - to run an existing scientific application. This would pave the way for a test with the DCE (Distributed Computing Environment) products from OSF (Open Software Foundation). However, partially due to personnel movement the test with the scientific application had to be postponed.

The results of this study are:
- a rudimentary form of DYNAMICPVM which combines PVM and Condor,
- a better understanding of the impact the introduction of distributed systems will have on the KSLA organisation.

I would like to thank Professor Hertzberger, who put me on the right track, Peter Sloot, who kept me on the track and Paul van Deurzen who patiently guided me past many obstacles during the project and had his interrupt bit turned off for me.

A. de Beer, Th. Breet, R.P. Bosma and G. Noordenbos have provided me royally, on behalf of Shell, with all the facilities I required for this course and for this project. This has made it possible for me to work, study and have a social life, which I greatly appreciate. Nico Trompé, Daan Loyens and Wouter Meiring I would like to thank for the constructive interest they have shown in my work.

Finally, I am most grateful to the PVM/Condor working group: René van Dantzig, Xander Evers, Frank van der Linden, Peter Trenning and Joep Vesseur for their contribution to the project. I should like to mention in particular the discussion with René van Dantzig and Miron Livny, in which they gave me the right hint for the design of the 'task migration protocol', which is an essential part of DYNAMICPVM.

And now I'm looking forward to spending more time with Nina and our two daughters, Elise and Marloes. They deserve it.

Leen Dikken, Amsterdam, November 1993

1 INTRODUCTION

Over the last ten years, the KSLA (Koninklijke/Shell-Laboratorium, Amsterdam) approach to scientific computing has changed dramatically. With the installation of a local area network (LAN) followed by the introduction of workstations at KSLA, a shift has taken place from centralised computing towards decentralised computing. Today, the scientific staff at KSLA have access to a wide range of computer services, most of them on UNIX workstations, via this local area network. The aggregate computing power has increased from 40 MFLOPS to 400 MFLOPS (Million Floating-point Operations Per Second). This significant expansion reflects the mutual interaction between the demand from the scientific community for more computing capacity and the development of computer technology.

With respect to these services, we observe a trend towards differentiation: file servers, specialised systems for communication, computing servers (Numerical Intensive Computing), database servers, etc. File servers contain and control access to a file system shared by a group of workstations in a network. Together they form a cluster. This allows transparent access to files from any of the workstations in the cluster. At KSLA, workstations which form a logical unit (e.g. which belong to one department) are clustered by means of NFS (Network File System).

In the current situation the different services can be accessed via the network but each user application is handled by a single computer system. To exploit the specific property of a service, users have to perform specific actions. This implies that when a user wants to run an application a workstation - probably the least heavily loaded - has to be selected from the local cluster and execution started. As a result the potential of the computer systems in the network cannot be utilised to its full extent, while at the same time individual machines might be overloaded [Eager84] and [Artsy86].

Distributed systems form an emerging technology which enables autonomous computer systems in a network to cooperate, yielding a better overall performance than any of the individual systems can offer [Lindh91]. Individual users profit since either programs will be completed earlier or else programs with additional functionality will be completed in the same amount of time. In both cases the distribution of the application over multiple computers is hidden from the user. From a capacity planning perspective, distributed computing may be advantageous since - in the short term - the increasing demand for computing power can be fulfilled by utilising the potential of the installed base.

PVM (Parallel Virtual Machine) and **Condor** (for batch job scheduling) are two examples of systems already available, which can be characterised as distributed systems. Both PVM and Condor can be executed as user processes and hence do not require modifications of the UNIX kernel, which increases applicability. Both systems - already in wide spread use - are ongoing developments of the Emory University - Atlanta and the University of Wisconsin - Madison respectively.

PVM offers good facilities for parallel distributed computing but a poor scheduling facility for the distribution of the workload. In contrast, Condor makes use of advanced scheduling, facilitated by a job migration mechanism, but does not support parallel distributed computing. Merging the PVM and Condor facilities offers an opportunity to improve control over tasks in parallel distributed computations. In this report the construction of a new system out of PVM and Condor is described, which we called: DYNAMICPVM.

The report is organised as follows. In the next section it is explained how optimisation by means of distributed systems can be accomplished. In section three the concepts of DYNAMICPVM are given. In this section the emerging question: should PVM be ported to Condor or vice versa, is addressed. The design issues of DYNAMICPVM are discussed in section four. The next section lists the limitations imposed by DYNAMICPVM on PVM-applications. Implementation details are described in section six. Preliminary results and the status of DYNAMICPVM are given in section seven. A users manual which explains the steps to be taken for running PVM tasks under DYNAMICPVM can be found in section eight. In section nine an attempt is made to indicate a direction for future work. Finally, a discussion and the conclusions are given in section ten.

2 DISTRIBUTED SYSTEMS AND PERFORMANCE IMPROVEMENT

Leading researchers and developers in the field of distributed systems claim that distributed systems, in comparison with conventional systems, will lead to performance improvement. Mullender: "*... a distributed system is more powerful than a conventional, centralized one ...*" [Mullender89]. Lindh: "*The idea with Distributed Systems is to combine different products (hardware and software) with different strengths, to achieve a combination which is better then any of the individual components.*" [Lindh91].

In the following sections the 'what' and 'why' of distributed systems are briefly reviewed and the 'how' with respect to the above mentioned statement will be given. Furthermore, examples of three existing systems (OSF/DCE, PVM and Condor) are given. At the end of this section, we introduce a system in which PVM and Condor are combined: DYNAMICPVM.

2.1 Distributed systems

The topic of distributed systems is relatively new in the area of Information and Computing technology. Although there is consensus about what distributed systems are, there is not yet a widely accepted definition. Mullender identified two fundamental properties of distributed systems: **fault tolerance** and the possibility to use **parallelism**. Instead of an exact definition he stated a list of symptoms which a system should exhibit to be qualified as a distributed system:

- **multiple processing elements** (each containing at least a CPU and memory)
- **interconnection hardware** (which allows parallel processing)
- **processing elements fail independently** (no single point of failure)
- **share state** (in order to recover from failure of peer nodes)

The most important (and most tangible) motivation for building a distributed system is the performance/cost ratio [Coulouris88], [Mullender89] and [Lindh91]. While computers are getting cheaper and cheaper, communication costs are going down much less rapidly than computer costs. However, man-machine interfaces nowadays are graphics based and highly interactive, demanding a high communication speed between computer and screen. Mullender concludes:

> "*These effects make distributed systems not only economic, but necessary. No centralized computer could give the required number of cycles to its customers and the cost of the network technology that gets the required number of bits per second out of the user's screen is prohibitive.*"

This will become even more apparent when workstations in (near) future will have a high-resolution colour display, and voice and real-time video input and output devices.

Other frequently used motivations for building distributed systems are expansibility and availability. Expansibility in distributed systems means that they are capable of incremental growth by adding components one at a time. The ability to recover from (single-point) failures, makes distributed systems to have a high level of availability.

2.2 Performance improvement

As stated before, an important aspect of distributed systems is that it offers an opportunity for performance improvement. This will be realised by means of **load managing**: the decomposition and distribution of the workload.

Performance can be expressed in terms of user expectations like: response time (with interactive jobs), turnaround time (with batch jobs) and fairness, and in terms of system metrics like: efficiency (of CPU usage) and throughput. The decomposition and distribution of the load are the key factors for the level of the overall performance.

One of the symptoms of distributed systems is that they support parallel distributed computations. Here, several processors will cooperate - share load - to run one application fast. For this, the application is **decomposed** into smaller tasks which have to be distributed over the distinct processors [Fox88] and [Nutt92]. However, the actual speedup (σ) which can be obtained, is limited by 'Amdahl's Law':

$$\sigma = \frac{t_s}{t_p}$$

with

$$t_p = \alpha \cdot t_s + (1 - \alpha) \frac{t_s}{n}$$

where

σ = speedup

t_s = calculation time: sequential case

t_p = calculation time: parallel case

α = sequential fraction in calculation

n = number of processors

E.g.: for a sequence of 100 operations from which 80 can be executed in parallel on 80 processors (t_s = 100, α = 0.2, n = 80), t_p will be 21 (100 on a uniprocessor) and the speedup (σ) will be about 5 (not 80!). These formulae disregard performance degradation due to inter-process communication, load imbalances etc.

In distributed systems, **load distribution** is the task of the scheduler. For workstations in a network without a global job scheduler, users have to select a host for the execution of their jobs themselves. This can easily lead to severe load imbalances, which in turn will lead to performance degradation. Scheduling in distributed systems aims to attain optimal utilisation of the available computational resources. Furthermore, parallel distributed computing is even not feasible without a global scheduler.

2.3 Scheduling in distributed systems

The salient aspects of scheduling in distributed systems are discussed in the following sections. For an overview of this complex subject, see [Evers92].

2.3.1 Load sharing versus load balancing

Two distinct strategies which are applied to optimise the distribution of the workload in distributed systems, are **load sharing** and **load balancing** [Krueger87] and [Artsy88]. Load sharing algorithms prevent jobs waiting for execution while there are idle processors and load balancing algorithms attempt to distribute the total workload equally among the processors. Load balancing has a significantly broader performance objective than load sharing, because nodes which are not idle may be scheduled also. Hence, the performance objective of load balancing can be viewed as a superset of that of load sharing.

Load distribution strategies have been studied by many researcher, resulting in proposals for and evaluations of algorithms for scheduling: see [Eager84] and [Wang85] for load sharing, and [Hać85], [Huang86], [Kara92] and [Xu93] for load balancing.

2.3.2 Static versus dynamic scheduling policies

Load distribution algorithms implement a **static** or **dynamic** scheduling policy. A static scheduling policy is based on *a priori* knowledge of average behaviour (e.g. average execution time and interconnection requirements), while a dynamic scheduling policy reacts to changes in the system state. Static policies focus on initial placement, while dynamic policies require process migration facilities in addition. With **process migration**, a running process will be suspended and the information required to restart it on another host, will be saved. Next the process restart information will be moved to another host to enable the resumption of the execution of the process. Although process migration can be quite demanding with regard to I/O bandwidth, research on this topic showed that it will still lead to performance improvement [Krueger87], [Artsy86], [Eskicioğlu90] and [Suen92]. But also see [Eager88] for a dissentient opinion.

2.3.3 Which load distribution strategy?

It is recognised that the load distribution strategy chosen, plays a critical role in the overall performance of the distributed system [Wang85] and [Huang86]. It is also recognised that the choice of the local scheduling discipline may be of equally or more importance in determining performance [Krueger87]. Ideally, the load distribution strategy and the local scheduling discipline have to be chosen in relation with each other. However, the local scheduling discipline is adjusted to meet the needs of the users of the system. The following is of relevance for the choice for a load distribution strategy for clusters of multi-user workstations (as is the situation at KSLA).

A study of Livny and Melman [Livny82] showed that for a network of autonomous nodes, there is a large probability that a least one node is idle while tasks are queued at some other node. In another study, Mutka and Livny [Mutka87] analyzed the usage patterns of a group of **single-user** workstations in a network. It showed that the nodes were in use approximately 30% of the available time. For a network of **multi-user** workstations the probability that a node is idle is assumed to be smaller, as is the figure for availability. Despite this we observe that the percentage of the aggregate computing power of these multi-user workstations that remains unused, is still significantly.

In a multi-user environment, very little *a priori* knowledge is available about the resource needs of the processes. This will become even more apparent in the case of parallel distributed computations where tasks can be spawned dynamically. For this reason, a dynamic load distribution policy will be more suited than a static policy [Eager84]. Inherent to a dynamic load distribution policy, a process migration facility is required.

As stated, we assume the probability that a node is idle in a multi-user environment will be smaller

than in the above mentioned study of Livny and Melman. This and the fact that the performance objective of load balancing is significantly broader than that of load sharing, in combination with the choice for a dynamic load distribution policy, plead for **dynamic load balancing**.

Based on this strategy a scheduling algorithm has to be developed. The **scheduler** has to decide **which** process should be (re)allocated to **where** and **when**. For this, information is required about the current state of the distributed system (e.g. workload, queue length of the cooperating computer systems) and of the individual user processes in the system.

2.4 Examples of implementations

In the last view years, several distributed systems, each with different objectives, have been proposed and implemented. For an overview see [Coulouris88]. A recent development is OSF/DCE (Distributed Computing Environment) which became commercially available in the course of this year. Recent academic developments are PVM and Condor.

OSF/DCE promotes the Client/Server paradigm for distributed computing. Multiple servers may run on different machines. Servers are allocated to clients by means of a binding service. DCE also supports the execution of parallelised distributed applications.

PVM (Parallel Virtual Machine) is an example of a system which supports the execution of parallelised distributed applications. A cyclic allocation discipline is applied for the distribution of the cooperating tasks.

Condor is an example of a system in which dynamic load sharing - this implicates job migration - is applied to support the execution of longer running background jobs.

As stated, OSF/DCE is a new commercial product on the market. Both PVM and Condor are public domain software packets. PVM has become the *de facto* standard for parallel distributed computing and is widely used. Checkpointing in Condor - required for job migration - has proven to be robust for real-life applications. Both PVM and Condor are implemented as (a special kind of) user applications which does not require any modification of the UNIX kernel. This promotes the portability of the systems and makes it easier to explore the products. In contrast, DCE is a comprehensive product with advanced features which makes it a good candidate for the commercially market, but these features also make it more difficult to apply.

2.4.1 OSF/DCE

The OSF/DCE products provide the tools to create a distributed system [Lindh91] and [Rosenberry92]. The OSF/DCE products have been developed under the auspices of the Open Software Foundation. Major hardware and software vendors as well as users - among which Shell - participates in this development. DCE is the Shell recommended standard for distributed computing.

Some of the basic technologies employed by OSF/DCE that are required to build a distributed computing environment have already been available for several years: Remote Procedure Calls (RPC), eXternal Data Representation (XDR), and Naming services. Other facilities required by OSF/DCE have had to be developed for this purpose: Security (Kerberos/DES), dynamic binding (NCS) and Distributed File Systems (DFS). OSF/DCE does not include tools for parallelising existing user applications. This still has to be done manually.

2.4.2 PVM

Inherent to the objective of the PVM (Parallel Virtual Machine) development - to permit a collection of heterogeneous machines on a network to be viewed as a general-purpose concurrent computational resource [Sunderam90] - PVM supports sharing of the workload. For this, PVM provides primitives (implemented as library routines) to enable the decomposition of user application programs (forming PVM tasks) and facilities to distribute the PVM tasks over the machines in order to run them simultaneous. Hosts are selected cyclical, irrespective of the current work load. Once a task is allocated to a host it will remain there till it runs to completion. The task distribution facility is part of the PVM daemon process, referred to as the **pvmd**, which run on each of the machines in the PVM system. The pvmds facilitate also the required inter-task message transfer and system control.

Whether this approach of sharing the workload will also result in a significant improvement of the response (or turnaround) time of the executable depends on a number of factors. With regard to the decomposition of the program, the communication/calculation ratio should not be too high. If so, the advantage of running different PVM tasks simultaneously will be outweighed by the communication overhead. The same holds for the task-control/calculation ratio. It is up to the application developer to decide on and carry out the decomposition of the application program. With regard to the host machines for the PVM tasks, the workload of individual machines may be of great influence on the response time. A PVM task running on a heavy loaded machine may become the execution bottleneck for the application.

2.4.3 Condor

The Condor system - developed to provide a high quality of service in a highly utilised network of workstations for both light and heavy users [Litzkow88] - offers the ability for utilising the idle periods of interactively used workstations. This approach of dynamic load sharing is intended for longer running background jobs. On regular intervals (typically every 5 minutes) Condor attempts to allocates jobs (application programs) waiting for execution to workstations that have become idle. If interactive usage of an active host (workstations) is resumed the Condor job running in background will be killed and resubmitted to the queue of jobs waiting for execution. To prevent loss of work, Condor provides a checkpoint mechanism. Running jobs will be checkpointed on a fixed time basis (typically every 30 minutes). When a killed job is scheduled again, the latest checkpoint will be used to restart the process. This checkpoint mechanism guaranties normal termination of the application.

Condor allows remotely running jobs to execute system calls (e.g. for file handling) on their initiating machines by means of a Remote System Call (RSC) mechanism. For this, each stub in the standard system call library (libc.a) have had to be replaced by a stub in a special Condor system call library (libcondor.a).

User programs have to be linked with this special Condor library to enable checkpointing. An important limitation of Condor is that user programs should be single process jobs and thus parallel distributed computing is not supported.

2.5 DYNAMICPVM

The positive effect of the reduced elapsed time of a parallelised application as a result of sharing workload in PVM can be strongly reduced by an individual PVM task running on a heavy loaded machine as indicated above. With regard to Condor, jobs are migrated from host to host, supported

by the checkpoint mechanism, driven by the actual usage of the hosts. However, in contrast with PVM, Condor does not support IPC (InterProcess Communication). Both Condor and PVM focus on user applications and are to some extent complementary. Combining PVM and Condor offers the interesting possibility of **dynamic load balancing** of PVM tasks running in a network of computer systems. Under control of a scheduler, PVM tasks are migrated from hosts with a high load to hosts with low load.

These considerations are summarized in Table 1.

	Condor (Resource Allocation)	PVM (Parallel Virtual Machine)	**DYNAMICPVM**
intended usage	longer running background jobs	parallelised distributed application programs	
unit of execution	job	task	
load managing objective	load distribution	load decomposition	both
schedule policy	dynamic load sharing	cyclic allocation	dynamic load balancing
scheduling objective	resource utilization	response time of application	both
performance objective	efficiency	effectiveness	both

Table 1: Aspects of load managing in Condor, PVM and DYNAMICPVM

3 CONCEPTS OF DYNAMICPVM

In addition to PVM, DYNAMICPVM is intended to support efficient and effective parallel distributed computing, offering portability for PVM applications. The DYNAMICPVM system allows to gain control over both aspects of load managing: the decompositions and distribution of the workload. To be more specific: the decomposition of an application program into several cooperating PVM tasks by means of primitives provided by PVM and the dynamic distribution of these task over the host machines in the system by means of task migration.

In the previous section, we introduced PVM and Condor as the basic components for DYNAMICPVM. PVM provides facilities for program decomposition and Condor provides facilities for process migration: i.e. schedule and checkpoint facilities as explained in the next section. The first question which should be addressed is, should PVM be ported to Condor or should it be the other way round. The arguments for choosing PVM as basis for DYNAMICPVM are the following:

- With regard to the design concepts of Condor and PVM we found that Condor is designed to:
  a support the execution of longer running background jobs
  b schedule single process jobs independently
  c utilise idle CPU cycles of remote machines, granted by the owners of these machines
  d support execution of system calls on the initiating machine by means of the Remote System Call (RSC) mechanism (e.g. for file handling).

  and that PVM is designed to:
  a support the execution of background jobs as well as interactive jobs
  b support parallel distributed computing by means of InterProcess Communication (IPC) and coordinated scheduling of PVM tasks
  c run PVM tasks on machines for which users are entitled to consume CPU time
  d route all task-task communication and system control messages through the PVM daemons[1] which reside on each of the machines.

  For this we conclude that the concept of DYNAMICPVM resembles more with the concept of PVM than with that of Condor.

- The composition of the checkpoint mechanism and the rest of Condor allows the checkpoint software to be isolated from Condor and imbedded in DYNAMICPVM. The Condor scheduler software is an integrated part of the software but the concept can be used for the design of a scheduler for DYNAMICPVM. These two facilities can be incorporated into the PVM software without changing the application programmers interface within DYNAMICPVM.

- PVM is currently accepted as a *de facto* standard for distributed parallel computing.

DYNAMICPVM can be characterised as an enhancement to PVM, facilitating the migration of PVM tasks. The components of DYNAMICPVM are PVM 3.1, the Condor checkpoint mechanism and the Condor scheduling concept both from version 4.1b. An overview of the PVM and Condor documentation and related articles can be found in Appendix A.

[1] PVM version 3.1 from patches 5, supports direct task-task communication by means of TCP/IP sockets.

4 DESIGN ISSUES AND CHOICES

To create a facility for the migration of PVM tasks requires at least a scheduler and a checkpoint mechanism. The scheduler should be able to decide, based on the actual system state, which PVM task has to be moved to which host, when and under what name in case multiple instances of the same executable are running concurrently. The checkpoint mechanism is part of the migration protocol to disconnect the PVM task from its pvmd, make the actual checkpoint, move the checkpoint of the PVM task to its new host, restart it and reconnect it to its new pvmd. In the front of migrated tasks, communication transparency should be guaranteed. All these facilities have to be integrated with the PVM software to form DYNAMICPVM.

In this section the description of the required components, their function and interaction is given. In section 6 the implementation details and use of the components are given.

4.1 Scope and general design issues

The objective of this initial work on DYNAMICPVM is to show the feasibility of the concept. This phase mainly deals with the design and implementation of the PVM task migration facility because it is a vital building block for scheduling. A clear and compact interface between the scheduler and the pvmd daemon processes allows the simple scheduler, which comes with this distribution, to be replaced by a scheduler for dynamic load balancing.

Design issues for DYNAMICPVM:

a portability for PVM applications
b minimal modifications to the PVM and Condor software
c where possible existing PVM and Condor facilities are used
d same level of robustness as PVM and Condor

4.2 The scheduler interface

The scheduler for DYNAMICPVM should be able to activate the migration of a PVM task. For this, DYNAMICPVM has to provide an interface routine. To include the interface routine in the PVM library allows the scheduler to be implemented as a normal PVM task which benefits flexibility.

For the design of the scheduler for DYNAMICPVM the Condor scheduler may serve as a starting point. How scheduling in Condor is established, is described in [Bricker91]. In addition to the papers mentioned in section 2.3, the following references may be of help for the design of a scheduler for distributed systems: [Clark92] and [Choudhary93].

Next, a possible scheme for the DYNAMICPVM scheduler is given. In analogy to the Condor scheduler two parts can be distinguished: a **central** part which collects information sent by the **local** parts which reside on each of the host machines. The central scheduler resides on one of the host machines. The local schedulers periodically gather the hosts overall performance parameters as well as the performance parameters of the individual PVM tasks running on that host. Based on the actual performance parameters of all hosts and all 'migratable' PVM tasks in the system, the central scheduler will select candidates for migration and decide on the destination hosts, if any. In this ranking process the task/processor workload ratio in relation with the processor workload

should be taken into account. A task should be ignored as a possible candidate for migration in case it generates all the workload. Because the local and central parts of the proposed scheduler can be implemented as normal tasks in PVM, routines provided by PVM like: `pvm_config()` and `pvm_tasks()` as well as standard communication primitives can be used.

The current implemented scheduler is capable to activate the migration of selected PVM tasks (one at a time) to a selected host. Initial placement of tasks will still be carried out by the pvmd at which the tasks are spawned. The scheduler is implemented as a normal PVM task. This design allows for an easy replacement of the scheduler by another - more sophisticated - one.

The migration of a task will be activated by calling routine: `pvm_move(tid, host)`. How a PVM task will be migrated is described in section 4.6. Tasks in PVM are uniquely identified by their task identifier (tid). However, various tasks may be running instances of one and the same executable, identified by the name of the executable. The consequences of this in relation with task migration is described in section 4.4. The routing of packets to and from migrated tasks is detailed in section 4.5. The next section deals with checkpointing.

4.3 The checkpoint facility

The checkpoint facility used in Condor is developed to support the migration of processes under UNIX. Although it is part of the Condor package it can be used in 'standalone' mode for which a special library: libckpt.a is provided. The implementation aspects of the checkpoint facility is extensively described in [Litzkow92a]. How to use it in 'standalone' mode is described in [Litzkow92b].

As with Condor jobs, PVM tasks should, in addition to the PVM library, also be linked with the Condor checkpoint library. As a consequence, limitations imposed by Condor on jobs to insure successful migration will also hold for PVM tasks and hence for the PVM library. This leads to the following conflicting situation. The PVM library routines for task-pvmd communication make use of IPC calls (e.g. `socket()`, `send()`, `recv()` etc.). However, IPC calls are not supported in the Condor checkpoint library: i.e. sockets can not be restored. To relax this problem a mechanism is needed to coordinate communication and checkpointing in case the signal for checkpointing arrives while the PVM task is communicating with its pvmd. For this, the communication routines in the PVM library are marked as critical sections. When a checkpoint signal arrives during a critical section, checkpointing is deferred till the end of that section. At restart time, the connection between the PVM task and its (new) host pvmd has to be build up again.

As stated above, PVM tasks should be linked with the Condor checkpoint library (libckpt.a) in which the Condor remote execution (RSC) mechanism has been disabled. During the execution of a job, Condor records all file handling (`open()`, `read()`, `write()`, etc.) to be able to restore the file descriptors and file pointers upon the restart of a migrated Condor job. However, this mechanism disturbs the PVM task startup procedure. To make libckpt.a suitable for checkpointing PVM task, this feature has to be disabled to.

The interaction between the DYNAMICPVM checkpoint library and the PVM library is limited to the checkpointing/communication coordination mechanism.

The design and implementation of process migration is studied by many researchers like: [Artsy86] and [Artsy88] for the Charlotte system, [Hać86] where file replication and migration is also regarded, [Lu89] and [Eskicioğlu90] presenting more comprehensive work and finally [Eager88]

where the performance benefits of process migration are doubted. All implementations mentioned in these papers, require modifications of the UNIX kernel which limits their applicability.

4.4 Checkpoint creation and file naming

To prepare a PVM task for checkpointing, an initial checkpoint of the process has to be created. Consecutive checkpoints will be created out of the preceding checkpoint and the core dump file as a result of checkpointing a task. These facilities have to be integrated into DYNAMICPVM.

In PVM multiple instances of the same executable can be spawned, resulting in as many PVM tasks, each with an unique task identifier (tid). For the spawning of multiple instances, one initial checkpoint will suffice. When a task (say *tc*) is checkpointed for the first time, the initial checkpoint must be used to create a new checkpoint (*tc'*). This new checkpoint (*tc'*) should be renamed for two reasons: 1) it has to serve as basis for the next checkpoint (*tc''*), 2) for the spawning of a new instance of the executable, the initial checkpoint should remain available. However, the Condor checkpoint mechanism expects consecutive checkpoints to have the same name. Hence, each instance requires its own initial checkpoint.

To support the execution and checkpointing of multiple PVM task originating from the same executable, a task must be spawned with its name extended with its tid (e.g. slave1.40003). Because tids are unique within a PVM system, there will be no undesirable interferences. A drawback of this solution is the disk space required to contain the various initial executables.

DYNAMICPVM should be notified whether checkpointing of a task is allowed by adding the name of the executable in file 'pvm.ckptable'. To restart a task previously checkpointed, the routine: `pvm_spawn()` should be used.

4.5 Packet routing

Upon enrolment, a task receives an unique task identifier (tid) from the PVM system which serves as the task's address and hence may be distributed to other PVM tasks for communication purposes. For this reason tids must remain unchanged during the lifetime of a PVM task, thus also when the task is migrated. To achieve this, additional routing functionality have to be added to the pvmd software. The components for communication in PVM, their structure and routing are discussed in the next section. In section 4.5.2 routing in DYNAMICPVM is discussed. Detailed information of communication in PVM can be found in [Sunderam90] and [Geist93].

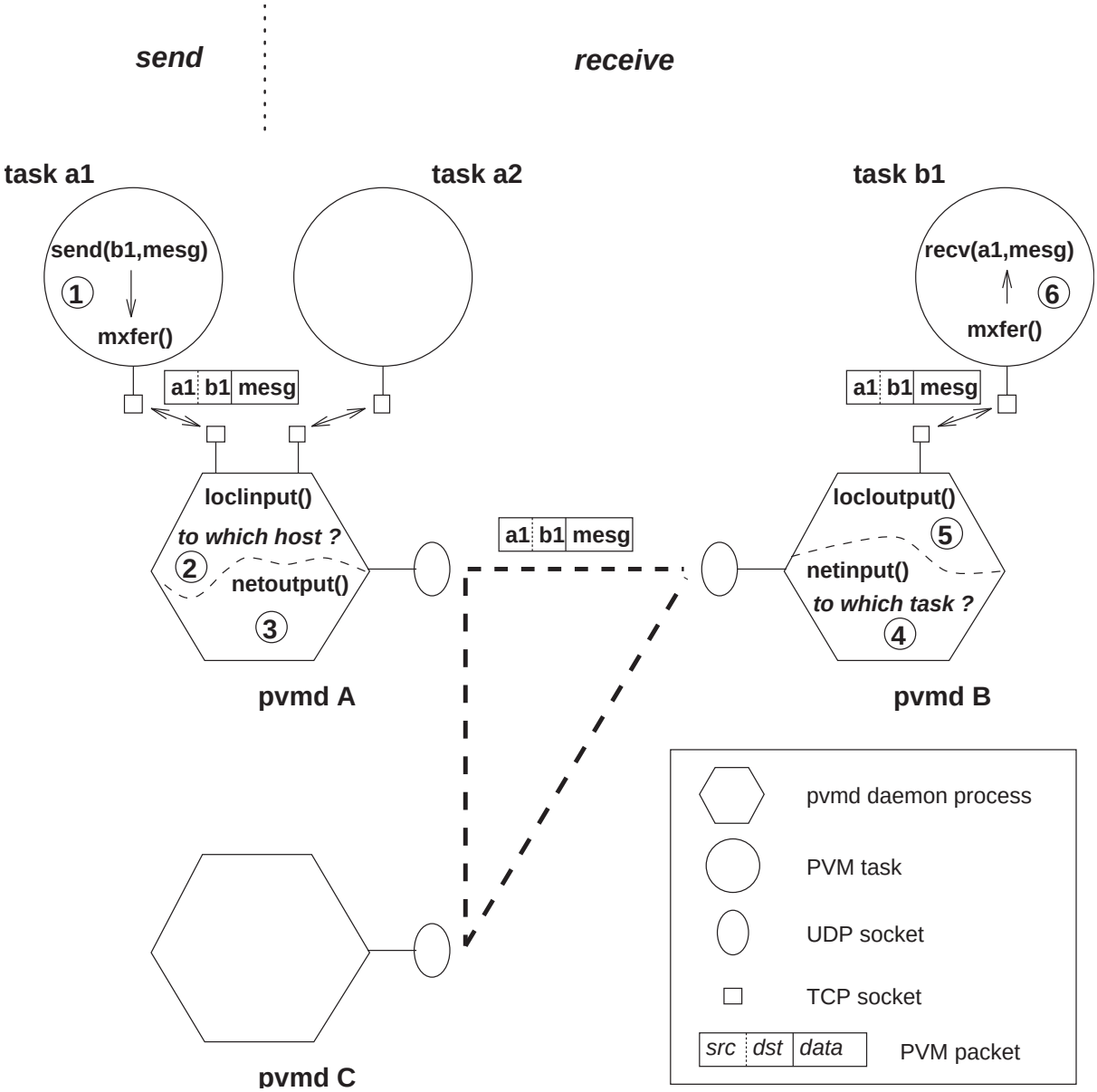

Figure 1 *Inter-task communication in PVM*

4.5.1 Communication in PVM

Routing of inter-task communication in PVM is handled by the pvmd daemon processes. On each of the machines (referred to as hosts) in the PVM system runs a pvmd. Pvmds are interconnected using UDP sockets and share status: i.e. each pvmd maintains a host list which contains the host identifier, UDP socket address, architecture, etc., of peer pvmds. Upon enrolment of a PVM task, a TCP connection between the task (via the PVM library routines) and its host pvmd will be established. Next, the host pvmd will assign a task identifier (tid) to the task. The tid is designed to contain the host identifier of the machine at which the task is enrolled and a task sequence number. A stated above, the task identifier (tid) of a PVM task serves as the task's address. The task's

```
main(){
      :
      'init'
      work();
      :
}

work() {
      :
      for(; ; ) {
            netoutput();
            nready = select(nr_fds, &read_fds, &write_fds, time_out);
            if (nready > 0) {
                  if (FD_ISSET(netsock, &read_fds)) {
                        netinput();
                  }
                  if (FD_ISSET(loclsock, &read_fds)) {
                        loclconn();
                  }
            }
            if (loclsock >= 0) {
                  for(task_pointer = locltasks->t_link; ..; ..) {
                        loclinput();
                        locloutput();
                  }
            }
      }
}
```

Figure 2 *pvmd main loop*

status: i.e. tid, TCP socket address, task name, etc. will be added to the list of task descriptors, maintained at the pvmds for local running tasks. The host list and task list are essential for communication in PVM. Besides addressing information they also buffer the packets floating through the PVM system.

Figure 1 shows the main components involved in communication in PVM. Figure 2 shows the pvmd main loop. The highlighted routines together with routine `mxfer()` form the backbone for communication in PVM. Routine: **`mxfer()`** is the PVM library interface routine which handles all communication with the host pvmd. The pvmd routines: **`loclinput()`** and **`locloutput()`** are the `mxfer()` counterparts for data receiving and sending respectively. Via routine: **`loclconn()`** the task-pvmd TCP socket connection is established. Messages for remote running tasks are routed via: **`netoutput()`** and **`netinput()`**. Messages, as referred to in the PVM library environment, are turned into PVM packets before transmission.

In the pvmd main loop (figure 2) first the packets for remote tasks are processed by `netoutput()`. For this, `netoutput()` will loop over the entries in the host list and transmit data waiting for sending to the host indicated by the entry. The **`select()`** routine (on the next line) marks the sockets for which data is ready for sending to a local task or receiving from a local task or a remote pvmd. Next, incoming packets from remote pvmds will be handled by `netinput()`. Finally, incoming and outgoing packets for the tasks in the task list are handled by `loclinput()` and `locloutput()`, respectively.

We return to figure 1 to describe routing of PVM packets for inter-task communication. Suppose task *ta1* wants to send task *tb1* a message (*ta1* and *tb1* are task identifiers, i.e. addresses in PVM). The **`pvm_send()`** routine (send(b1,mesg) in the figure) hands over the message to **`mxfer()`** which first turns it into a PVM packet and then sends it to the local pvmd (1). **`loclinput()`** extracts host id(entifier): *dB* from tid: *tb1* and places the packet in the send queue of host *dB* (2). The next time **`netoutput()`** is called for host *dB*, it takes the packet from the host's send queue and transmits it to this host (3). Upon receiving the packet, **`netinput()`** at host *dB* has to find out which the sending pvmd was, in this case pvmd *dA*, in order to send it an acknowledgement (via `netoutput()`). For this, the host identifier of the sending task *ta1* is used. Next, **`netinput()`** places the packet in the send queue of the task identified by its tid: *tb1* (4). The next time **`locloutput()`** is activated, it takes the packet from the queue and transmits it to task *tb1* (5). Finally, **`pvm_recv()`** (recv(a1,mesg) in the figure) at task *tb1* receives the message via **`mxfer()`** (6).

In case task *ta1* wants to send task *ta2* a message, **`loclinput()`** at pvmd *dA* places the packet in the send queue of the task *ta2*.

The pvmd processes use a positive acknowledgement scheme, according to which, in this case pvmd *dB* has to send pvmd *dA* an acknowledgement. Upon receiving an acknowledgement the transmitted packet will be released by `netinput()`. Unacknowledged transmissions are retried a number of times, after which the recipient 4pvmd is presumed to be inoperative.

4.5.2 Routing of inter-task communication in DYNAMICPVM

With regard to communication in the front of migrated PVM tasks we can identify two situations: 1) a migrated tasks wants to **send** a message, 2) it wants to **receive** one. These situations are depicted in figures 3 and 4, respectively. These figures may help to determine the requirements for routing in DYNAMICPVM.

Figure 3 illustrates the situation where task *tc1* is migrated from pvmd *dC* to pvmd *dA* and wants to send a message to task *tb1*. The packet will be transferred via **`mxfer()`** to task's *tc1* current pvmd: *dA*. **`loclinput()`** at *dA* will determine to which host the packet has to be send. For this, the host identifier of task *tb1* can be used. Via **`netoutput()`** the packet will be send to pvmd *dB*, the host pvmd of task *tb1*. **`netinput()`** at pvmd *dB* will forward the packet to **`locloutput()`** which in turn will transmit the packet to task *tb1*. In this situation, **`netinput()`** at the receiving pvmd must be aware that pvmd *dA* is the source host of the packet and not pvmd *dC*, while task *tc1* was the requesting task. Hence the host identifier in the tid of the sending task can not be used in DYNAMICPVM to determine which pvmd is waiting for an acknowledgement for the packet. Because we do not want to alter the PVM packet format we decided to use the socket address and port number of the sending pvmd for the identification. This information is used in PVM to double check the packets source host.

In case task *ta2* is the destination of the packet from task *tc1*, **`loclinput()`** has to handle the routing of the packet properly.

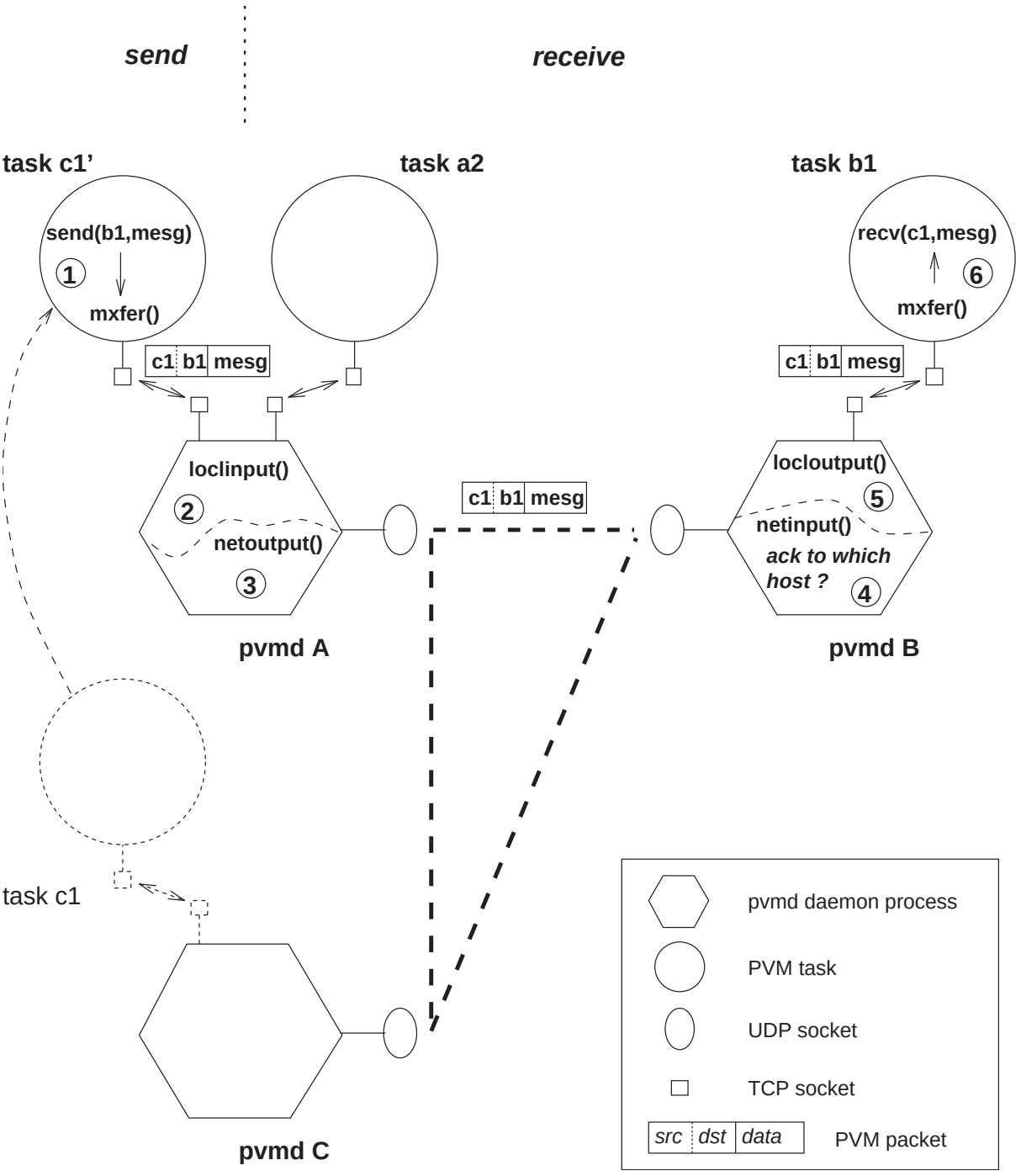

Figure 3 *Routing in DYNAMICPVM: sending PVM task migrated*

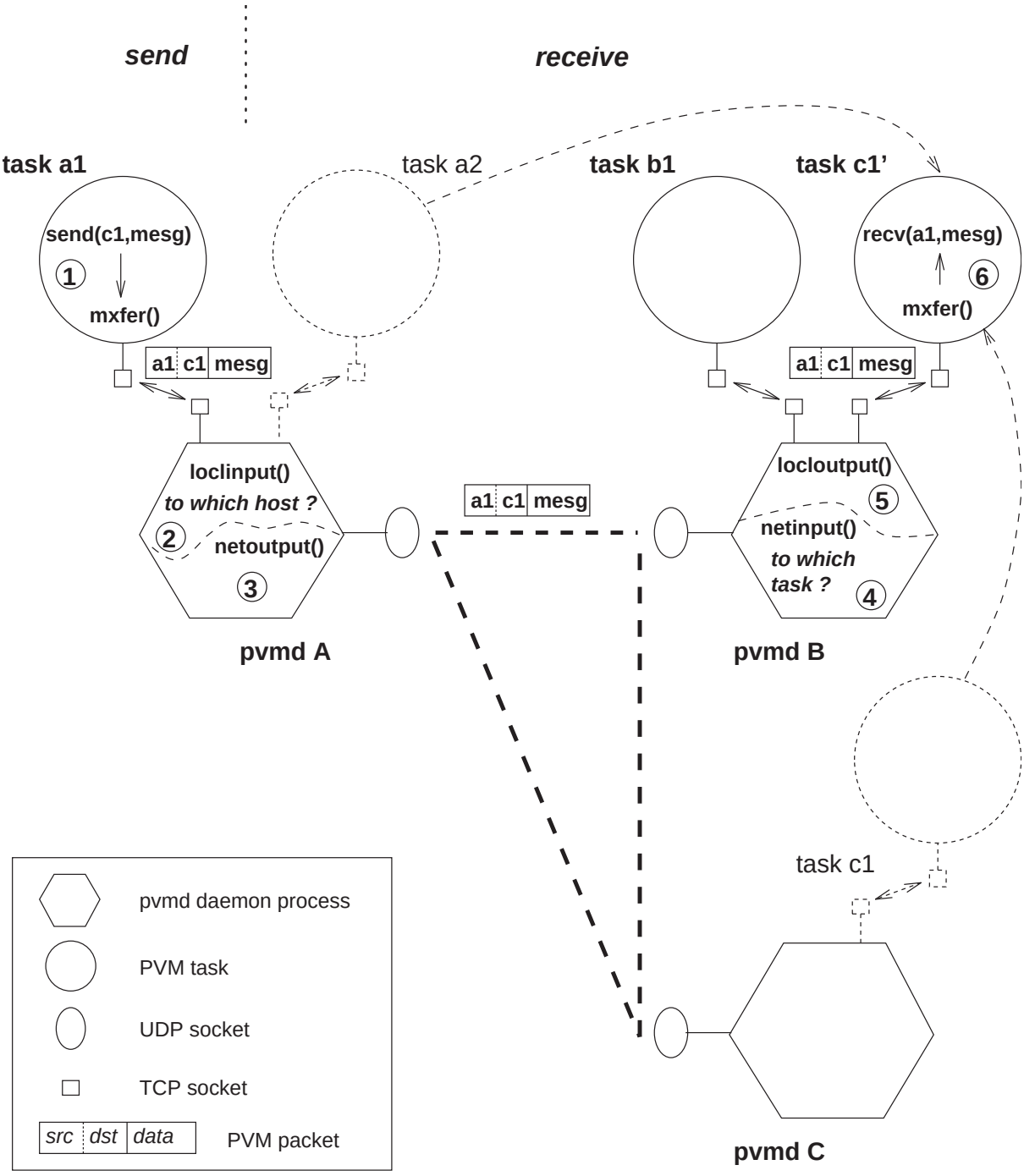

Figure 4 *Routing in DYNAMICPVM: recipient PVM task migrated*

In figure 4 the situation is illustrated where the recipient task, say task *tc1* is migrated and task *ta1* wants to send it a message. Upon receiving the packet from **`mxfer()`**, **`loclinput()`** at pvmd *dA* has to decide to which host the packet has to be transmitted. This decision requires additional routing information. **`netinput()`**, on the other site, has to decide to which task the packet has to be forwarded. Since task *tc1* is local for pvmd *dB*, no additional routing information is required.

In case task *ta2* is migrated to pvmd *dB* and task *ta1* wants to send it a message, **`loclinput()`** at pvmd *dA* must be aware that task *ta2* is not local any more. For the transmission of the data, additional routing information is required.

In case task *tb1* wants to send task *tc1* a message, **`loclinput()`** at pvmd *dB* must be aware that task *tc1* is local. For this, the task list can be used.

4.5.3 A scheme for routing in DYNAMICPVM

At the end of the previous section we stated that additional routing information is required for sending packets to migrated tasks. This information has to be provided by a routing facility in such a way that routing information is available or can be made available in the above mentioned situations.

The conditions formulated for the routing facility are:

- no changes to the tid format and contents
- no changes to message and packet format and contents
- efficient routing
- the routing scheme should permit scalability.

To meet these conditions, we choose to extend the pvmd processes with primitives for routing. These primitives operate on a route table (in analogy with primitives for task management) and are called by the pvmd routines involved in communication.
A **route table** is a double linked list, composed from structures of the form:

```
struct route {
        struct route *r_link;
        struct route *r_rlink;
        int r_tid;
        int r_route;
};
```

For maintaining the route tables, a **lazy route table updating scheme** is chosen. In this respect **an address contained in a tid or in the route table** can be viewed as a **hint**. The notion of hints can be found in [Mullender89]:

> *A hint is the saved result of some operation that is stored away so that carrying out the operation again can be avoided by using the hint. There are two additional important things to note about hints:*
>
> *1. A hint may be wrong (obsolete), and*
>
> *2. When a wrong hint is used, it will be found out in time (and the hint can be corrected).*

An implementation of the idea of hints can be found in [Kaashoek92] where the FLIP (Fast Local

Internet Protocol) is introduced. The FLIP protocol makes it possible for routing tables to automatically adapt to changes in the network topology.

In DYNAMICPVM hints are applied as follows:

1) the master pvmd[2] contains and controls the **master route table** which mirrors the actual location of **migrated** tasks in the system
2) pvmds always accept packets for local running tasks
3) hints are used for sending packets.

Route tables will be updated at all pvmds concerned with the migration of a PVM task (including the master pvmd) and consulted upon **inter-task communication**. When in inter-task communication a wrong host pvmd is addressed (e.g. by a pvmd not involved with the migration of the addressed task), the acknowledgement with a the 'TASK-UNKNOWN' flag turned on will be returned. Upon receiving this acknowledgment, the first pvmd will queue the rejected packet and request the master pvmd to send a correct route. When the requested route information is received the initiating pvmd can update its local route table and resend the postponed packet via the new route.

In this scheme, rejection of a packet is not an error but a routing failure. A drawback of this scheme is that it heavily depends on a (single) master route table. Every request for routing information has to be handled by the master pvmd. However, when the master route table appeared to be corrupted it can be corrected by consulting the other pvmds in the system. To overcome the problem of the single server, several pvmds in the system can act as a routing server for a subset of hosts.

Pvmds can perform certain actions with respect to local running tasks on request of **task information and control commands** (e.g. `pvm_pstat(tid)`, to check the status of a task). The host pvmd of the PVM task which called the PVM command and the pvmd where the action has to be taken, possibly are different ones. In these commands, not the target task itself is addressed but its host pvmd. Unfortunately, it is not possible to make use of the above mentioned routing scheme. For this class of PVM commands, first the master route table has to be consulted after which the request can be send to the task's host pvmd.

4.6 PVM task migration protocol

The main objective of the PVM task migration protocol is to guarantee migration transparency: i.e. to allow the movement of objects (PVM tasks) within the system without affecting the operation of users or application programs [Coulouris88]. With respect to a PVM task selected for migration this implicates transparent suspension and resumption of execution. With respect to the total of cooperating PVM tasks in an application, communication may be delayed but not fail due to the migration of one of the tasks. Research on this topic is reported in: [Manzo91] and [Liu92].

---

[2] The master pvmd is essential for the operation of the PVM system.

In a protocol for task migration we can distinguish four phases (task *tm* is the PVM task selected for migration):

1) **disconnect** task *tm* from its local pvmd. Communication between this task and its host pvmd has to be **flushed**.
2) **checkpoint** task *tm* and **create** a new executable: task *tm'*
3) **move** task *tm'* to its new host
4) **restart** and **reconnect** task *tm'* to its new local pvmd

There will always be some interaction required between pvmds during the disconnect phase. At least the pvmds involved with the migration have to coordinate their actions and the routing information has to be send to the relevant components of the system. Communication between the task and its local pvmd can only be interrupted at a point from where it can be recovered. Of course, no incoming or outgoing packet may get lost. The arrival of the signal at the task to checkpoint itself, initiates the disconnect phase at the task's side. The task first has to check whether it is save for checkpointing. E.g. it first has to complete all communication with its (local) pvmd. This all makes the disconnect phase complicated and vulnerable. Furthermore, incoming packets for the task during migration have to be handled correctly.

For the development of the protocol, three schemes were considered:

A) Disconnect (phase 1) task *tm* by **broadcasting** a message to notify all pvmds in the system to suspend the sending of packets to task *tm* and **wait for acknowledgements** of all pvmds. Perform phases 2, 3, and 4 and again broadcast a message with the task's routing information (no acknowledgements now required).
B) Disconnect (phase 1) task *tm* by **broadcasting** a message to notify all pvmds in the system to suspend the sending of packets to task *tm* and **reject packets** for the task after a predefined **delay**. Perform phases 2, 3, and 4 and again broadcast a message with the task's routing information.
   As an alternative, instead of rejecting packets, let the old host pvmd **forward** the packets to the new host till the second broadcast message.
C) Create an (extra) **context** for task *tm'* at the new host pvmd to receive rerouted packets for the task. **Send** the new host pvmd, the master pvmd and the current host pvmd (in that sequence) the task's routing information as start of the disconnect phase (phase 1). Perform phases 2, 3 and 4.

All schemes require a facility to queue packets destined for task *tm* in the period between the disconnect and the reconnect of the task. When task *tm* is reconnected, the postponed packets have to be send (again) to task *tm'* at its new location.

With scheme A), to broadcast a message to all pvmds and especially to collect the corresponding acknowledgments may require a significant amount of time while its unknown whether all pvmds need this information. This last remark holds for scheme B) as well where also twice a message has to be broadcasted but where the time required for the collection of acknowledgments is replaced by a fixed period after which incoming packets for task *tm* are no longer accepted. A drawback of this scheme is that the delay will be system specific and hence makes control and debugging hard. In contrast with the previous schemes, scheme C) requires a lazy routing facility to allow pvmds, not involved with the migration, to deliver packets from their local tasks to the migrated task. The advantages of this scheme over the first two are: 1) migration and routing information will only be send to relevant components and 2) lack of 'dangling' packets because of the (preliminary) receive buffer for the task at the new host pvmd. This guaranties reliable handling of packets during (and after) the migration of a PVM task.

An important property for a migration protocol is that is should be **fail safe**. At every point in the sequence of actions a lot of things can go wrong. A pvmd can die or a host crashes, the network can (partially) go down. With regard to the task, it might be unable to make a checkpoint (no core dump) or to create the new executable. To make the system more robust, these conditions have to be detected and where possible corrective actions have to be taken.

5 LIMITATIONS

The limitations with regard to PVM tasks only holds for ’migratable’ PVM tasks in DYNAMICPVM. The checkpoint and routing facility will not disturb the operation of normal PVM tasks.

The constraints with regard to the checkpoint facility are:

- Only single process PVM task are supported, i.e. the fork(2), exec(2), and similar calls are not supported
- Signals and signal handlers are not supported, i.e. the signal(3), sigvec(2), and kill(2)
- Interprocess communication (IPC) calls are not supported, i.e. the socket(2), send(2), recv(2), etc.

The limitations with regard to the schedule mechanism are:

- Only one task migration at a time can be performed
- Each condor job has an associated "checkpoint file" which is approximately the size of the address space of the process. The user should check whether sufficient disk space is available to store the checkpoint file. (Condor configuration files are not supported.)

The limitations with regard to DYNAMICPVM are:

- Currently, the use of DYNAMICPVM is restricted to a NFS cluster of IBM RS/6000, AIX32 workstations
- PVM tasks can only migrate to a host of the same platform
- File I/O is not supported
- Multicast messages[3] (for all PVM tasks) and dynamic groups are not supported.

[3] A multicast can be replaced by a repeated send, however this is less efficient.

6 IMPLEMENTATION DETAILS

This section describes the steps taken to put PVM, the Condor checkpoint facility and the scheduler into DYNAMICPVM. Both systems are programmed in C: modifications and additions required by DYNAMICPVM are performed following the programming styles of PVM and Condor. These modifications and additions are marked with the 'DY_PVM' preprocessor directive. This requires the '-DDY_PVM' option for the compilation and linking of DYNAMICPVM.

The distributions of both PVM and Condor are suited to be installed on a large number of computer platforms. However, the target platform for the development and usage of DYNAMICPVM is an IBM RISC System/6000 running under the AIX/32 operating system. With regard to the checkpoint facility, porting DYNAMICPVM to an other platform is not trivial. This because of architecture specifics. However, for doing so the information given in sections 4.3 and 6.2.1 and appendix C may be helpful.

In section 6.3, the implementation details of the lazy routing scheme and in section 6.4 the PVM task migration protocol are given. Because these items are considered the most essential parts of the system, their description is more detailed. The source code of the modifications and the additions to PVM have been made available by means of local anonymous ftp.

6.1 Setting up a development environment

For the development and testing of DYNAMICPVM, a directory structure was created to contain the relevant sources and utilities of both PVM and Condor (preserving their directory structure) and the DYNAMICPVM specifics. This allows to work with the original distribution of PVM and Condor and DYNAMICPVM under the same user account but prevent from conflicts. In appendix B.1 the directory structure is given.

To compile and link Condor jobs with the right options on and with the right libraries etc., the Condor distribution provides stanzas (in file: ~/CONDOR/config/xlc.cfg.AIX32). Stanzas for compilation and linking of the pvmd, the PVM library, the checkpoint library, 'migratable' PVM tasks etc. can be found in appendix B.2.

Some helpful aliases and facilities can be found in appendix B.3.

6.2 Creation of the DYNAMICPVM checkpoint facility

6.2.1 The DYNAMICPVM checkpoint library

The DYNAMICPVM checkpoint library is based on the Condor library: libckpt.a. This checkpoint library contains: 1) Condor stubs to handle system calls (required for the RSC option), 2) routines for the checkpoint mechanism and 3) utilities routines. Despite the RSC option is turned off, system calls are still handled by the Condor stubs in the libckpt.a library. Because the Condor stubs for file handling disturb the pvmd-task authentication procedure and because the Condor stubs form a substantial part of the library, the stubs are removed from the DYNAMICPVM checkpoint library.

For coordination between the DYNAMICPVM checkpoint library and the PVM library, critical sections in the latter are surrounded with the 'SyscallInProgress' flag (already defined by Condor) which is turned on at the beginning of the section and turned off just before the section is left. When turning the flag off, it should be checked whether checkpointing is wanted: e.g. a checkpoint signal arrived while the process was in the critical section. If so, the routine for checkpointing will be called. See also section 6.4.2.

Appendix C.1 gives a listing of the for DYNAMICPVM modified routine: `MAIN()` from source file: ckpt_main.c. The routines from the other source files: ckpt.c and restart.c are treated likewise. In Appendix C.2 the file: Imakefile.ed is listed. From this file a regular Imakefile for the imake utility used for Condor, can be created by removing all remarked ('REM') lines.

6.2.2 Checkpointing support routines

To create initial and consecutive checkpoint files of PVM tasks, the Condor routines: **`_mkckpt()`** and **`_updateckpt()`** respectively, can be used unmodified. These routines are contained in a separate program: pvmckpt (like the Condor: mkckpt program), which can be called from the PVM software with system call: `system()`. This construction promotes portability. See appendix C.3 for program pvmckpt.

The creation of the initial checkpoint file of a task requires the task identifier (tid). With the tid appended to the task's name, an unique name for the initial checkpoint file is created, to prevents from conflicts. Upon spawning a PVM task, a child process will be forked off from the task's host pvmd and executed by pvmd routine: **`forkexec()`**. In this routine the task's tid will be created. For DYNAMICPVM, `forkexec()` is modified to create the initial checkpoint file in case the task spawned appears to be 'migratable'. The task will be executed under its initial name extended with its tid. The initial names of 'checkpointable' PVM tasks have to be added to file: pvm.ckptable. With routine: **`pvmckptable()`**, this file can be examined.

In contrast with the creation of an initial checkpoint file, the creation of a consecutive one may consume significantly more time. For this reason, this process should be part of the scheduler (PVM task) and not of the pvmd. See further at section 6.4.

6.3 Lazy routing facility

Implementation of the lazy routing facility required the modification of the pvmd routines: **`netoutput()`**, **`netinput()`**, **`loclinput()`** and **`sendmessage()`** all in source file: pvmd.c. Two additional routines in the dd (daemon-to-daemon messages) subsystem of the pvmd are added: **`dm_rtget()`**, **`dm_rtgetack()`**, **`dm_rtotid()`** and **`dm_rtotidack()`** in source file ddpro.c. Furthermore, routines for route table management are provided: **`route_init()`**, **`route_new()`**, **`route_find()`**, **`route_free()`** and **`route_dump()`** contained in an additional source file route.c. The routines for task management (in task.c) served as an example.

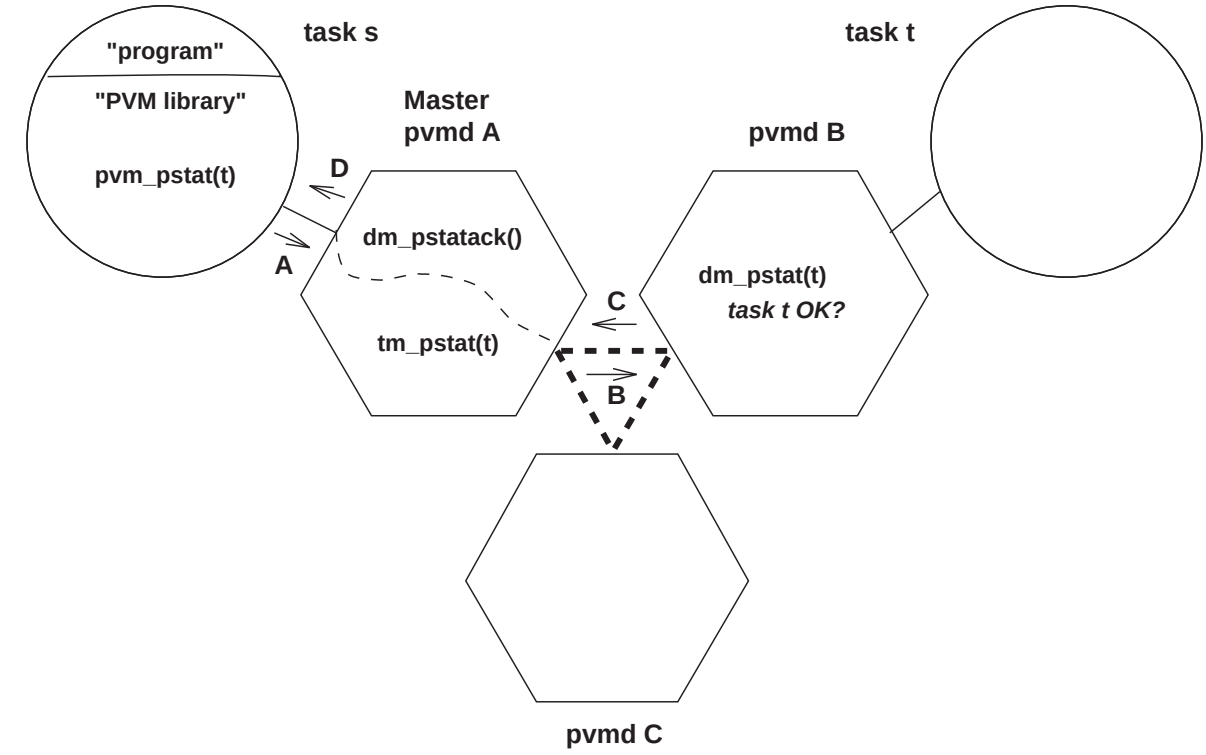

Figure 5 *Calling sequence for information & control commands in PVM*

The layout of the structure 'route' on which these routines operate is defined in header file: route.h (see also in section 4.5.3):

```
struct route {
        struct route *r_link;
        struct route *r_rlink;
        int r_tid;
        int r_route;
};
```

6.3.1 Routing of task information & control commands

The task information and control commands in the PVM library are: `pvm_kill()`, `pvm_pstat()`, `pvm_sendsig()`, `pvm_tasks()`. In all these commands, the host pvmd of the task which tid is in the argument list, is addressed, not the task.

Figure 5 gives an example of how a typical information and control call to the PVM library is handled in PVM. The routine `pvm_pstat()` is part of the PVM library and is called in user application, say task *ts* to inspect the status of task *tt*. In turn it will call routine `tm_pstat()` at pvmd *dA* (A). Routine `tm_pstat()` determines on which host task *tt* resides to forwards the request (B). On the host pvmd of task *tt*, routine `dm_pstat()` is called to inspect the task's status. The status of task *tt* is returned to `dm_pstatack()` (C) at the calling pvmd and finally forwarded to the user application: task *ts* (D).

Even in the situation of migrated tasks, these commands have to function correctly in

DYNAMICPVM. For this, the call in `tm_pstat()` to `dm_pstat()` is replaced by a call to `dm_rtotid()` to consult the master route table at the master pvmd. Upon receiving the routing information by `dm_rtotidack()` from the master pvmd, `dm_pstat()` is called at the stated pvmd.

6.3.2 Routing of inter-task communication in DYNAMICPVM

In this section is described how lazy routing is achieved in DYNAMICPVM.
Recall that routing in DYNAMICPVM is based on the concept of hints, where hints are the addresses of hosts contained in tids or in the route table. The rules for the use of hints mentioned in section 4.5.3 are:

1) the master pvmd contains and controls the **master route table** which mirrors the actual location of **migrated** tasks in the system
2) pvmds always accept packets for local running tasks
3) hints are used for sending packets

The master route table will be updated during the task migration process. To update a route table, routine: `route_find()` will be called to check whether an entry for the migrating task exists. If so, the route will be updated, otherwise a new route will be added by calling: `route_new()`. In section 6.4.1 is described at which stage of the migration process the master route table will be updated.

The PVM the pvmd routines: `netinput()` and `loclinput()` read packets from remote pvmds or local running tasks respectively, next they perform some communication related processing and finally route them to the destination task. In DYNAMICPVM they have to check, after the packet is read, whether the destination task runs local or not. How `netinput()` handles incoming packets for task which are not longer running local is described in the next section. Routine: `loclinput()` will call `route_find()` to examine the local route table. In case an entry in the local route table for the destination task is found, the route from the route table will be used, otherwise the route in the task identifier (tid) of the destination task will be used for sending.

There is one other routine in the pvmd process which has to be modified to support routing in DYNAMICPVM, routine: **`sendmessage()`**. This routine is used by pvmd routines to send a (information or control) message to a local task or a remote host. The modifications are similar to the those required for `loclinput()`.

6.3.3 Implementation of the lazy route table update scheme

Here we describe the interaction between the `netoutput()` and `netinput()` routines in case a packet arrived for a task which is not local any more, see figure 4. We assume that the task is running at another host and that the master route table is updated. Updating of the master route table is part of the PVM task migration protocol.

A packet from a task will enter the host pvmd at routine: `loclinput()`. This routine has to determine the target pvmd to which the packet has to be transmitted. First it will check whether the local route table includes an entry for the destination task (`route_find()`). If not, the host identifier included in the tid of the destination task is used. The packet will be placed by `loclinput()` in the send queue of the selected host, after which it will be transmitted by `netoutput()` to this host. At this event, `netoutput()` raises a flag to inform the local `netinput()` that it should expect an acknowledgement from this host for this packet.

Upon receiving the packet by the target's host `netinput()`, it will detect that the destination task is not a local task. At this event, it will raise a flag to inform its local `netoutput()` that a 'TASK-UNKNOWN' acknowledgement should be returned instead of a normal acknowledgement.

At the first pvmd, when `netinput()` detects the 'TASK-UNKNOWN' acknowledgement, it will first move the packet from the send queue to the wait queue, after which it will request the master pvmd to send it a route update by calling `dm_rtget()`. The master pvmd returns the requested routing information by calling `dm_rtgetack()` at the requesting pvmd. This routine will first update the local routing table and than move the packet from the wait queue of the rejecting host to the send queue of the new target host. `netoutput()` will take care of the transmission of the packet to this host.

To reduce the inter-pvmd communication required for the route update request, the following optimisation is performed. When the master pvmd rejects a packet for a task, it will return a 'TASK-UNKNOWN-UPDATE' acknowledgement to the host pvmd after which it will send the route update information to this pvmd directly.

6.4 Implementation of the PVM task migration protocol

6.4.1 Routines and phases

To implement the PVM task migration protocol routine: **`pvm_move(tid, host)`** is added to the PVM library and is intended to be used by the central part of the DYNAMICPVM scheduler (to be implemented as a PVM task). The pvmd daemon program is extended with three routines: **`tm_move(code, tid, host)`** **`dm_move(code, tid, hostpart)`** and **`dm_moveack(code)`** These four routines are implemented in analogy with existing pvm routines like `pvm_pstat()`, see figure 5 and the text in 6.3.1.

Description of routines for task migration:

`pvm_move(tid, host)`	called by the DYNAMICPVM scheduler (pvmsched) to migrate a task. It controls the migration process. With argument: *tid* a task for migration is selected and with argument: *host* the destination host.
`tm_move(code, tid, host)`	counterpart of the pvm_move routine, residing at the host pvmd of pvmsched. With argument[4]: *code* the phase of the migration protocol is selected.
`dm_move(code, tid, hostpart)`	called by `tm_move()`, residing at the target pvmd. Argument *hostpart* is the PVM address of a host.
`dm_moveack(code)`	concludes protocol phases by returning to the `pvm_move()` routine.

[4] Actually, the arguments are passed as messages.

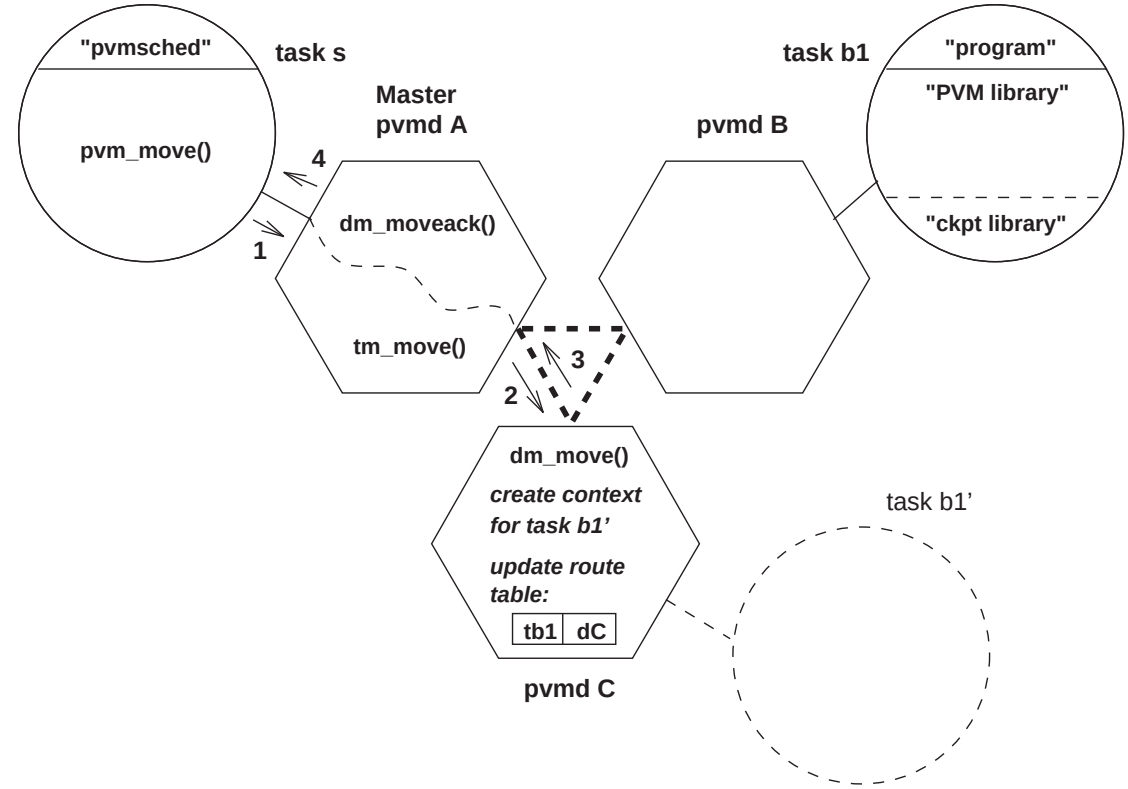

Figure 6 *DYNAMICPVM task migration protocol, phase:* ***init***

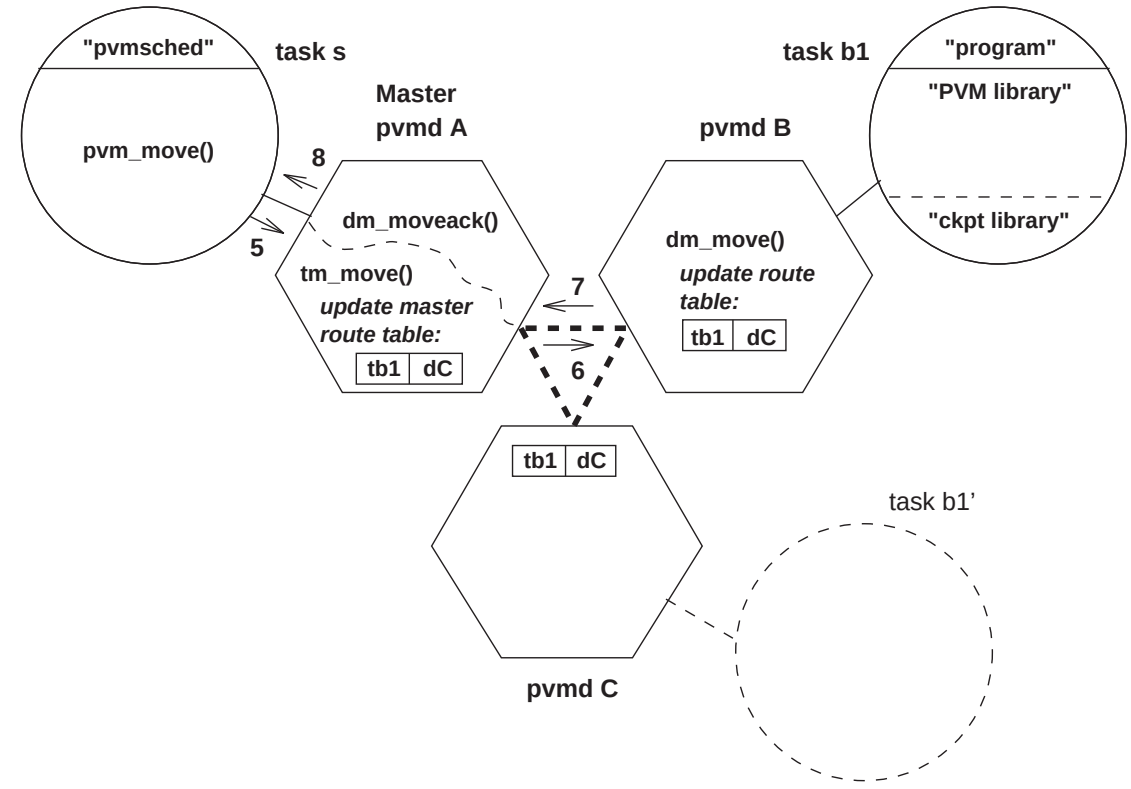

Figure 7 *DYNAMICPVM task migration protocol, phase:* ***flush***

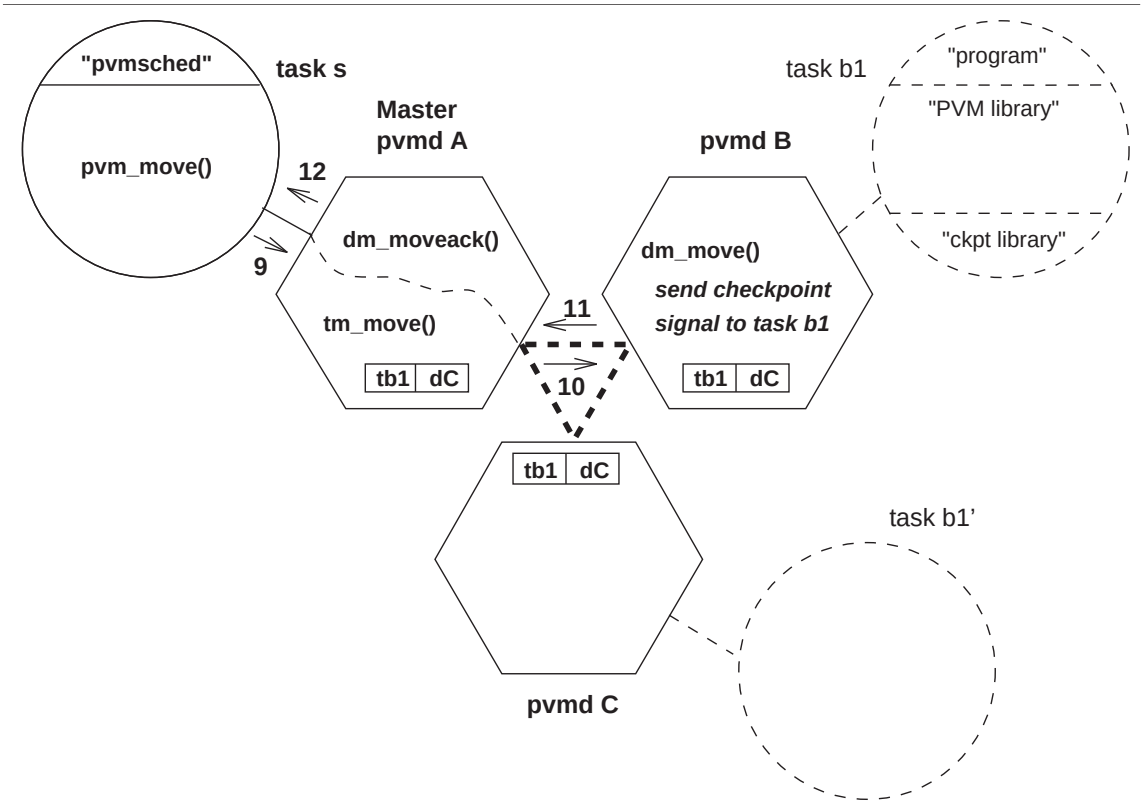

Figure 8 *DYNAMICPVM task migration protocol, phase:* ***ckpt***

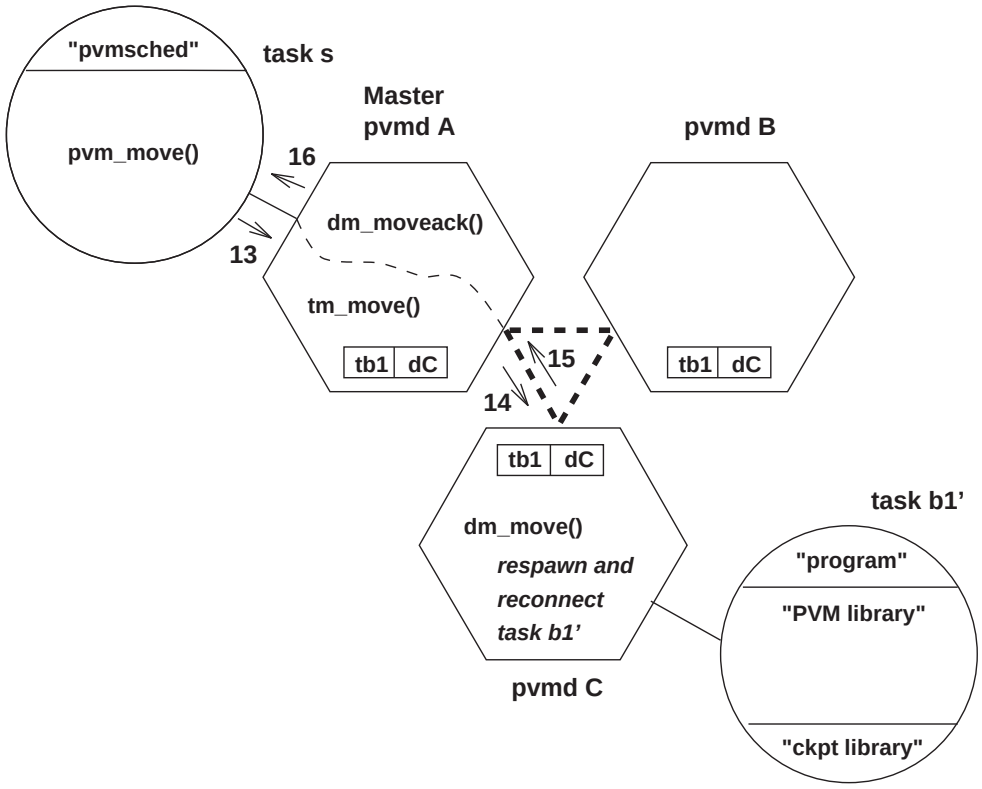

Figure 9 *DYNAMICPVM task migration protocol, phase:* ***spawn***

The migration process takes four consecutive phases to sequence the required actions. Each phase requires a calling sequence of the above mentioned routines. The migration process is illustrated in the figures 6 to 9 inclusive. This figures also illustrate that ’checkpointable’ PVM tasks are linked with both the PVM library and the checkpoint library. As an example, the migration of task *tb1* at pvmd *dB* to pvmd *dC* is requested.

Description of phases:

Init First, an extra context for task *tb1’* will be created at task’s *tb1’* new host pvmd: *dC*. Next, the local route table at pvmd *dC* will be updated. See figure 6.

A task context contains a receive buffer for incoming packets. According to the PVM documentation, the task context will accept packets even if the connection with the task is not yet established.

From this point packets for task *tb1* from other local tasks will be accepted and placed by `loclinput()` at pvmd *dC* in the receive buffer of task *tb1*.

Flush First the master route table at the master pvmd will be updated. Next, the route table at the current host pvmd will be updated. See figure 7.

Because the master route table is updated, all packets for task *tb1* will be routed to its new host pvmd. However, this will cause a problem in case task *tb1* is blocked by routine `mxfer()` while waiting for an incoming packet. Task *tb1* is in a critical section at this point (see also below at 6.4.2) and checkpointing will be deferred till the end of this critical section. Hence, task *tb1* will be blocked forever. To tackle this problem, a packet has to be send to the task in order to ’unblock’ `mxfer()`. This packet is: 1) a packet already queued for sending with an extra flag ’UNBLOCK’ raised, b) a dummy packet with an extra flag ’DUMMY-UNBLOCK’ raised. Flag ’DUMMY-UNBLOCK’ informs `mxfer()` at task *tb1* to discard the packet. Both extra flags will cause `mxfer()` to wait, just before its return, on the arrival of the checkpoint signal. At phase ’Spawn’ it is described how to recover from this.

The return of `dm_moveack()` to the `pvm_move()` routine at this point is required in case task’s *tb1* current pvmd is the master pvmd. This forces routine `locloutput()` at task’s *tb1* current pvmd to send it the ’unblock’ message. See figure 2.

Ckpt Here, the TSTP signal for task *tb1* to checkpoint itself, will be send. If the call was successful, the application name of task *tb1* will be returned via `dm_moveack()` to `pvm_move()`. If the call was unsuccessful, an empty string will be returned to indicate this event. See figure 8.

Upon receiving the TSTP signal, the signal handler of the checkpoint facility will be called. The signal handler and the PVM library have to coordinate their actions to prevent interference with packets currently being transmitted. See also below at 6.4.2.

It is possible that a packet had arrived at task’s *tb1* socket, in advance of the call to the `pvm_recv()` or `pvm_nrecv()` routine by the task. To prevent from loss of data caused by the checkpoint process when closing the socket, the data have to be

saved first. How to recover from this is described in the next phase.

When the actual checkpoint process is started, it will call PVM library routine: `pvmendtask()` to close the socket and mark the task as 'unconnected'.

Spawn In this last phase the checkpoint file of task *tb1* will be updated and spawned again if the application name, returned in the previous phase, is not empty. The task at the old host pvmd will vanish. Figure 9 illustrates this.

The updating of task *tb1* checkpoint file will be carried out in routine: `pvm_move()` of the scheduler task and not by a pvmd routine. This, because the updating process is relative CPU and I/O intensive (the process will take up to a few seconds, even for small tasks). Hence, it is not acceptable as a pvmd action.

Upon spawning task *tb1*, the program will recover from checkpointing and continue from where it was interrupted. First, the PVM routine: `pvmbeatask()`, has to be called by which task *tb1* will be reconnected with its task context at the new pvmd. See also below at 6.4.3.

In case a task was blocked while waiting for incoming packets at phase 'Flush', as last step of the recovery process the `mxfer()` routine will be called again. From here, the interrupted process can continue.

In case data was pending at the socket in phase 'Ckpt', as last step of the recovery process the data will be send to the new socket. From here, the interrupted process can continue.

6.4.2 Timing aspects of checkpointing

To prevent interference of checkpointing and transmission of packets, the signal handler first checks whether the program is in a critical section. If so, the signal handler will raise a flag to indicate that checkpointing is wanted. Upon leaving the critical section, the flag will trigger checkpointing. This mechanism is used in Condor in case a TSTP signal arrived during the execution of a (local or remote) system call. See also section 6.2.

The critical sections are the PVM library routines: `mxfer()`, `pvmendtask()` and `pvmbeatask()`.

According to the IBM RS/6000 AIX32 documentation, arrivals of a signal will not interfere with system calls from standard libc.a.

6.4.3 Restart and reconnect

In the process of spawning a task in PVM, the pvmd routine: `forkexec()` will be called. In this routine a task context will be created in regular cases. In case of spawning a checkpointed task, `forkexec()` has to use the task context already created in a previous phase.

Routine `pvmbeatask()` of the PVM user library initiates connection of a task with its pvmd. The counterparts at the pvmd daemon program are the `tm_connect()` and `tm_conn2()` routines by which the task context will be completed. In case of a checkpointed task the original tid and the tid of its original master will be restored.

6.4.4 Recovery from failures

It will be checked whether the checkpoint process was successful or failed. The existence of the executable and the core file will be checked to. However, no extended fault recovery mechanism is implemented yet.

6.5 The DYNAMICPVM scheduler

To activate the migration of a running PVM task, program: pvmsched can be used. This small program is intended to show the migration of a PVM task. It prompts for the task's identifier and the host name to which the task should be moved.

Pvmsched is implemented as a normal PVM task which has to run on the master pvmd. The program can quite easily be extended with the PVM information routines: `pvm_pstat()`, `pvm_mstat()`, `pvm_config()`, `pvm_tasks()`. These routines provide information about the PVM configuration and tasks running locally or remotely and may be used as building blocks for a real scheduler.

6.6 Starting and stopping DYNAMICPVM

Provided DYNAMICPVM is installed as described in appendix B, DYNAMICPVM can be started and stopped in the same way as PVM.

7 PRELIMINARY RESULTS AND STATUS OF DYNAMICPVM

The objective of this initial work on DYNAMICPVM is to show the feasibility of the concept. The main components for DYNAMICPVM are: 1) a checkpoint facility, 2) a facility for routing of communication and 3) a protocol for task migration.

We first focused on master/slave type applications. The masters task spawns a slave task, sends it data and waits for the results returned by the slave. Upon receiving data, the slave starts working and when finished, sends the results back to the master. This scheme shows: 1) the operation of the checkpoint facility, 2) part of the routing facility: the migrated task should be able to return the result and 3) part of the migration protocol: no data have to be transmitted to the slave during migration. After the process is started and the slave is spawned by the master, the migration of the slave can be activated by means of 'pvmsched'. This task prompts for the task identifier of the slave and the new host for the slave. We found that under these conditions, the task was successfully migrated in DYNAMICPVM.

Next, a master/slave application where multiple slaves were spawned was tested. Before returning the results to the master, they first exchange results. In addition to the previous scheme, this scheme shows bi-directional inter-task communication with a migrated task. However, no data is exchanged during the migration of one of the tasks. We found that bi-directional inter-task communication was handled properly in DYNAMICPVM.

The checkpoint mechanism and the routing facility are now in place in DYNAMICPVM and work properly. The framework for the task migration protocol was found to function correctly. The facility for routing of bi-directional inter-task communication during task migration and the facility for disconnecting a task from its pvmd during the migration of the task while data is pending, have been prepared.

8 USING DYNAMICPVM

Setting up the environment

The DYNAMICPVM development environment, given in appendix B.1, should be in place. The aliases and utilities listed in appendix B.3, are provided to support the compiling and linking of PVM applications.

Compiling and linking applications

First, one has to decide which PVM tasks should run as a normal (’not-checkpointable’) tasks and which tasks should run as ’checkpointable’ tasks.

Sources of ’non-checkpointable’ PVM tasks may be located in directory:

~/DynamicPVM/appl-devl/pvm-normal

The tasks can be compiled and linked using the standard Makefile and the PVM ’aimk’ facility.

Sources of ’checkpointable’ PVM tasks may be located in directory:

~/DynamicPVM/appl-devl/pvm-ckptable

These tasks have to be linked with the DYNAMICPVM checkpoint library. For this a stanza is provided (see appendix B.2). With alias ’Dpvmimk [executable]’, a Makefile for the executable can be created. For compilation and linking use ’aimk’.

The executables are expected by DYNAMICPVM to be in directory:

~/DynamicPVM/pvm3/bin/RS6K

This path may be defined in the environment variable: ’path’.

Executing an application

To notify DYNAMICPVM that a task is ’checkpointable’, the name of the executable has to be entered into file ’pvm.ckptable’. This file resides in the same directory as the executable.

Using the DYNAMICPVM scheduling facility

The DYNAMICPVM scheduler: pvmsched, should be compiled and linked as a normal PVM task. To activate the migration of a task, pvmsched has to be started under the master pvmd. It will prompts for the task identifier and the name of the new host for the task. The migration of task under the conditions given in the previous section, should function correctly.

9 DIRECTIONS FOR FUTURE WORK

In this section, directions for future work are given. We will discuss how some of the limitations listed in section 5 can be repealed and we will discuss possibilities for future work with regard to scheduling in DYNAMICPVM. We will start with the direction for further development of the task migration protocol in DYNAMICPVM.

Task migration protocol

As stated in section 7, the part of the task migration protocol which handles pending task-pvmd communication needs further development. The framework for this development may be found in section 6.4. However, some facilities for this have been prepared - like the mechanism required for the communication/checkpointing coordination.

Limitations

To port DYNAMICPVM to another platform, no modifications of the pvmd daemon program and the PVM library are required. Recompilation should suffice. However, due of system specifics, the DYNAMICPVM checkpoint facility has to be produced for each platform individually. The checkpointing support routines for the creation of initial and consecutive checkpoints, are already provided by Condor for various target platforms. The program: pvmckpt (mkckpt in Condor), which is the interface between DYNAMICPVM and its checkpoint facility, has to be recompiled to.

To run tasks under DYNAMICPVM in a system without the support of a shared file system, the checkpoints have to be moved explicitly between the task's old and new host. This requires a quite trivial modification of the `pvm_move()`routine of the PVM library.

To support file I/O in 'checkpointable' PVM tasks, the DYNAMICPVM checkpoint facility has to be modified. File I/O is supported by the Condor checkpoint facility, but was removed from the checkpoint facility for DYNAMICPVM because it disturbs the pvmd-task authentication procedure. Enabling file I/O in 'checkpointable' PVM tasks, requires the Condor RSC (Remote System Call) mechanism which provides the stubs for file manipulation system calls.

To supported multicast messages and dynamic groups in DYNAMICPVM, the routing aspects of both PVM facilities have to be analysed. Where needed, the routing mechanism has to be extended with the routing primitives provided by DYNAMICPVM. As for normal messages, routing for multicast messages and dynamic groups are handled by the pvmd communication routines. The modifications will be similar to the modifications needed for the routing of regular messages.

Sufficient disk space should be available to store the "checkpoint file", in order to insure proper checkpointing. In case the required disk space is not available, checkpointing should be deferred. In Condor, configuration files are applied to provide the scheduler with information about the jobs to schedule: disk and memory requirements, architecture to run on, etc. This feature might be added to DYNAMICPVM.

Scheduling

Next we will discuss future work concerning scheduling in DYNAMICPVM. As stated, this report on DYNAMICPVM mainly deals with the design and implementation of a PVM task migration facility. The next step in the DYNAMICPVM development might be the design and implementation of the scheduler. DYNAMICPVM allows and encourages exploration of different schedule strategies for parallel distributed computations.

There are a number of intertwined issues with regard to scheduling in DYNAMICPVM which probably are best solved by an integral approach. These issues are:

- initial placement
- access to a restricted set of machines by DYNAMICPVM
- global scheduling.

The problem of initial placement is not solved by DYNAMICPVM. Tasks are still allocated to machines, irrespective of the current work load. However, the scheduler interface can easily be extended with a facility which allows the scheduler to decide to which hosts new tasks have to be allocated. Next, the 'scope' of the DYNAMICPVM scheduler is restricted to the machines to which DYNAMICPVM users have access. This is because the DYNAMICPVM daemon processes and the PVM tasks run as user processes under the user's account. It does not seem a good solution to give PVM users accounts on **all** available machines, i.e. also outside their local cluster. Finally, race conditions may occur if multiple PVM users are on the same cluster. This is caused by different schedulers which are competing for the same resource. This requires a global scheduling facility for all DYNAMICPVM users.

Much of the work to tackle these above scheduling problems has been done by the Condor development team. At the "1993 PVM Users group meeting", held in Knoxville, Miron Livny and Jim Pruyne presented their paper entitled: "Scheduling PVM on workstation clusters using Condor", which concerns the use of the Condor scheduling mechanism for the allocation of PVM tasks to idle workstations. For this, Condor was extended to support the PVM calls and an interface required for scheduling was built. Bringing this concept and the concept of DYNAMICPVM together yields a system with maximum support for parallel distributed computing in a network of workstations. The scheduler interface of DYNAMICPVM allows easy integration of both products.

10 DISCUSSION AND CONCLUSIONS

Leading researchers in the field of distributed computing envisage that distributed systems will become increasingly important, first for scientific and data base applications, later also for office and financial systems. The development of the OSF/DCE (Open Software Foundation/Distributed Computing Environment) products, in which several major vendors participate, indicates that distributed systems are also of commercial interest.

At KSLA, with an installed base of tens of workstations, most of which are used to run scientific applications, distributed systems may contribute to better utilisation of the machines. Just as file servers offer transparent file access to users, distributed systems are able to offer transparent program execution. Both aspects are in the interest of users, who no longer have to worry about which machine is best suited to run their applications, which may then run faster.

To introduce distributed systems at KSLA, the cooperation of different disciplines is required. First, systems management has to install and maintain the products at the machines selected for this purpose. The Network File System (NFS) offers good facilities to support this. Furthermore, existing software has to be modified for running in a distributed system. This modification may be trivial, as relinking, or complex, as decomposing the program into cooperating subtasks. For this, software development skills from research and/or support departments are required. Due to interprocess communication, it is expected that running applications in a distributed system will have a major impact on the network utilisation. In turn, the volume of interprocess communication will largely depend on the decomposition of the program. Sometimes it might be inevitable to transmit large amounts of data between tasks. In this respect we note that KSLA is currently in the process of upgrading the network, which will facilitate distributed computing.

To support distributed systems, the resources in the system as for example, the CPU and memory usage of machines, the global and local network load, have to be monitored. Distributed systems will turn the network and the machines into a network of machines as more and more users will be unaware on which machine their tasks are executed. The systems and the network management job may be lightened by distributed systems because they allow more centralised management.

Several distributed systems are available, but as most of them are public domain, resulting from research development by universities. As stated, OSF/DCE - a Shell recommended standard - is a commercially available product.

Distributed systems are promising because they make it possible to combine systems with different strengths, which thus become more powerful than conventional systems. Furthermore, distributed systems may help solve the problem of system management in an environment with a large number of processors and file systems. However, the understanding of and knowledge required for building distributed systems are not mature yet. Many research issues have to be addressed. Scheduling in distributed systems for parallel distributed computing is one of them.

We have introduced PVM and Condor as examples of systems which can be classified as distributed systems: PVM supporting parallel distributed computations and Condor scheduling batch jobs in a network of workstations. The two systems are to some extent complementary. Combining the PVM and Condor systems offers the interesting opportunity for dynamic scheduling of PVM tasks. The resulting system we name: DYNAMICPVM.

The final goal of the development of DYNAMICPVM is to provide users with an environment for efficient and effective parallel distributed computing on a network of workstations. DYNAMICPVM is intended to allow users to gain full control over PVM applications, namely the decomposition of the application and the distribution of the PVM tasks. For the decomposition of applications, the PVM programmers' interface offers powerful facilities. For the distribution of PVM tasks, the scheduling mechanism as in Condor allows efficient resource management.

The work on DYNAMICPVM reported here embodies the design and implementation of a facility for migration of PVM tasks. This facility is essential for dynamic scheduling, of which scheduling in Condor is an example. An interface routine is added to the PVM programming environment with which task migration can be activated. Inclusion of this routine in the programming environment, makes it possible to implement the scheduler for DYNAMICPVM as a normal PVM task. This approach encourages experimenting with scheduling policies in DYNAMICPVM.

Portability for PVM tasks is the general design issue for DYNAMICPVM. Cooperating PVM tasks should not be aware that one of the tasks is being moved from one machine to another. The main components for DYNAMICPVM are: 1) a checkpoint facility, 2) a facility for routing of communication and 3) a protocol for task migration. This protocol is the most compound part which had to be added to PVM. The disconnect phase of the protocol, where the task is disconnected from the system in order to checkpoint it, was found to be the most complex and vulnerable one. The PVM system has no *a priori* knowledge about the state of the task. The task might be blocked while waiting for data, it might have been receiving data in advance of the reading, it might be sending data or it might just be doing some calculations. For each of the situations, the protocol has to take the proper steps so no data will get lost while checkpointing and restarting a task. For this part of the protocol further development is needed.

The routing scheme adopted for DYNAMICPVM, based on hints, is simple and scalable. The lazy route table update strategy minimises the number of packets to be sent to exchange routing information. Tests showed that this facility functions properly.

Checkpointing of tasks on an IBM RS/6000 under AIX32 is far from trivial: however, the Condor checkpoint facility provides a solid basis for the DYNAMICPVM implementation. Checkpointing in DYNAMICPVM showed to function properly.

Because the scheduler interface of DYNAMICPVM can be called from a normal PVM task, DYNAMICPVM encourages experiments with advanced strategies for dynamic scheduling. In this respect, DynamicPVM may be of importance for the system presented by Miron Livny and Jim Pruyne at the "1993 PVM Users group meeting". On this occasion they presented their paper entitled: "Scheduling PVM on workstation clusters using Condor", which concerns the use of the Condor scheduling mechanism for the allocation of PVM tasks to idle workstations.

Bringing this scheduling concept and the task migration concept of DYNAMICPVM together yields a system with maximum support for parallel distributed computing.

11 BIBLIOGRAPHY

[Almasi93] G.S. Almasi, D. Hale, T. McLuckie, J. Bell and A. Gordon, "Parallel distributed seismic migration", *Concurrency: practice and experience*, vol. **5**(2), pp. 105-131, April 1993.

[Artsy86] Y. Artsy, H-Y. Chang and R. Finkel, "Processes migrate in Charlotte", Computer Sciences Technical Report #655, University of Wisconsin-Madison, August 1986.

[Artsy88] Y. Artsy and R. Finkel, "Designing process migration: the Charlotte experience", Technical Report 131-88, University of Kentucky, November 1988.

[Beguelin92] A. Baguelin, J. Dongarra, A. Geist, R. Manchek and V. Sunderam, "PVM and HeNCE: traversing the parallel environment", *Gray channels*, vol **14**, no. 4, pp. 22-25, Fall 1992.

[Bricker91] A. Bricker, M.J. Litzkow and M. Livny, "Condor technical summary", Version 4.1b, University of Wisconsin-Madison, October 1991.

[Choudhary93] A.N. Choudhary, B. Narahari and R. Krishnamurti, "An efficient heuristic scheme for dynamic remapping of parallel computations", *Parallel computing*, vol. **19**, pp. 612-632, 1993.

[Clark92] H. Clark and B. McMillin, "DAWGS - A distributed computer server utilizing idle workstations", *Journal of parallel and distributed computing*, vol. **14**, pp. 175-186, 1992.

[Coulouris88] G.F. Coulouris and J. Dollimore, "*Distributed systems: concepts and design*", Addison-Wesley, 1988.

[Dongarra93] J. Dongarra, G.A. Geist, R. Manchek and V.S. Sunderam, "Integrated PVM framework supports heterogeneous network computing", *Computers in physics*, vol. **7**, no. 2, pp.166-175, March/April 1993.

[Eager84] D.L. Eager, E.D. Lazowska and J. Zahorjan, "Dynamic load sharing in homogeneous distributed systems", Technical report 84/16, University of Saskatchewan, Saskatoon, October 1984.

[Eager88] D.L. Eager, E.D. Lazowska and J. Zahorjan, "The limited performance benefits of migrating active processes for load sharing", *Proceedings of the '88 ACM SIGMETRICS Conference*, pp. 63-72, 1988.

[Eskicioğlu90] M.R. Eskicioğlu, "Process migration in distributed systems: a comparative survey", Technical Report TR 90-3, University of Alberta, Edmonton, Alberta, January 1990.

[Evers92] X. Evers, "A literature study on scheduling in distributed systems", Delft

University of Technology, October 1992.

[Evers93] X. Evers, "Condor flocking: load sharing between pools of workstations", Graduation thesis, Delft University of Technology, April 1993.

[Fox88] G. Fox, M. Johnson, et al., "*Solving problems on concurrent processors*", Volume 1, Prentice Hall, 1988.

[Geist92] G.A. Geist and V.S. Sunderam, "Networked-based concurrent computing on the PVM system", *Concurrency: practice and experience*, vol. **4**(4), pp. 293-311, June 1992.

[Geist93] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek and V. Sunderam, "PVM 3 user's guide and reference manual", *Oak Ridge National Laboratory TM-12187*, May 1993.

[Hać85] A. Hać and T. Johnson, "A study of dynamic load balancing in distributed systems", Technical report 85/15, The Johns Hopkins University, Baltimore, Maryland, 1985.

[Hać86] A. Hać, "A distributed algorithm for performance improvement through file replication, file migration and process migration", Technical Report R86/04, The Johns Hopkins University, Baltimore, Maryland, 1986.

[Huang86] H. C-Y. and J.W.S. Liu, "Dynamic load balancing algorithms in homogeneous distributed systems", Report no. R-86-1261, University of Illinois, Urbana, 1996. Also appeared in the *Proceedings of the 6th international conference on distributed computing systems*.

[Kaashoek92] M.F. Kaashoek, R. van Renesse, H. van Staveren and A.S. Tanenbaum, "FLIP: an internetwork protocol for supporting distributed systems", Vrije Universiteit Amsterdam, 1992.
The article can be obtained from:
ftp://ftp.cs.vu.nl/amoeba/papers/amoeba/tocs9?.ps.Z
It is accepted for publication by the *ACM Transactions on computer systems*.

[Kara92] M. Kara, "A coherent approach to cooperation for distributed dynamic load balancing algorithms", Research report series: Report 92.18, University of Leeds, July 1992.

[Krueger87] Ph. Krueger and M. Livny, "Load balancing, load sharing and performance in distributed systems", Technical report #700, University of Wisconsin-Madison, 1987.

[León93] J. León, A.L. Fisher and P. Steenkiste, "Fail-safe PVM: A portable package for distributed programming with transparent recovery", School of Computer Sciences Report 93-124, Carnegie Mellon University, Pittsburgh, February 1993.

[Lindh91] B. Lindh, "What is OSF/DCE?", IBM Report, Chalmers Science Park, Sven Hultins gata 9, Göteborg, Sweden, October 1991.

[Litzkow88] M.J. Litzkow, M. Livny and M.W. Mutka, "Condor - A hunter of idle workstations", *8th International Conference on Distributed Computing Systems*, San Jose, California, June 1988.

[Litzkow90] M.J. Litzkow and M. Livny. "Experience with the Condor distributed batch system", *Proceedings of the IEEE workshop on experimental distributed systems*, Huntsville, AL, 1990.

[Litzkow91] M.J. Litzkow, "Condor installation guide", version 4.1b, University of Wisconsin-Madison, September 1991.

[Litzkow92a] M.J. Litzkow and M. Solomon, "Supporting checkpointing and process migration outside the UNIX kernel", *Usenix winter conference*, San Francisco, California, 1992.

[Litzkow92b] M.J. Litzkow and M. Livny, "Using the Condor test suite (draft)", University of Wisconsin-Madison, July 1992.

[Liu92] Y-H. Liu and S. Handelman, "Checkpoint and restart function for parallel UNIX system", Research Report 18080, T.J. Watson Research Centre, Yorktown Heights, NY, June 1992.

[Livny82] M. Livny and M. Melman, "Load balancing in homogeneous broadcast distributed systems", *Proceedings of the ACM Computer network performance symposium*, pp. 47-55, April 1982.

[Lu89] C. Lu, "Process migration in distributed systems", Ph.D. Thesis, University of Illinois at Urbana-Champaign, 1989.

[Manzo91] W.A. Manzo, "Performance evaluation of checkpoint rollback-recovery algorithms in distributed systems", Technical Report R-91-1721, University of Illinois at Urbana-Champaign, 1991.

[Mullender89] S. Mullender (editor), "*Distributed systems*", Addison-Wesley, 1989.

[Mutka87] M.W. Mutka and M. Livny, "Profiling workstations' available capacity for remote execution", *Performance '87, Proceedings of the 12th IFIP WG 7.3 symposium on computer performance*, Brussels, December 1987.

[Nutt92] G.J. Nutt, "*Centralized and distributed operating systems*", Prentice-Hall, 1992.

[Rosenberry92] W. Rosenberry, D. Kenny and G. Fisher, "Understanding DCE", O'Reilly & Associates, Inc, 1992.

[Suen92] T.T.Y. Suen and J.S.K. Wong, "Efficient task migration algorithms for distributed systems", *IEEE Transactions on parallel and distributed systems* **3**, no. 4, pp. 488-499, July 1992.

[Sunderam90] V.S. Sunderam, "PVM: A framework for parallel distributed computing", *Concurrency: practice and experience*, vol. **2**(4), pp. 315-339, December 1990.

[Wang85] Y-T. Wang and R.J.T. Morris, "Load sharing in ditributed systems", *IEEE Transactions on computers*, vol. **C-34**, no. 3, March 1985.

[Xu93] J. Xu and K. Hwang, "Heuristic methods for dynamic load balancing in a message-passing multicomputer", *Journal of parallel and distributed computing*, vol. **18**, pp. 1-13, 1993.

APPENDICES

A REFERENCES TO PVM AND CONDOR

PVM:

- The article where PVM was introduced: "PVM: A framework for parallel distributed computing", can be found in [Sunderam90].
- In "Network-based concurrent computing", [Geist92], are the experiences with and enhancements to version 1.0 of PVM described.
- The "PVM 3 user's guide and reference manual", can be found in [Geist93].
- In the recently published article: "Integrated PVM framework supports heterogeneous network computing", referring to Linda, Express, P4 and PVM, the general field of heterogeneous network computing and some emerging research issues are discussed [Dongarra93].
- In the article: "Parallel distributed seismic migration", the results of porting a scientific parallel program - designed to run on network-connected IBM RISC/6000 workstations using RPC - to PVM and Linda are presented [Almasi93].
- The interest in PVM from the site of vendors is illustrated with: "PVM and HeNCE: traversing the parallel environment", which can be found in an issue of *Cray channels* [Beguelin92].
- Closely related with the work on DYNAMICPVM, the article: "Fail-safe PVM: A portable package for distributed programming with transparent recovery", reports on a major enhancement to PVM carried out by a third party [León93].

Condor:

- The Condor scheduling system was first presented in: "Condor - A hunter of idle workstations", [Litzkow88].
- The motivation for the Condor development can be found in: "Profiling workstations' available capacity for remote execution", [Mutka87].
- The checkpoint mechanism used in Condor is described in: "Supporting checkpointing and process migration outside the UNIX kernel", [Litzkow92a].
- In: "Experience with the Condor distributed batch system" results with Condor are reported, [Litzkow90].
- The Condor support documentation can be found in: "Condor installation guide", "Condor technical manual" and "Using the Condor test suite", [Litzkow91], [Bricker91] and [Litzkow92b] respectively.

- How a collection of Condor pools are turned into a Condor Flock is reported in: "Condor Flocking: load sharing between pools of workstations", [Evers93].

Related work:

- In "DAWGS - A distributed computer sever utilizing idle workstations", [Clark92], is reported on a system which has much in common with DYNAMICPVM, however DYNAMICPVM is implemented outside the UNIX kernel while DAWGS applies kernel hooks.

- Miron Livny and Jim Pruyne presented a paper entitled: "Scheduling PVM on workstation clusters using Condor". They have build a prototype system which uses Condor's scheduler to select a cluster for a PVM job from a large pool of workstations, brings up PVM daemon processes on each of these machines, and starts processes on each of the workstations as required by the user. This paper was presented at the "1993 PVM Users group meeting", held in Knoxville, May 1993.

B THE DYNAMICPVM DEVELOPMENT ENVIRONMENT

B.1 Directory structure

The main structure of the root directory and the subdirectories for DYNAMICPVM is as follows:

```
~/DynamicPVM/CONDOR/
                appl-devl/config/
                        pvm-ckpt/
                        pvm-normal/
                include/
                lib/
                pvm3/
```

The subdirectories 'CONDOR' and 'pvm3' have the same structure as the directories used by Condor and PVM. Directory 'appl-devl' contains the sources and required files for the 'make' utility. Directories 'include' and 'lib' contain the include files and libraries required for compiling and linking normal and 'checkpointable' PVM tasks.

B.2 Compiler/linker stanzas

```
* @(#) xlc.cfg 1.2 12/19/91 19:30:55
*
* COMPONENT_NAME: (CC) AIX XL C Compiler/6000
*
* FUNCTIONS: C Configuration file
*
* ORIGINS: 27
*
* (C) COPYRIGHT International Business Machines Corp. 1989, 1990, 1991
* All Rights Reserved
* Licensed Materials - Property of IBM
*
* US Government Users Restricted Rights - Use, duplication or
* disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
*


* standard c compiler
xlc:    use         = DEFLT
        crt         = /lib/crt0.o
        mcrt        = /lib/mcrt0.o
        gcrt        = /lib/gcrt0.o
        libraries   = -lc
        proflibs    = -L/lib/profiled,-L/usr/lib/profiled
        options     = -H512,-T512,-D_ANSI_C_SOURCE,-qansialias

* DynamicPVM pvmckpt compiler
```

```
Dmkcpcc:  use =  DEFLT
        crt         = /lib/crt0.o
        mcrt        = /lib/mcrt0.o
        gcrt        = /lib/gcrt0.o
        libraries   = -lbsd,-lc,-L/home3/kaldi0/DynamicPVM/CONDOR/lib, \
                      /home3/kaldi0/DynamicPVM/CONDOR/lib/libckpt.a, \
                      -bI:/lib/syscalls.exp
        proflibs    = -L/lib/profiled,- L/usr/lib/profiled
        options     = -H512,-T512, -qlanglvl=extended, -qnoro, -DDY_PVM, -D_BSD, \
                      -D_NONSTD_TYPES, -D_NO_PROTO, -D_BSD_INCLUDES, -bnodelcsect,\
                      -U__STR__, -U__MATH__,-bnoso


* DynamicPVM checkpointing compiler for PVM programs
Dpvmcpcc:  use  =  DEFLT
        crt         = /home3/kaldi0/DynamicPVM/CONDOR/lib/condor_rt0.o
        mcrt        = /lib/mcrt0.o
        gcrt        = /lib/gcrt0.o
        libraries   = -lbsd,-lc,-L/home3/kaldi0/DynamicPVM/lib, \
                      /home3/kaldi0/DynamicPVM/lib/libpvm3.a, \
                      /home3/kaldi0/DynamicPVM/lib/libckpt.a, \
                      -bI:/lib/syscalls.exp
        proflibs    = -L/lib/profiled,- L/usr/lib/profiled
        options     = -H512,-T512, -qlanglvl=extended, -qnoro, -DDY_PVM, \
                      -DDO_CKPT, -D_BSD, -D_NONSTD_TYPES, -D_NO_PROTO, \
                      -D_BSD_INCLUDES, -bnodelcsect, -U__STR__, -U__MATH__,-bnoso

DEFLT:  xlc     = /usr/lpp/xlc/bin/xlcentry
        as          = /bin/as
        ld          = /bin/ld
        options     = -D_IBMR2,-D_AIX,-bhalt:4
        ldopt       = "b:o:e:u:R:H:Y:Z:L:T:A:V:k:j:"
```

B.3 Utilities

The file: .alias, (called by .cshrc) below contains some helpful utilities. The shell command:

```
limit coredumpsize unlimited
```

is contained by: .cshrc, to allow the creation of a core dump file.

```
##############################################################################
# File: .alias,v 1.3 1993/08/19 13:02:24 Rel
#
.
.
#
#DynamicPVM stuff
alias Dpvmimk    '$home/bin/imake -s Makefile -DPROGRAM=\
                        !$-I'$home'/DynamicPVM/appl_devl/config'
alias pvmd3      '$home/DynamicPVM/pvm3/lib/RS6K/pvmd3'
alias pvmd       '$home/DynamicPVM/pvm3/lib/pvmd'
alias pvm        '$home/DynamicPVM/pvm3/lib/RS6K/pvm'
#
#PVM stuff
#
# append this file to your .cshrc to set path according to machine type.
# you may wish to use this for your own programs (edit the last part to
# point to a different directory f.e. ~/bin/_$PVM_ARCH.
#
if (! $?PVM_ROOT) then
        if (-d ~/DynamicPVM/pvm3) then
                setenv PVM_ROOT ~/DynamicPVM/pvm3
        else
                echo PVM_ROOT not defined
                echo To use PVM, define PVM_ROOT and rerun your .cshrc
        endif
endif

if ($?PVM_ROOT) then
        setenv PVM_ARCH `$PVM_ROOT/lib/pvmgetarch`
#
# uncomment the following line if you want the PVM executable directory
# to be added to your shell path.
#
        set path=($path $PVM_ROOT/lib/$PVM_ARCH $PVM_ROOT/bin/$PVM_ARCH)
endif
```

C BUILDING THE DYNAMICPVM CHECKPOINT FACILITY

C.1 Routine: MAIN()

ckpt_main.c 5 Nov 1993 15:10:17

amsi15 ~/DynamicPVM/CONDOR/R6000_AIX32/condor_syscall_lib_aix/

```
 1  /*
 2  ** Copyright 1986, 1987, 1988, 1989, 1990, 1991 by the Condor Design Team
 3  **
 4  ** Permission to use, copy, modify, and distribute this software and its
 5  ** documentation for any purpose and without fee is hereby granted,
 6  ** provided that the above copyright notice appear in all copies and that
 7  ** both that copyright notice and this permission notice appear in
 8  ** supporting documentation, and that the names of the University of
 9  ** Wisconsin and the Condor Design Team not be used in advertising or
10  ** publicity pertaining to distribution of the software without specific,
11  ** written prior permission.  The University of Wisconsin and the Condor
12  ** Design Team make no representations about the suitability of this
13  ** software for any purpose.  It is provided "as is" without express
14  ** or implied warranty.
15  **
16  ** THE UNIVERSITY OF WISCONSIN AND THE CONDOR DESIGN TEAM DISCLAIM ALL
17  ** WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES
18  ** OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE UNIVERSITY OF
19  ** WISCONSIN OR THE CONDOR DESIGN TEAM BE LIABLE FOR ANY SPECIAL, INDIRECT
20  ** OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
21  ** OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE
22  ** OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
23  ** OR PERFORMANCE OF THIS SOFTWARE.
24  **
25  ** Authors:  Allan Bricker and Michael J. Litzkow,
26  **           University of Wisconsin, Computer Sciences Dept.
27  */
28  /* DynamicPVM
29  ** Author: Leen Dikken
30  ** November 1993, Shell Research - Amsterdam
31  */
32  #include <stdio.h>
33  #include <signal.h>
34  #include <sys/types.h>
35  #include <sys/param.h>
36  #include <sys/file.h>
37  #include <sys/time.h>
38  #include <sys/resource.h>
39
40  #ifndef DY_PVM
41  #include "condor_sys.h"
42  #include "ckpt_file.h"
43  #endif DY_PVM
44  #include "trace.h"
45  #include "except.h"
46  #ifndef DY_PVM
47  #include "fileno.h"
48  #endif DY_PVM
49  #include "debug.h"
50
51  static char *_FileName_ = __FILE__;          /* Used by EXCEPT (see
except.h)  */
52
53  char          CkptName[ MAXPATHLEN ];
54
55  #ifndef DY_PVM
56  extern RESTREC  RestartInfo;
57
58  extern int      RunningAsCondor;
```

```
59  #endif DY_PVM
60
61  /* static int HasRun = 0; */
62  int HasRun = 0;
63
64  #ifdef VOID_SIGNAL_RETURN
65  #define sighandler_t void
66  #else
67  #define sighandler_t int
68  #endif
69
70
71  /*
72  **          Checkpoint Main routine
73  */
74  MAIN( argc, argv, envp )
75  int          argc;
76  char **argv;
77  char **envp;
78  {
79          sighandler_t                    CKPT();
80          struct sigvec                   action;
81          register int i;
82          extern int (*_EXCEPT_Cleanup)(), abort();
83
84  #ifndef DY_PVM
85  #ifdef CONDOR
86          RunningAsCondor = 1;
87  #endif CONDOR
88  #endif DY_PVM
89
90          if( HasRun ) {
91                  restart( CkptName );
92          }
93          HasRun = 1;
94
95  #ifndef DY_PVM
96          /*
97          **          Initialize table of open files
98          */
99          for( i = 0; i < NOFILE; i++ ) {
100                 RestartInfo.rr_file[i].fi_flags = 0;          /* Not open */
101                 RestartInfo.rr_file[i].fi_fdno = -1;          /* Map to invalid fd */
102         }
103         if( RunningAsCondor ) {
104                 InitStaticFile( RSC_SOCK, FI_WELL_KNOWN );
105                 InitStaticFile( CLIENT_LOG, FI_WELL_KNOWN );
106         } else {
107                 InitStaticFile( 0, FI_PREOPEN | FI_NFS );
108                 InitStaticFile( 1, FI_PREOPEN | FI_NFS );
109                 InitStaticFile( 2, FI_PREOPEN | FI_NFS );
110         }
111         /*
112         DumpOpenFds();
113         */
114
115         /*
116         **          Initial system calls should be local and unrecorded
117         */
118         (void) SetSyscalls( SYS_LOCAL | SYS_UNRECORDED );
119
120         /*
121         display_syscall_mode( __LINE__, __FILE__ );
122         */
123 #endif DY_PVM
124
125         action.sv_handler = CKPT;
126         action.sv_mask = 0;
127 #ifdef DYNIX
128         action.sv_onstack = 0;
129 #else DYNIX
130         action.sv_flags = 0;
131 #endif DYNIX
132
133         if( sigvec(SIGTSTP,&action,NULL) < 0 ) {
134                 EXCEPT( "can't set sigaction for TSTP" );
135         }
136
137         /*
138         **          Unblock any blocked signals (SIGTSTP in particular)
139         */
140         sigsetmask(0);
141
142 #ifndef DY_PVM
```

```
143             /*
144             **      Further system calls will be remote and recorded...
145             */
146             if( RunningAsCondor ) {
147                     RSC_Init( RSC_SOCK, CLIENT_LOG );
148                     dprintf_init( CLIENT_LOG );
149                     DebugFlags |= D_NOHEADER;
150                     (void) SetSyscalls( SYS_REMOTE | SYS_RECORDED );
151             } else {
152                     (void) SetSyscalls( SYS_LOCAL | SYS_RECORDED );
153             }
154             /*
155             DebugFlags |= D_SYSCALLS;
156             */
157     #endif DY_PVM
158
159     #ifdef DEBUG
160             DebugFlags = -1;
161     #endif DEBUG
162
163             (void) strcpy( CkptName, *argv );
164
165     #ifndef DY_PVM
166             /*
167             **      chdir has the side effect of setting the variable Condor_CWD
168             */
169
170             /*
171             display_syscall_mode( __LINE__, __FILE__ );
172             */
173
174             if( chdir(".") < 0 ) {
175                     EXCEPT("Initialize Condor_CWD");
176             }
177
178             /*
179             display_syscall_mode( __LINE__, __FILE__ );
180             */
181
182             if( RunningAsCondor ) {
183                     open_std_files( argv[1], argv[2], argv[3] );
184                     argv += 3;
185                     argc -= 3;
186             } else {
187                     PreOpen( fileno(stdin) );
188                     PreOpen( fileno(stdout) );
189                     PreOpen( fileno(stderr) );
190             }
191
192             /*
193             display_syscall_mode( __LINE__, __FILE__ );
194             */
195     #endif DY_PVM
196
197             _EXCEPT_Cleanup = abort;
198
199             /*
200             DumpOpenFds();
201             fflush( stdout );
202             */
203
204             return( main(argc, argv, envp) );
205     }
206
207     #ifndef DY_PVM
...     51 lines deleted
259     #endif DY_PVM
```

C.2 Imakefile.ed

```
Imakefile.ed                    5 Nov 1993          15:10:16

amsi15    ~/DynamicPVM/CONDOR/R6000_AIX32/condor_syscall_lib_aix/

  1       SRC_DIR = $(SRC_TREE)/condor_$(SYSCALL_LIB)
  2       CFLAGS = $(STD_C_FLAGS) -DCONDOR -g
  3
  4       TO_UPPER = $(PLATFORM_DIR)/condor_$(SYSCALL_LIB)/ToUpper
  5
  6       REM #define SourceFiles \
  7       REM     CONDOR_sysnames.c CONDOR_syscalls.c \
  8       REM     _mkckpt.c _updateckpt.c display.c ckpt.c restart.c syscall_mode.c \
  9       REM     xfer_file.c finfo.c ckpt_main.c map_file.c C_getenv.c \
 10       REM     stubs.c extern_path.c fake_getmnt.c nfs.c intern_path.c extern_name.c
 11
 12       #define SourceFiles \
 13                   _mkckpt.c _updateckpt.c display.c ckpt.c restart.c \
 14                   ckpt_main.c map_file.c C_getenv.c
 15
 16       REM #define UtilObjects \
 17       REM      ../condor_util_lib/blankline.o ../condor_util_lib/config.o \
 18       REM      ../condor_util_lib/do_connect.o ../condor_util_lib/dprintf.o \
 19       REM      ../condor_util_lib/dprintf_config.o ../condor_util_lib/except.o \
 20       REM      ../condor_util_lib/expr.o ../condor_util_lib/history.o \
 21       REM      ../condor_util_lib/job_queue.o ../condor_util_lib/ltrunc.o \
 22       REM      ../condor_util_lib/mkargv.o ../condor_util_lib/perror.o \
 23       REM      ../condor_util_lib/proc.o ../condor_util_lib/condor_config.o \
 24       REM      ../condor_util_lib/condor_errlst.o ../condor_util_lib/signames.o \
 25       REM      ../condor_util_lib/status.o ../condor_util_lib/strdup.o \
 26       REM      ../condor_util_lib/stricmp.o ../condor_util_lib/update_rusage.o
 27
 28       #define UtilObjects \
 29                   ../condor_util_lib/dprintf.o ../condor_util_lib/dprintf_config.o \
 30                   ../condor_util_lib/except.o ../condor_util_lib/condor_errlst.o
 31
 32       REM #define XdrObjects \
 33       REM      ../condor_xdr_lib/xdr_Init.o ../condor_xdr_lib/xdr_direntries.o \
 34       REM      ../condor_xdr_lib/xdr_expr.o ../condor_xdr_lib/xdr_fdset.o \
 35       REM      ../condor_xdr_lib/xdr_io.o ../condor_xdr_lib/xdr_itimerval.o \
 36       REM      ../condor_xdr_lib/xdr_mach_rec.o ../condor_xdr_lib/xdr_ports.o \
 37       REM      ../condor_xdr_lib/xdr_prio_rec.o ../condor_xdr_lib/xdr_proc.o \
 38       REM      ../condor_xdr_lib/xdr_ptr.o ../condor_xdr_lib/xdr_record.o \
 39       REM      ../condor_xdr_lib/xdr_rlimit.o ../condor_xdr_lib/xdr_rusage.o \
 40       REM      ../condor_xdr_lib/xdr_stat.o ../condor_xdr_lib/xdr_statfs.o \
 41       REM      ../condor_xdr_lib/xdr_status.o ../condor_xdr_lib/xdr_timeval.o \
 42       REM      ../condor_xdr_lib/xdr_timezone.o ../condor_xdr_lib/xdr_wait.o
 43
 44       REM #define ObjectFiles \
 45       REM     UtilObjects XdrObjects CONDOR_sysnames.o CONDOR_syscalls.o \
 46       REM     _mkckpt.o _updateckpt.o display.o ckpt.o restart.o syscall_mode.o \
 47       REM     xfer_file.o finfo.o ckpt_main.rem.o map_file.o \
 48       REM     extern_path.o fake_getmnt.o nfs.o intern_path.o \
 49       REM     extern_name.o stubs.o condor_shr.o
 50
 51       #define ObjectFiles \
 52                   UtilObjects _mkckpt.o _updateckpt.o display.o ckpt.o restart.o \
 53                   ckpt_main.loc.o map_file.o
 54       REM condor_shr.o
 55
 56       REM all_target( libcondor.a libckpt.a condor_rt0.o syscall.shr.o syscall.exp libfort.a
)
 57       all_target( libckpt.a condor_rt0.o libfort.a )
 58
 59       tags_target(SourceFiles,$(NULL))
 60       depend_target(SourceFiles)
 61       program_target(ToUpper,ToUpper.o,$(NULL),release)
 62
 63       #if IS_R6000_AIX32
 64       program_target(zap,zap.o,$(NULL),norelease)
 65       #endif
 66
 67       REM prev_library_target(UtilObjects,../condor_util_lib,$(SRC_DIR))
```

```
68      REM prev_library_target(XdrObjects,../condor_xdr_lib,$(SRC_DIR))
69
70      release::
71      REM       syscall.exp $(RELEASE_DIR)/lib
72                cp C_getenv.o $(RELEASE_DIR)/lib
73
74      CMNT
75      CMNT Create a replacement for libc.a by copying it and replacing all
76      CMNT the system calls with our own version.
77      CMNT
78      REM libcondor.a:    ObjectFiles
79      REM       -f libcondor.a
80      REM       cp $(LIBC) libcondor.a
81      REM    chmod u+w libcondor.a
82      REM    AR_DELETE( libcondor.a, shr.o )
83      REM    AR_REPLACE( libcondor.a, ObjectFiles )
84      REM release:: libcondor.a
85      REM    cp libcondor.a  $(RELEASE_DIR)/lib
86      REM    RANLIB_TOUCH( $(RELEASE_DIR)/lib/libcondor.a )
87      REM clean::
88      REM    rm -f libcondor.a ObjectFiles
89
90      CMNT
91      CMNT Create a replacement for libc.a to be linked with programs for
92      CMNT checkpointing, but not remote execution.  N.B. This is done differently
93      CMNT than other libraries in that we start out with a copy of libcondor.a,
94      CMNT and replace ckpt_main.rem.o with ckpt_main.loc.o, (checkpointing
95      CMNT only, no remote system calls).  Don't use the default rule for this one!
96      CMNT
97      REM libckpt.a: ../condor_$(SYSCALL_LIB)/libcondor.a ckpt_main.loc.o
98      REM    rm -f libckpt.a
99      REM    cp libcondor.a  libckpt.a
100     REM    chmod u+w libckpt.a
101     REM    AR_DELETE( libckpt.a, ckpt_main.rem.o )
102     REM    AR_REPLACE( libckpt.a, ckpt_main.loc.o )
103     REM release:: libckpt.a
104     REM    cp libckpt.a $(RELEASE_DIR)/lib
105     REM    RANLIB_TOUCH( $(RELEASE_DIR)/lib/libckpt.a )
106     REM clean::
107     REM    rm -f ckpt_main.rem.o libckpt.a
108
109     libckpt.a: ObjectFiles
110               rm -f libckpt.a
111               AR_REPLACE( libckpt.a, ObjectFiles )
112               chmod u+w libckpt.a
113     release:: libckpt.a
114               cp libckpt.a $(RELEASE_DIR)/lib
115               RANLIB_TOUCH( $(RELEASE_DIR)/lib/libckpt.a )
116     clean::
117               rm -f ckpt_main.loc.o libckpt.a
118
119     CMNT
120     CMNT Condor version of libc.a/shr.o.
121     CMNT Take the original shr.o and strip out the import from /unix.
122     CMNT This will force these symbols to be resolved from stubs.o which
123     CMNT will satisfy them with calls to syscall().
124     CMNT
125     REM #if IS_R6000_AIX31
126     REM condor_shr.o: $(LIBC)
127     REM    AR_EXTRACT( $(LIBC), shr.o )
128     REM    ld -o condor_shr.o \
129     REM       -bnoautoimp -r \
130     REM       shr.o
131     REM clean::
132     REM    rm -f shr.o
133     REM #else            /* IS_R6000_AIX32 */
134     REM condor_shr.o: $(LIBC) zap
135     REM     AR_EXTRACT( $(LIBC), shr.o )
136     REM     mv shr.o condor_shr.o
137     REM     zap condor_shr.o
138     REM clean::
139     REM     rm -f shr.o condor_shr.o
140     REM #endif
141
142     CMNT
143     CMNT Provide the syscall() routine.
144     CMNT Imports: all system calls from /unix
145     CMNT Exports: syscall() and CondorErrno
146     CMNT
147     REM syscall.shr.o: syscall.o syscall.exp
148     REM    ld -o syscall.shr.o \
149     REM       -bM:SRE -T512 \
150     REM       -bE:syscall.exp \
151     REM       -bI:$(SYSCALLS_EXP) \
```

```
152     REM       syscall.o
153     REM release::
154     REM     rm -f $(RELEASE_DIR)/lib/syscall.shr.o
155     REM     cp syscall.shr.o $(RELEASE_DIR)/lib
156     REM clean::
157     REM     rm -f syscall.shr.o syscall.o
158
159     CMNT
160     CMNT Be sure every Condor program contains one object compiled with "-g".
161     CMNT Otherwise there will not be a DEBUG section in the final executable,
162     CMNT and the checkpointing code will fail!
163     CMNT
164     CMNT In this version we compile with CONDOR defined, and get a
165     CMNT checkpointing plus remote system call version.
166     CMNT
167     REM ckpt_main.rem.o: ckpt_main.c
168     REM     $(CC) -c $(CFLAGS) -g ckpt_main.c
169     REM     mv ckpt_main.o ckpt_main.rem.o
170
171     CMNT
172     CMNT Be sure every Condor program contains one object compiled with "-g".
173     CMNT Otherwise there will not be a DEBUG section in the final executable,
174     CMNT and the checkpointing code will fail!
175     CMNT
176     CMNT In this version we compile without defining CONDOR, and get a
177     CMNT checkpoint only, (no remote system calls) version.
178     CMNT
179     ckpt_main.loc.o: ckpt_main.c
180                 $(CC) -c $(STD_C_FLAGS) -g ckpt_main.c
181                 mv ckpt_main.o ckpt_main.loc.o
182
183     CMNT
184     CMNT Replaces /lib/crt0.o in programs linked for execution with Condor
185     CMNT
186     condor_rt0.o: $(CRT0) $(TO_UPPER)
187                 $(TO_UPPER) $(CRT0) condor_rt0.o main MAIN
188     release::
189                 cp condor_rt0.o $(RELEASE_DIR)/lib
190     clean::
191                 rm -f condor_rt0.o
192
193     CMNT
194     CMNT The aix fortran library libxlf.a defines a getenv() routine which takes 2
195     CMNT arguments, the name of the environment variable, and buffer in which to
196     CMNT put the associated value.
197     CMNT
198     CMNT The aix C library libc.a defines a getenv routine which takes 1 argument,
199     CMNT the name of the environment variable, and it returns a pointer to the
200     CMNT associated value.
201     CMNT
202     CMNT Condor programs need to be linked with the condor libraries, which will
203     CMNT end up calling getenv(), and expecting the C version.  We therefore
204     CMNT supply our own C version, so that we can get it linked in before the
205     CMNT fortran version gets linked.
206     CMNT
207     CMNT Of course this will break any fortran programs which call getenv and
expect
208     CMNT the fortran version!  If some AIXpert knows how to fix this, I would be
209     CMNT very happy to learn about it. -- mike
210     CMNT
211     library_target(libfort.a,C_getenv.o,release)
```

C.3 Program: pvmckpt

```
#define D_CKPT          (1<<3)

/* ARGSUSED */
main( argc, argv, envp )
int argc;
char **argv, **envp;
{
        int DebugFlags;

        if ( argc > 3 )
                DebugFlags |= D_CKPT;

        if( argc == 3 ) {
                _mkckpt( argv[1], argv[2] );
        } else {
                _updateckpt( argv[1], argv[2], argv[3] );
        }

        exit( 0 );
}
```