Technical Memorandum

Strong Mobile Agent Architecture based on the Beowulf System

Written and Prepared by

Sungwoo Tak

Passakon Prathombutr

Donghoon Lee

E.K. Park

Jerrold Stach

Table of Contents

1 Beowulf Architectures

1.1 Introduction

Beowulf is a multi-computer architecture which can be used for parallel computations. A Beowulf cluster is a computer system conforming to the Beowulf architecture, which consists of one master node and one or more compute nodes. The nodes are connected together via Ethernet or some other network, and are typically built using commodity hardware components, such as any PC capable of running Linux and standard Ethernet adapters. The nodes usually do not contain any custom hardware components and are trivially reproducible. The master node controls the entire cluster and serves parallel jobs and their required files to the compute nodes. The master node is typically the cluster's administration console and its gateway to the outside world. In most cases the compute nodes in a Beowulf cluster are "dumb", they are configured and controlled by the master node. Typically, these compute nodes do not have keyboards or monitors and are accessed remotely from the master node. Simply put, Beowulf is a technology of clustering Linux computers together to form a parallel, virtual supercomputer, a Beowulf cluster. While Linux-based Beowulf clusters provide a cost effective hardware alternative to the supercomputers of the past for high performance computing applications, the original software implementations for Linux Beowulfs were not without their problems [1].

1.1.1 Beowulf System Software

[2] reports the components of Beowulf system software.

- ❖ BPROC: Beowulf Distributed Process Space

This packages allows a process ID space to span multiple nodes in a cluster environment and also provides mechanisms for starting processes on other nodes. (This package is for Linux 2.2.x kernels.)

- ❖ Network Device Drivers

We have long contributed to the development of the Linux networking code. Many of the Linux ethernet device drivers, and most of the device drivers for cost-effective high-performance network adapters were written by Donald Becker at CESDIS. A large portion of his time goes into maintaining and enhancing the performance of these device drivers. Fast Ethernet has long been a cluster staple, while Gigabit Ethernet has recently been tested.

- Beowulf Ethernet Channel Bonding

One of the goals of the goals of the Beowulf project is to demonstrate scalable I/O using commodity subsystems. For scaling network I/O we devised a method to join multiple low-cost networks into a single logical network with higher bandwidth.

- PVM-TCL

A set of extensions to TCL that allow you to manipulate the PVM virtual machine directly. NOTE: This currently does not extend the message-passing calls to TCL, just the PVM configuration ones. One could, however, implement them from the base provided here.

- Virtual Memory Pre-Pager

A loadable kernel module for Linux 2.0 that implements a non-blocking page read in system call. With the proper run-time support, this new system call can dramatically reduce the run-time of "out-of-core" programs (programs that have data sets larger than the available physical memory) by allowing multiple pages to be read from disk prior to their actual use.

- PPro Performance Counter Patches

Kernel patches (for 2.0.36, 2.2.2 and 2.2.9) and a small library to add kernel support for the performance counters found in the Pentium Pro and Pentium II.

- LM78 Hardware Monitor Driver

A loadable kernel module for Linux 2.0 and 2.1 that provides a /proc interface for the LM78 Hardware Monitor. (Note: This driver speaks to the hardware monitors via the ISA interface. This driver does not support boards that only have the serial interface on the hardware monitor connected.)

- Intel PR440FX Netbooting Tools

Tools to assist with netbooting from built in network interface on the Intel PR440FX and possibly other AMI Bios motherboards.

1.1.2 Scyld Beowulf System Software

The following is a list of the major software components distributed with the Scyld Beowulf Cluster Operating System [1].

- bproc - the Beowulf process migration technology; an integral part of Scyld Beowulf
- beosetup - a GUI interface for configuring the cluster
- beostatus - a GUI interface for monitoring cluster status
- beostat - a text-based tool for monitoring cluster status
- beoboot - a set of utilities for booting the compute nodes
- beofdisk - a utility for remote partitioning of hard disks on the compute nodes
- beoserv - the beoboot server; it responds to compute nodes and serves the boot image
- bpmaster - the bproc master daemon; it only runs on the master node
- bpslave - the bproc compute daemon; it runs on each of the compute nodes
- bpstat - a bproc client; it maintains status information for all nodes in the cluster
- bpctl - a bproc client; a command line mechanism for controlling the nodes
- bpsh - a bproc client; a replacement utility for "rsh" (remote shell)
- bpcp - a bproc client; a mechanism for copying files between nodes
- beompi - the Message Passing Interface; optimized for use with Scyld Beowulf
- beopvm - the Parallel Virtual Machine; optimized for use with Scyld Beowulf
- mpprun - a parallel job creation package for Scyld Beowulf
- perf - support for platform specific hardware performance counters

1.2 BPROC (Beowulf Distributed Process Space)

Scyld Beowulf is able to provide a single system image through its use of BProc, the Beowulf cluster process management kernel enhancement. BProc enables the processes running on cluster compute nodes to be visible and manageable on the master node. Processes start on the master node and are migrated to the appropriate compute node by BProc. Process parent-child relationships and UNIX job control information are both maintained with migrated tasks. Because cluster compute nodes are not required to contain resident applications, their hard disks are available for application data and cache. This approach eliminates both the need to have full installations on the compute nodes and the version skew problem common with previous generation cluster software.

Shortly, BPROC provides a mechanism to start processes on remote hosts while keeping them visible in the process tree on the front end of a cluster.
Bproc aims to provide a single process space within the tightly controlled environment of a beowulf cluster. Bproc doesn't address resource allocation or load balancing at all.
Bproc should avoid most if not all of the performance penalties associated with Mosix style migration.

1.2.1 Goal

The goal of Bproc is to provide key elements needed for a single system image on Beowulf cluster. Currently Beowulf style clusters still look like a collection of PC's on a network. Once logged into the front end of the cluster, the only way to start processes on other nodes in the system is via rsh. MPI and PVM hide this detail from the user but users but it's still there when either of them starts up. Cleaning up after jobs is often made tedious by this as well, especially when the jobs are misbehaving.

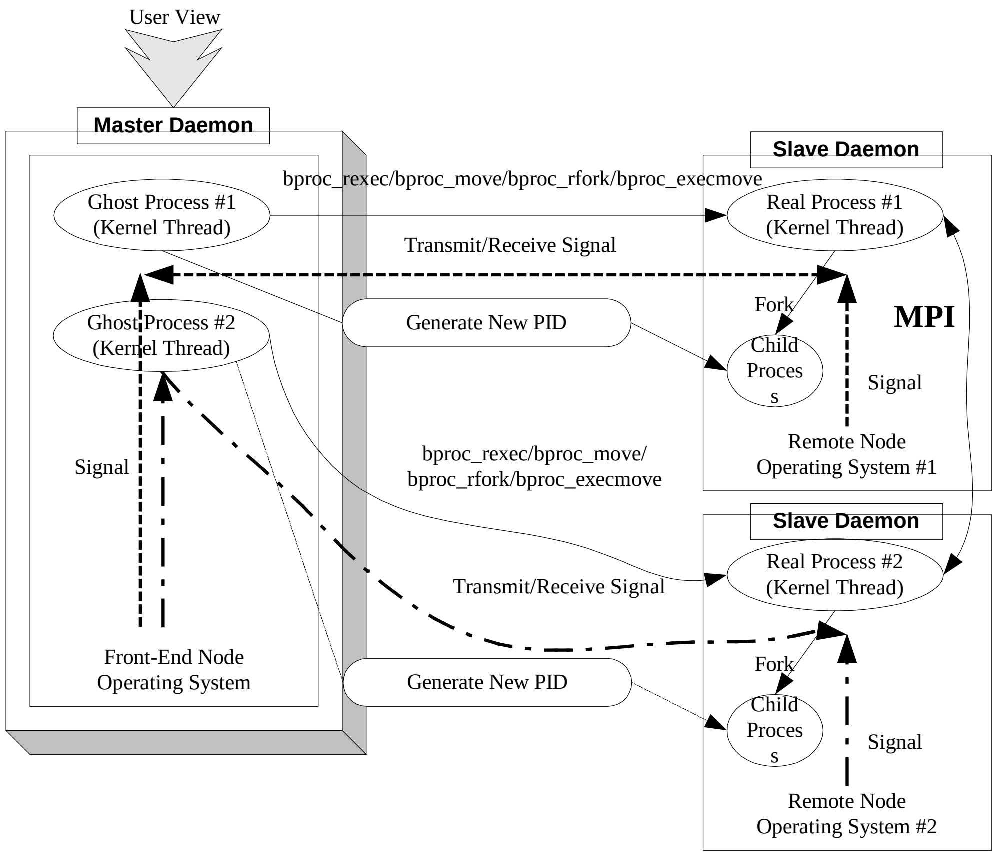
The bproc distributed PID space (bproc) addresses these issues by providing a mechanism to start processes on remote nodes without ever logging into another node and by making all the remote processes visible in the process table of the cluster's front-end node. The hope is that this will eliminate the need for people to be able to login on the nodes of a cluster.

Bproc can play a role of “Global Naming Scheme” in the mobile agent architecture. It can provide the environment of weak mobility in mobile software architectures.

1.2.2 Overview

BPROC introduces a distributed process ID (PID) space. This allows a node to run processes which appear in its process tree even though the processes are physically present on other nodes. The remote processes also appear to be part of the PID space of the front end node and not the node which they are running on. The node which is distributing its pid space is called the master and other nodes running processes for the master are the slaves. Each PID space has exactly one master and zero or more slaves. Each PID space corresponds to a real PID space on some machine. Therefore, each machine can be the master of only one PID space. A single machine can be a slave in more than one PID space.

1.2.3 Ghost processes



User View
Master Daemon
Ghost Process #1 (Kernel Thread)
Ghost Process #2 (Kernel Thread)
Signal
Front-End Node Operating System
bproc_rexec/bproc_move/bproc_rfork/bproc_execmove
Transmit/Receive Signal
Generate New PID
bproc_rexec/bproc_move/ bproc_rfork/bproc_execmove
Transmit/Receive Signal
Generate New PID
Slave Daemon
Real Process #1 (Kernel Thread)
Fork
Child Proces s
Signal
MPI
Remote Node Operating System #1
Slave Daemon
Real Process #2 (Kernel Thread)
Fork
Child Proces s
Signal
Remote Node Operating System #2

Figure 1. Brpoc Architecture

Remote processes on are represented on the master node by "ghost" processes. These are kernel threads like any other kernel thread on the system (i.e. nfsiod, kswapd, etc). They have no memory space, open files, or file system context but they can wake up for signals or other events and do things in kernel space. Using these threads, the existing signal handling code and process management code remains unchanged. Ghosts perform these basic functions:

- Signals they receive are forwarded to the real processes they represent. Since they are kernel threads, even SIGKILL and SIGSTOP can be caught and forwarded without destroying or stopping the ghost..
- When the remote process exits it will forward its exit code back to the ghost and the ghost will also exit with the same code. This allows other processes to wait() and receive meaningful exit status for remote processes.
- When a remote process wants to fork, it will need to obtain a PID for the new child process from the master node. This is obtained by asking the ghost process to fork and return the PID of the new child ghost process. (This also keeps the parent-child relationships in sync.)
- When a remote process waits on a child, the ghost will do the same. This prevents accumulation of ghost zombies and keeps the process trees in sync.

1.3 MPI (Message Passing Interface)

MPI is a library specification for writing message-passing programs for parallel computers, clusters and heterogeneous networks e.g., a Beowulf where parallel programming requires the MPI library run on a cluster. The MPI provides library of message-passing programs written in C, C++ or Fortran across a various heterogeneous parallel architectures.

The MPI becomes a de facto standard for portable message-passing parallel programs standardized by the MPI Forum and available on all massively-parallel supercomputers. The MPI forum designs the base set of routines that ones can implement efficiently, practical, portable, efficient, and flexible. The current version of MPI is MPI-2 which is open to public at http://www.mcs.anl.gov/mpi . The vendors that implements MPI include IBM, Intel, TMC, Meiko, Cray, Convex and Ncube.

The MPI contains point-to-point message passing, collective communication, support for process groups, support for communication contexts, support for application topologies, environmental inquiry routines, profiling interface and error control. The MPI is primarily for SPMD/MIMD parallel architectures but there is no mechanism for loading code onto processors, or assigning processes to processors or creating/destroying processes. Besides, the MPI does not provide remote memory transfers, multithreading and virtual shared memory but it is designed to be thread-safe.

In MPI, a process is defined in a group and a rank (a unique integer for labeling each process in the group). Process groups can be created and destroyed. A message label is specified by a context and a corresponding tag. In a point-to-point communication, it is a communication between pairs of processes. The Message selectivity is by rank and tag. In Collective communication, it involves all processes in the scope of the communication specified by the communicator. The collective communication routines do not take message tag arguments.

The feature of MPI includes six basic functions and 125 extension functions. The six basic functions are

- ❖ MPI_Init – start MPI.
- ❖ MPI_Finalize – exit MPI.
- ❖ MPI_Comm_size – the number of processes.
- ❖ MPI_Comm_rank – a number between zero and size-1.
- ❖ MPI_Bcast – this routine sends data from one process to all others.
- ❖ MPI_Reduce – this routine combines data from all processes and returning the result to a single process.

The implemented software versions of MPI include MPICH [3] and LAM [4]. They are widely deployed in the researches.

The MPICH is an open-source, portable implementation of the Message-Passing Interface Standard libraries. It contains a complete implementation of version 1.2 of the MPI Standard and also significant parts of MPI-2, particularly in the area of parallel I/O. MPICH is developed under Mathematics and Computer Science Division Argonne National Laboratory. The current version is 1.2.3 (January 2002) [3]. The MPICH is available in all UNIX and Window NT/2000 platforms.

The LAM (Local Area Multicomputer) is an MPI programming environment and development system for heterogeneous computers on a network. With LAM, a dedicated cluster or an existing network computing infrastructure can act as one parallel computer solving one problem. LAM features extensive debugging support in the application development cycle and peak performance for production applications. LAM features a full implementation of the MPI communication standard [4]. The LAM is developed at the Ohio Supercomputer, the University of Notre Dame.

1.4 Current projects related to Beowulf

The hybrid cluster computing is the computing based on the platform of hybrid clusters, which is the computer cluster consisting of both the stationary and mobile computers, interconnected by wireless and wired networks [5]. It implements a prototype for the hybrid cluster computer with Java mobile objects and the mobile IP. It studies the performance on the trade-off between the communication load and the computational load.

Mobile agents (MA) are of growing interest as base for distributed and parallel applications to achieve an efficient utilization of cluster systems. The MAs are mobile and autonomous SW units that can execute tasks given to the system and allocate independently all the needed resources. However, with growth of cluster sizes, the probability of a failure of system components increases. Holger Pals, Stefan Petri, and Claus Grewe [6] conducted the Fault Tolerance for Mobile Agents in Clusters (FANTOMAS). It focused on the failure of one or more system components and the loss of mobile agents. The FANTOMAS concept has been derived to offer a user transparent fault tolerance that can be activated on request, according to the needs of the task.

Another paper introduces a new SW system model for improving the performance of parallel and distributed applications adaptively and on a real-time base, using intelligent agents as adaptive controllers [7]. These intelligent agents are responsible for collecting and analyzing the performance parameters and metrics, and deciding on the required modification.

[8] develops the Scalable Computing Environment (SCE). The SCE is the Software (SW) tool with a cluster builder tool, complex system mgmt tool (SCMS), scalable real-time monitoring, web based monitoring SW (KCAP), parallel Unix command, and batch scheduler (SQMS).

Mobile agent techniques for autonomous data process and information discovery on the Synthetic Aperture Radar Atlas (SARA) digital library enable automatic and dynamic configuration of distributed parallel computing and support on-demand processing of such a remote-sensing archive efficiently [9]. It provides the architecture design and implementation status of the prototype system.

Another attempt to implement the MAs system on Beowulf Cluster has been reported at the Electrical Engineering Conference (EECON-22) in 1999. It has deployed on the SMILE Beowulf Cluster environment [10].

2 Distributed Software Tools for Process Migration

- ❖ Problem Definition

Given a mobile agent application in which each agent has to visit multiple resources in order to complete a task, find a schedule for giving agents access to resources that optimizes the system throughput

- ❖ Introduction

Mobile-agent, multi-agent, multi-resource scheduling has some similarities with scheduling in traditional computing environments, but there are major differences. Many assumptions used in traditional scheduling algorithms become unrealistic in the context of mobile agent systems, which are characterized by large data-transfer delay, diversified network links and a wide spectrum of machine speeds. Scheduling algorithms for a mobile-agent system must work in a heterogeneous environment where

(1) the number of machines is limited;

(2) the task graph structure is general;

(3) the data transfer delay is general; and

(4) the task duplication is not allowed.

This problem is NP-complete.

- ❖ A hierarchical scheduling framework in heterogeneous environment.
- ❖ Algorithms for scheduling multi-task

The objective is to optimize the system throughput. The algorithms work in heterogeneous networks in that they assume different host speeds and different data transfer and communication delays between host pairs. They developed both centralized and distributed algorithms. The centralized algorithm has a provable performance bound and is used as a module in the distributed scheduler.

2.1 CONDOR

Condor is sophisticated and unique distributed job scheduler developed by the condor research project at the University of Wisconsin-Madison Department of Computer Sciences. Condor exists to address problems of resource allocation over very large numbers of systems owned by different people. It includes some process migration capabilities as well [12].

2.1.1 Features of Condor

- Checkpoint and migration.

Where programs can be linked with Condor libaries, users of Condor may be assured that their jobs will eventually complete, even in the ever changing environment that Condor utilizes. As a machine running a job submitted to Condor becomes unavailable, the job can be checkpointed. The job may continue after migrating to another machine. Condor's periodic checkpoint feature periodically checkpoints a job even in lieu of migration in order to safeguard the accumulated computation time on a job from being lost in the event of a system failure such as the machine being shutdown or a crash.

- Remote system calls.

Despite running jobs on remote machines, the Condor standard universe execution mode preserves the local execution environment via remote system calls. Users do not have to worry about making data files available to remote workstations or even obtaining a login account on remote workstations before Condor executes their programs there. The program behaves under Condor as if it were running as the user that submitted the job on the workstation where it was originally submitted, no matter on which machine it really ends up executing on.

- No Changes Necessary to User's Source Code.

No special programming is required to use Condor. Condor is able to run non-interactive programs. The checkpoint and migration of programs by Condor is transparent and automatic, as is the use of remote system calls. If these facilities are desired, the user only re-links the program. The code is neither recompiled nor changed.

- Pools of machines can be hooked together.

Flocking is a feature of Condor that allows jobs submitted within a first pool of Condor machines to execute on a second pool. The mechanism is flexible, following requests from the job submission, while allowing the second pool, or a subset of machines within the second pool to set policies over the conditions under which jobs are executed.

- Jobs can be ordered.

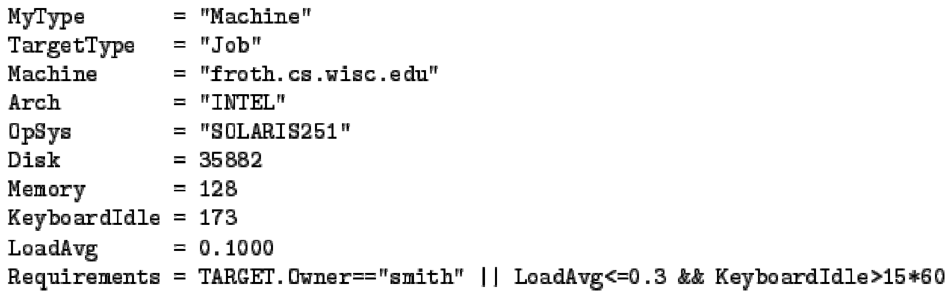
The ordering of job execution required by dependencies among jobs in a set is easily handled. The set of jobs is specified using a directed acyclic graph, where each job is a node in the graph. Jobs are submitted to Condor following the dependencies given by the graph.

- Sensitive to the desires of machine owners.

The owner of a machine has complete priority over the use of the machine. An owner is generally happy to let others compute on the machine while it is idle, but wants it back promptly upon returning. The owner does not want to take special action to regain control. Condor handles this automatically.

- ClassAds.

The ClassAd mechanism in Condor provides an extremely flexible, expressive framework for matchmaking resource requests with resource offers. Users can easily request both job requirements and job desires. For example, a user can require that a job run on a machine with 64 Mbytes of RAM, but state a preference for 128 Mbytes, if available. A workstation owner can state a preference that the workstation runs jobs from a specified set of users. The owner can also require that there be no interactive workstation activity detectable at certain hours before Condor could start a job. Job requirements/preferences and resource availability constraints can be described in terms of powerful expressions, resulting in Condor's adaptation to nearly any desired policy.



```
MyType       = "Machine"
TargetType   = "Job"
Machine      = "froth.cs.wisc.edu"
Arch         = "INTEL"
OpSys        = "SOLARIS251"
Disk         = 35882
Memory       = 128
KeyboardIdle = 173
LoadAvg      = 0.1000
Requirements = TARGET.Owner=="smith" || LoadAvg<=0.3 && KeyboardIdle>15*60
```

Figure 2: An Example of ClassAd

2.1.2 Condor Architecture

A Condor pool is comprised of a single machine that serves as the central manager, and an arbitrary number of other machines that have joined the pool. Conceptually, the pool is a collection of resources (machines) and resource requests (jobs). The role of Condor is to match waiting requests with available resources. Every part of Condor sends periodic updates to the central manager, the centralized repository of information about the state of the pool. Periodically, the central manager assesses the current state of the pool and tries to match pending requests with the appropriate resources.

- ❖ Central Manager

There can be only one central manager for your pool. The machine is the collector This machine plays a very important part in the Condor pool and should be reliable. If this machine crashes, no further matchmaking can be performed within the Condor system (although all current matches remain in effect until they are broken by either party involved in the match). Therefore, choose for central manager a machine that is likely to be online all the time, or at least one that will be rebooted quickly if something goes wrong. The central manager will ideally have a good network connection to all the machines in your pool, since they all send updates over the network to the central manager. All queries go to the central manager.

- ❖ Execute

Any machine in your pool (including your Central Manager) can be configured for whether or not it should execute Condor jobs. Obviously, some of your machines will have to serve this function or your pool won't be very useful. Being an execute machine doesn't require many resources at all. About the only resource that might matter is disk space, since if the remote job dumps core, that file is first dumped to the local disk of the execute machine before being sent back to the submit machine for the owner of the job. However, if there isn't much disk space, Condor will simply limit the size of the core file that a remote job will drop. In general the more resources a machine has (swap space, real memory, CPU speed, etc.) the larger the resource requests it can serve. However, if there are requests that don't require many resources, any machine in your pool could serve them.

- ❖ Submit

Any machine in your pool (including your Central Manager) can be configured for whether or not it should allow Condor jobs to be submitted. The resource requirements for a submit machine are actually much greater than the resource requirements for an execute machine. First of all, every job that you submit that is currently running on a remote machine generates another process on your submit machine. So, if you have lots of jobs running, you will need a fair amount of swap space and/or real memory. In addition all the checkpoint files from your jobs are stored on the local disk of the machine you submit from. Therefore, if your jobs have a large memory image and you submit a lot of them, you will need a lot of disk space to hold these files. This disk space requirement can be somewhat alleviated with a checkpoint server (described below), however the binaries of the jobs you submit are still stored on the submit machine.

- ❖ Checkpoint Server

One machine in your pool can be configured as a checkpoint server. This is optional, and is not part of the standard Condor binary distribution. The checkpoint server is a centralized machine that stores all the checkpoint files for the jobs submitted in your pool. This machine should have lots of disk space and a good network connection to the rest of your pool, as the traffic can be quite heavy. Now that you know the various roles a machine can play in a Condor pool, we will describe the actual daemons within Condor that implement these functions.

2.1.3 Condor Daemons

The following list describes all the daemons and programs that could be started under Condor and what they do:

- ❖ condor_master

This daemon is responsible for keeping all the rest of the Condor daemons running on each machine in your pool. It spawns the other daemons, and periodically checks to see if there are new binaries installed for any of them. If there are, the master will restart the affected daemons. In addition, if any daemon crashes, the master will send e-mail to the Condor Administrator of your pool and restart the daemon. The *condor_master* also supports various administrative commands that let you start, stop or reconfigure daemons remotely. The *condor_master* will run on every machine in your Condor pool, regardless of what functions each machine are performing.

- ❖ condor_startd

This daemon represents a given resource (namely, a machine capable of running jobs) to the Condor pool. It advertises certain attributes about that resource that are used to match it with pending resource requests. The startd will run on any machine in your pool that you wish to be able to execute jobs. It is responsible for enforcing the policy that resource owners configure which determines under what conditions remote jobs will be started, suspended, resumed, vacated, or killed. When the startd is ready to execute a Condor job, it spawns the *condor_starter*, described below.

- ❖ condor_starter

This program is the entity that actually spawns the remote Condor job on a given machine. It sets up the execution environment and monitors the job once it is running. When a job completes, the starter notices this, sends back any status information to the submitting machine, and exits.

- ❖ condor_schedd

This daemon represents resources requests to the Condor pool. Any machine that you wish to allow users to submit jobs from needs to have a *condor_schedd* running. When users submit jobs, they go to the schedd, where they are stored in the job queue, which the schedd manages. Various tools to view and manipulate the job queue (such as *condor_submit*, *condor_q*, or *condor_rm*) all must connect to the schedd to do their work. If the schedd is down on a given machine, none of these commands will work.

The schedd advertises the number of waiting jobs in its job queue and is responsible for claiming available resources to serve those requests. Once a schedd has been matched with a given resource, the schedd spawns a *condor_shadow* (described below) to serve that particular request.

- ❖ condor_shadow

This program runs on the machine where a given request was submitted and acts as the resource manager for the request. Jobs that are linked for Condor's standard universe, which perform remote system calls, do so via the *condor_shadow*. Any system call performed on the remote execute machine is sent over the network, back to the *condor_shadow* which actually performs the system call (such as file I/O) on the submit machine, and the result is sent back over the network to the remote job. In addition, the shadow is responsible for making decisions about the request (such as where checkpoint files should be stored, how certain files should be accessed, etc).

❖ condor_collector

This daemon is responsible for collecting all the information about the status of a Condor pool. All other daemons (except the negotiator) periodically send ClassAd updates to the collector. These ClassAds contain all the information about the state of the daemons, the resources they represent or resource requests in the pool (such as jobs that have been submitted to a given schedd). The *condor_status* command can be used to query the collector for specific information about various parts of Condor. In addition, the Condor daemons themselves query the collector for important information, such as what address to use for sending commands to a remote machine.

❖ condor_negotiator

This daemon is responsible for all the match-making within the Condor system. Periodically, the negotiator begins a negotiation cycle, where it queries the collector for the current state of all the resources in the pool. It contacts each schedd that has waiting resource requests in priority order, and tries to match available resources with those requests. The negotiator is responsible for enforcing user priorities in the system, where the more resources a given user has claimed, the less priority they have to acquire more resources. If a user with a better priority has jobs that are waiting to run, and resources are claimed by a user with a worse priority, the negotiator can preempt that resource and match it with the user with better priority.

❖ condor_kbdd

This daemon is only needed on Digital Unix and IRIX. On these platforms, the *condor_startd* cannot determine console (keyboard or mouse) activity directly from the system.

❖ condor_ckpt_server

This is the checkpoint server. It services requests to store and retrieve checkpoint files. If your pool is configured to use a checkpoint server but that machine (or the server itself is down) Condor will revert to sending the checkpoint files for a given job back to the submit machine.

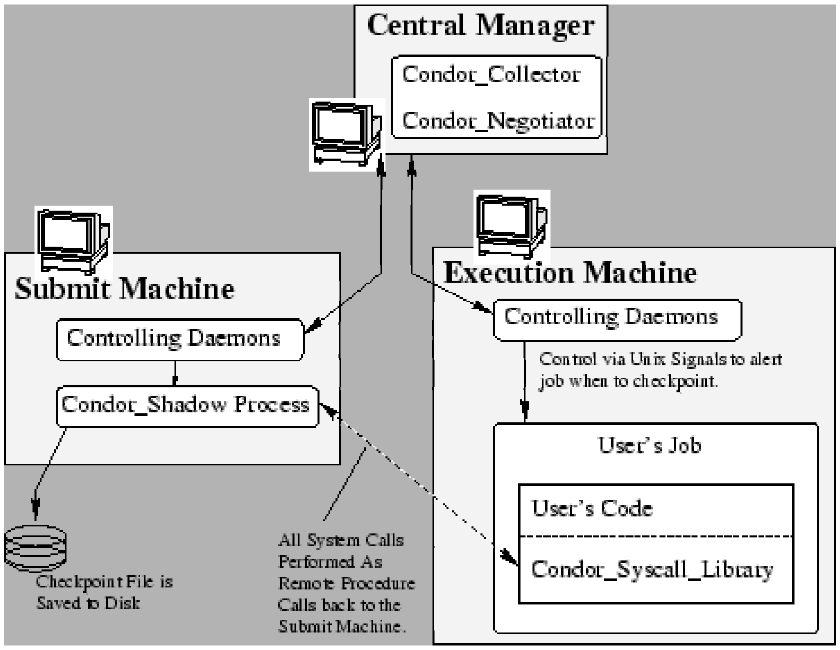


Figure 3: Condor Architecture

2.1.4 Submitting Different Types of Jobs: Alternative Universes

A Universe in condor defines an execution environment. Condor supports the following Universes:

- ❖ Vanilla
- ❖ MPI: The MPI Universe allows parallel program written with MPI to be managed by Condor
- ❖ PVM
- ❖ Globus
- ❖ Scheduler: DAGMan Scheduler
- ❖ Standard:
  - Transparent process checkpoint and restart
  - Transparent process migration
  - Remote system calls
  - Configurable file I/O buffering

2.2 MAUI

Maui is a batch scheduler capable of administrative control over resources, such as processors, memory and disk, and workload. It allows a high degree of configuration in the areas of job prioritization, scheduling, allocation, fairness, fairshare, QOS levels, backfill and reservation policies. The Maui is an advance cluster schedule suited for high performance computing (HPC) platforms including PC clustering like Beowulf. The Maui itself is not a resource manager but it makes decisions by querying and controlling a resource management system such as OpenPBS, PBSPro, Loadleveler, SGE, etc. For example, the Maui may query jobs and nodes information from the PBS server and direct PBS to manage job in response with specified Maui policies, priorities, and reservations. In the PBS users' view, the Maui is a set of external commands which provide additional information and capabilities intended to improve the user's ability to run jobs "when", "where", and "how" they want. It term of quality of service, Maui allows a user to request improved job turnaround time, access to additional resources, or exemptions to particular policies automatically

2.2.1 Brief history

The Maui scheduler was originally developed to be dependent on the IBM SP Load-Leveler API. As interest in the Maui scheduler for Linux, IRIX, HP-UX and Windows NT grew it was necessary either to write the interfaces to existing Resource Managers, or the develop a Resource Manager (RM) specifically for the Maui Scheduler. In 1998, a Maui High Performance Computing Center team started to develop a generic resource manager called Wiki, from what was the Wiki RM. The Linux Resource Manager development began at the Albuquerque High Performance Computing Center and is know as the Linux Resource Manager. The Maui Scheduler is an advanced reservation based High Performance Computing batch scheduler supported on SP, O2K, and Linux clusters. It can be used to extend the functionality and improve the efficiency of sites utilizing the PBS and Loadleveler batch system.

2.2.2 Maui Features

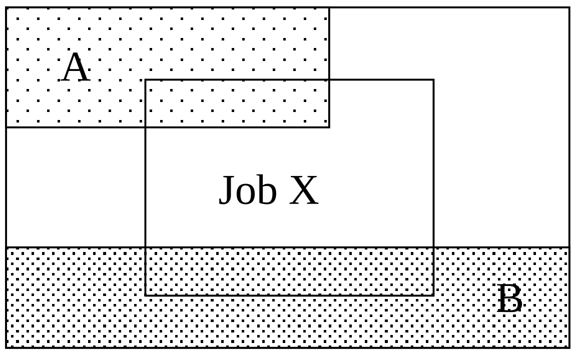
❖ Backfill

Backfill is one of scheduling approaches that allows user to run some jobs out of the queue order as long as they do not delay the highest priority jobs in the queue. Each job must be defined the estimate running time called wallclock limit. It is an estimation of the elapsed time from start to finish of the job. Since the scheduler may be configured to kill jobs which exceed their wallclock limits, it is often wise to slightly overestimate this limit. With this information, it allows Maui to determine whether or not a high priority job will be delayed. The more accurate the wallclock limit, the more 'holes' Maui can find to start the desired job early.

The backfill increase the utilization and throughput because it schedules job to the available resource for immediate use while decreasing the average job queue time. The command *showbf* in the Maui is to show a backfill window i.e., showing the available resources for immediate use. Users are able to configure a job that will be able to run as soon as it is submitted by utilizing only available resources.

❖ Advance Reservations

Maui provides the advance reservations by reservation-specific access control list (ACL) to specify the reserved resource and users who can use them. Also it allows setup of timeframe for certain resources to be used in particular site. Maui will attempt to locate the best possible combination of available resources whether these are reserved or unreserved without forcing the job to utilize the resources. For example, in the figure below, note that job X, which meets access criteria for both reservation A and B, allocates a portion of its resources from each reservation and the remainder from resources outside of both reservations.



Maui can configure jobs to be run within accessible reservations on a job-by-job basis or by the QoS constraints depending on the policy.

- ❖ Quality of Service (QOS)

The Maui QOS features allow a site to grant special privileges to particular users including the access to additional resources, exemptions from certain policies, the access to special capabilities, and improved job prioritization.

- ❖ Statistics

The Maui statistics features allow users to track the data that determine how well and how often their jobs are running. The *showstats* command provides detailed statistics per user, per group, and per account basis. Besides, the command *showgrid* displays various tables of scheduling or system performance. Therefore, user can determine what types of jobs the gain the highest performance and tune the jobs to optimal turnaround time.

- ❖ Diagnosis

The command *checkjob* in Maui allow users to view a detailed status report for each submitted job. This command shows all job attribute and state information and analyze if the job can run or not. If the job is unable to run, it will provide the reasons. Besides, the Maui logs viewed by system administrator can reveal the detail why the job did not start.

- ❖ Workload Information

In order to manage the workload, Maui provides an extensive array of job prioritization options for each site. Maui allows sites to control exactly how jobs run through the job queue. Maui provides the *showq* command to show queued jobs, a relevant listing of both active and idle jobs.

2.3 MOSIX

Bproc aims to provide a single system image similar to Mosix but does not attempt to do Mosix style transparent process migration. Bproc will allow a process to migrate from one node to another but this migration is not transparent or automatic. On the other hand, bproc should avoid most if not all of the performance penalties associated with Mosix style migration.

MOSIX is a software module for supporting scalable cluster computing with Linux. The core of MOSIX are kernel-level, adaptive load-balancing algorithms that are designed to respond to variations in resource usage among the nodes by migrating processes from one node to another (strong mobility), preemptively and transparently. MOSIX allows a cluster of PCs to work cooperatively as if part of a single system [13]. There are two versions: a kernel patch (K-MOSIX) that can be applied to a specific Linux kernel, and a user-level package (U-MOSIX) that can be used with different Unix platforms.

Both versions are based on the same principles, and are geared to achieve even work distribution and load balancing. K-MOSIX provides load-balancing by transparent process migration. The application developer need only fork new processes. U-MOSIX provides even load distribution using several of the algorithms of K-MOSIX.

[13] claims that MOSIX can support configurations with large numbers of computers, with minimal scaling overheads to impair the performance. A low-end includes several PCs by Ethernet, while a larger may include a large number of workstations (SMP and non-SMP) and servers by higher speed LAN such as Gigabit-Ethernet.

[14] is an procedural article to guide the user for installing and configuring MOSIX. During the configuration, the terminology such as “Migration” is used in MOSIX that gives the insight of MOSIX’s process mobility to other nodes. It continues with installation MOSIX on the Development Machine and installation testing.

Another attempt to deploy MOSIX is pursued and reported by Jelmer Vernooij [15]. It tried to use Mosix with Linux Terminal Server Project (LTSP). The reason is given as such that LTSP is to run remote X (Linux) without needing disks and MOSIX has the capability to migrate processes to machines with a lower load. The procedure to install MOSIX is detailed with some practical tips, gained at the project.

[16] presents one example of how clusters of MOSIX extension Linux systems were used to eliminate a performance bottleneck and to reduce the cost of building software. It concludes that this approach is beneficial to create high performance and distributed build environments form commodity hardware and open source software.

Barak and La'adan [17] experienced the resource sharing that is geared for performance scalability in a scalable computing cluster (CC), MOSIX over fast Ethernet and the Myrinet LANs. Parallel applications can be executed by allowing MOSIX to assign and reassign the processes to the best possible nodes. It demonstrated the low-cost, scalable CC from commodity components, such as PC's, UNIX and PVM.

2.4 SPRITE

Sprite was a UNIX-like distributed operating system developed at Berkeley from 1984 [18]. Processes run on a number of different machines, and had a number of interesting features, such as load-balancing, a high-speed, aggressively-caching, distributed file-system, and a fast log-structured local file-system.

Sprite [19] provides transparent process migration to allow load sharing. It provides a UNIX like system call interface. Each process appears to run on a single host node, but physically to execute on a different machine. [20] presents the implementation of a SPRITE system to provide the transparent process migration of processes. The simulated results reside in migration, load sharing over distributed systems. [21] estimates the performance measurements, especially on a multiprocessor Sprite kernel. Variety of macro- and micro-benchmarks were taken in place for this matter.

K. Shirriff [22] implemented of memory sharing and file mapping of 4.2 BSD Unix to Sprite with user-level control over paging. In the report, he stated two limited sharing capabilities: code and heap segments while having separate stacks. Mach model is similar in this matter of execution in a single address space. In addition, some of his effort has exhibited in collecting the reference of Sprite papers [23].

Despite the kernel level dynamic load balancing in a cluster system such as Sprite, [24] suggests to provide a high-level and portable implementation of migratable Java threads over Java Virtual Machine.

2.5 Comparison of Implementations

In this section we will explore some of the similarities and differences in the design decisions among several systems that support process migration. Specifically, we will examine MOSIX, developed at the Hebrew University of Jerusalem, Israel; Condor, developed at the University of Wisconsin-Madison; and Sprite, developed at the University of California at Berkeley. Condor, MOSIX, and Sprite appear similar on the surface in that they all have implemented a process migration. Underneath, however, each of the three systems is trying to solve a different problem. The design decisions that went into the systems are necessarily different, because the designs were based on different assumptions. Each of these decisions is discussed in more detail below.

2.5.1 MOSIX

MOSIX might be best described as an attempt to create a low-cost equivalent of a scalable, SMP (multi-CPU) server. In an SMP system such as the Digital Alpha Server or SGI Challenge, multiple CPUs are tightly coupled, and the operating system can do very fine-grained load balancing across those CPUs. In an SMP, any job can be scheduled to any processor with virtually no overhead. MOSIX, similarly, attempts to implement very low-overhead process migration so that the multicomputer, taken as a whole, might be capable of fine-grained load balancing akin to an SMP. The MOSIX designers have expended a great deal of effort implementing very fast network protocols, optimizing network device drivers, and doing other analyses to push the performance of their network as far as possible.

Also in line with the SMP model, MOSIX goes to great lengths to maintain the same semantics of a centralized OS from the point of view of both processes and users. Even when a process migrates, signal semantics remain the same, IPC channels such as pipes and TCP/IP sockets can still be used, and the process still appears to be on its “home node” according to programs such as `ps`.

As a result, the MOSIX implementation typically takes the form of a “pool of processors”--a large number of CPUs dedicated to acting as migration targets for high-throughput scientific computing. Although MOSIX can be used to borrow idle cycles from unused desktop workstations, that mode of operation is not its primary focus.

2.5.2 CONDOR

In contrast, Condor's primary motivation for process migration seems to be to provide a graceful way for processes that were using idle CPU cycles on a foreign machine to be evicted from that machine when it is no longer idle. **They made many simplifying assumptions; for example, that the remotely-executing processes will be running in a vacuum, not requiring contact with other processes via IPC channels. Their migration strategy does not provide a fully transparent migration model; processes ``know'' that they are running on a foreign machine, and the home machine has no record of the process' existence.** These assumptions, while more limiting than the MOSIX model, do buy a fair amount of simplicity: **Condor's designers were able to implement its process migration without modifying the kernel.**

2.5.3 SPRITE

In motivation, Sprite seems to be a cross between Condor and MOSIX. Like MOSIX, Sprite strives for a very pure migration model--one in which the semantics of the process are almost exactly the same as if the process had been running locally. IPC channels, signal semantics, and error transparency are all important to the Sprite design. However, their migration policy is much more akin to Condor's. Similar to Condor, they seem primarily motivated by the desire to gracefully evict processes from machines which are no longer idle. When processes are first created with *exec()*, they are migrated to idle workstations if possible; later, they are migrated again only if the workstation owner returns and evicts the process. Unlike MOSIX, Sprite has no desire to dynamically re-balance the load on systems once processes have been assigned to processors.

2.5.4 Specific Design Decisions

In this section, we will explore some of the specific design decisions of the three systems in more detail.

2.5.4.1 User Space vs. Kernel Implementation

Sprite and MOSIX both involve extensive modifications to their respective kernels to support process migration. **Amazingly, Condor is a process migration scheme that is implemented in user-space. Although no source code changes are necessary, users**

do need to link their programs with Condor's process migration library. The library intercepts certain system calls in cases where it needs to record state about the system call. The library also sets up a signal handler so that it can respond to signals from daemons running on the machines, telling it to checkpoint itself and terminate.

2.5.4.2 Centralized vs. Distributed Control

Condor and Sprite both rely on a centralized controller, which limits those systems' scalability and introduces a single point of failure for the entire system. In contrast, MOSIX nodes are all autonomous, and each uses a novel probabilistic information distribution algorithm to gather information from a randomly selected (small) subset of the other nodes in the system. This makes MOSIX much more scalable, and its completely decentralized control makes it more robust in the face of failures.

2.5.4.3 File system Model

Condor does not assume that file systems available on a process' home machine are also available on the target machine when a process migrates. Instead, it forwards all file system requests back to the home machine, which fulfills the request and forwards the results back to the migration target. In contrast, Sprite has a special cluster-wide network file system; it can assume that the same file system is available on every migration target. Sprite forwards the state of open files to the target machines and file system requests are carried out locally. Similar to Sprite, MOSIX assumes that the same file system will be globally available, but MOSIX uses standard NFS.

2.5.4.4 Fully Transparent Execution Location

MOSIX and Sprite support what might be called ``full transparency'': in these systems, the process still appears to be running on the home node regardless of its actual execution location. The process itself always thinks that it's running on its home node, even if it migrated and is actually running on some other node. This has several important side effects. For example, IPC channels such TCP/IP connections, named pipes, and the like, are maintained in MOSIX and Sprite despite migration of processes on either end of the IPC channel. Data is received at the communications endpoint--i.e., a process' home node--where it is then forwarded to the node on which the process is actually running. In contrast, **Condor does not support such a strong transparency model; a Condor process that migrates appears to be running on the migration target. For this reason, migration of processes that are involved in IPC is not allowed.**

2.5.4.5 Migration Policies

As mentioned earlier, MOSIX attempts to dynamically balance the load continuously throughout the lifetime of all running processes, in an attempt to maximize the overall CPU utilization of the cluster. Sprite schedules a process to an idle processor once, when the process is born, and migrates that process back to its home node if the foreign node's owner returns. Once an eviction has occurred, Sprite does not re-migrate the evicted process to another idle processor. Condor falls somewhere in between these two. Like Sprite, **Condor assigns a process to an idle node when the jobs is created.** However, unlike Sprite, Condor attempts to find another idle node for the process every time it gets evicted. Absent of evictions, **Condor does not attempt to dynamically re-balance the load as MOSIX does.**

2.5.4.6 Check pointing Capability

The mechanics of Condor's process migration implementation are such that the complete state of the process is written to disk when a migration occurs. After the process state is written to disk, the process is terminated, the state transferred to a new machine, and the process reconstructed. This implementation has a useful side effects. For example, the process state file can be saved. Saving it has the effect of ``check pointing'' the process, so that it can be restarted from a previous point in case of a hardware failure or other abnormal termination. The frozen process can also be *queued*; who is to say that the frozen process has to be restarted immediately? The state file, with its process in stasis, can be kept indefinitely--perhaps waiting for another idle processor to become available before restarting. The MOSIX and Sprite implementations are generally memory-to-memory and preclude these interesting possibilities.

Table: Comparison of process migration schemes

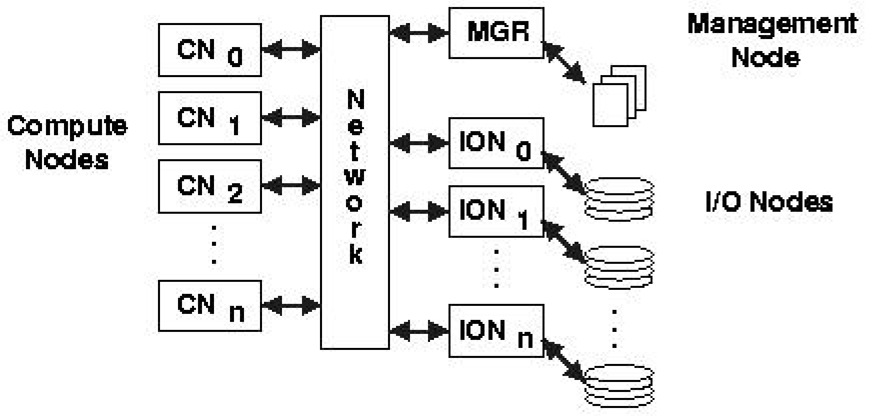
	MOSIX	CONDOR	SPRITE

Area Implementation	Kernel	User Space	Kernel
Control Algorithms	Distributed	Centralized	Centralized
Files ystem Model	State of open files transferred; all nodes have same view of file system	Requests forward to home node; results back to remote node	State of open files transferred; all nodes have same view of file system
Full Transparency (IPC, Signals, etc.)	Yes	No	Yes
Migration Policy	Continuous; Dynamic	Assign on eviction	Assign once return if evicted
Checking pointing Capability	No	Yes	No

3 Virtual File Systems

The distributed file system allows Beowulf to access inter-node file system. It makes users look like they are accessing the local file system. Basically it is capable of network transparency, location transparency and location independence. Beowulf clusters almost always use the Network File System (NFS) protocol to provide distributed file system services. However, NFS has a problem with scalability unlike the Andrew File System (AFS). The AFS is proved to be able to reduce CPU usage and network traffic. Also it overcomes the scaling problems. Recently AFS is available for Linux and has emerged in the Beowulf community. An alternative for the distributed file system is a virtual file system (VFS). One implemented open source version of VFS that operates on the Beowulf is the PVFS. It can be installed without any modifications to the hardware or kernel. The term virtual in the PVFS implies that file data is actually stored on multiple file systems on local disks, not by PVFS itself. The term parallel means that data is stored on multiple independent PCs, or cluster nodes, and that multiple clients can access this data simultaneously and transparently. PVFS maintains a consistent file name space across the machine [24].

In the user's view, the UNIX file commands such as ls, cp and rm can be used on PVFS files and directories. Also the PVFS supports the UNIX I/O interface and allows existing UNIX I/O programs to use PVFS files without recompiling. Since PVFS spreads data out across multiple cluster nodes, called *I/O nodes*, user can access data from various paths. This is to reduce bottleneck in case of one access path. Moreover, to reduce the kernel overhead, the PVFS clients directly contact PVFS servers rather than passing through the local kernel. Besides, the PVFS library can be utilized by applications or by libraries, such as the ROMIO MPI-IO library, for high speed PVFS access.



PVFS System

The figure above shows the PVFS system on the cluster. There are three types of nodes, the management node, the compute node and the I/O node.

The computer nodes are the nodes of the clients that access the PVFS files. Every node can be compute node depending on the configuration. The native API (libpvfs) in the compute node provides user-space access to the PVFS servers. This library handles the user-transparent scatter/gather operations necessary to move data between user buffers and PVFS servers.

There is only one management node in the PVFS. The management node serve as a metadata server. It has a daemon program named “mgr” running inside to manage the metadata of PVFS e.g., filename permissions, owners and its location in the directory. The client in the compute node will communicate through the library with the metadata server in the management node.

The I/O node serves as an I/O server. It has a daemon program name “iod” running inside to store and retrieve file data on its local disks. The client in the compute node will contact I/O servers in I/O nodes directly.

There are three interfaces through which PVFS may be accessed:

- PVFS native API. Not only the PVFS provides the UNIX-like interface, it also allows users to rearrange how files will be striped across the I/O nodes.
- Linux kernel interface. This allows applications to access PVFS file systems through the normal channels.

- ROMIO MPI-IO interface. This ROMIO implements the MPI2 I/O calls in a portable library. It allows parallel programmers using MPI to access PVFS files through the MPI-IO interface. [http://www.mcs.anl.gov/romio]

It is possible to share a globalize information between the processes or agents via PVFS. The PVFS libraries can be used either directly via the native PVFS calls or indirectly through the ROMIO MPI-IO interface or the MDBI interface.

4 Design Strong Mobility Environment

4.1 Mobility

The primary identifying characteristic of mobile agents is their ability to autonomously migrate from host to host. Thus, support for agent mobility is a fundamental requirement of the agent infrastructure. An agent which has “know-how” but lack of resources can request its host server to transport it to some remote destination equipped with resources. The agent server must then deactivate the agent, capture its state, and transmit it to the server at the remote host. The destination server must restore the agent state and reactivate it, thus completing the migration. The state of an agent includes all its data, as well as the execution state of its thread.

At the lowest level, this is represented by its execution context and call-stack. If this can be captured and transmitted along with the agent, the destination server can reactivate the thread at precisely the point where it requested the migration. This can be useful for transparent load-balancing, or fault-tolerant programs. An alternative is to capture execution state at a higher level, in terms of application-defined agent data. The agent code can then direct the control flow appropriately when the state is restored at the destination. However, this only captures execution state at a coarse granularity (e.g. function-level), in contrast to the instruction-level state provided by the thread context.

Agent systems execute agents using commonly available system or language environments, which do not usually provide thread-level state capture. Since mobile agents are autonomous, migration only occurs under explicit programmer control, and thus state capture at arbitrary points is usually unnecessary.

There are two models for supporting agent mobility, in the weak mobility model, on migration, the agent’s state essentially consists of the agent's program-defined data structures or set of reference to resources that can be shared among multiple agents called data state. Since there is no transferring of execution state, the execution has to start from the beginning on the destination host. If the fragment of code is transferred, it must be linked in the context of already running code in the destination host. Whereas the strong mobility model captures the agent's state at the level of the underlying thread or process, which consists of both data state and execute state. The execution can be resumed from the point it stopped on the previous host.

With weak mobility, an agent's migration is possible only at specific points in the agent's code, and typically a migration is explicitly requested in the agent's code. It is generally felt that program-controlled migration under weak mobility suffices for majority of the applications.

In agent migration, to keep the design simple and efficient, most agent programming systems do not support resumption of sessions, dealing with open files or communication channels, on migration. This avoids dependencies on remote nodes. Besides, under program-controlled mobility, one can properly close any open sessions before migration, and reopen them after migration. If an agent is multithreaded, then under the weak mobility model, the programmer needs to take special care when making explicit requests for migration in the agent's code. Problems can arise if one thread requests migration when other threads have not yet completed their tasks. In addition, one needs to prevent a situation when two threads issue migration requests to move to different hosts. Therefore, even when an agent programming system does not explicitly support a multithreaded model for its agents, the programmer must be cognizant of such implicitly created threads. Therefore, when requesting migration, it is the programmer's responsibility to ensure that all other threads have either terminated or reached a state when it is safe to terminate them and migrate the agent.

The strong mobility model allows an agent to be migrated at any point in its execution. This model is certainly useful if agents need to be moved at unpredictable points in time for fault-tolerance or load-balancing.

Another issue in agent mobility is the transfer of agent code. One possibility is for the agent to carry all its code as it migrates. This allows the agent to run on any server which can execute the code. Another possibility is not to transfer any code at all, but it requires that the agent's code must be pre-installed on the destination server. In a third approach, the agent does not carry any code but contains a reference to its code base -- a server that provides its code upon request. During the agent's execution, if it needs to use some code that is not already installed on its current server, the server can contact the code base and download the required code. This is referred to as code-on-demand (COD).

4.1.1 Weak Mobility

As mentioned before, the weak mobility makes only data state (i.e., the values of the internal variables) and code move, while the strong one allows the entire execution state (i.e., the stack and program counter) of a mobile agent, code, and data state to move . In fact, mobility requires the implementation of mechanisms to support execution stopping, state collection, data transfer and execution resuming; all these kinds of facilities must be provided by the runtime system support. The term mobility is used to indicate a change of location performed by the entities of a system. It also needs to know the capability of roaming among nodes in a network-aware fashion to find the needed resources and services. Starting from simple data, the mobility has had an evolution that has led to move the execution control, the code and the execution environment.

In weak mobility, after the movement, the agent is restarted and the values of its variables are restored, but its execution restarts from the beginning or from a given procedure (a method in case of objects). Usually, A new thread (or process) is created to execute the code. The newly created thread performs all the needed information to deliver the results to the source site. In addition, weak mobility has to explicitly synchronize in order to generate deadlocks and inconsistence state. In addition, the function of mobility need to encapsulate all the state involving a distributed computation, and can be easily traced, checkpointed, and possibly recovered locally, without any need for knowledge of global state. This function also is required in the strong mobility.

The UNIX `rshd` daemon is one example of weak mobility in that it allows the shell script to be run on a remote host.

4.1.2 Strong Mobility

In strong mobility, not only code and data state are moved, but also the execution state, in order to restart the execution exactly from the point where it was stopped before movement. In strong mobility, the mobility of a complex entity occurs with the following steps.

1. The execution flow is stopped.
2. The state of the migrating entity is collected.
3. The code and state of the migrating entity are shipped to the destination node.
4. The code and the state of the migrating entity are restored
5. The execution is restarted.

In strong mobility, not only code and data state are moved, but also the execution state, in order to restart the execution exactly from the point where it was stopped before movement. In strong mobility, the mobility of a complex entity occurs with the following steps. The execution state of a migrating agent is suspended, and its stack and program counter are sent to the destination site, together with the relevant data. At the destination site, the stack of the agent is reconstructed and the program counter is set appropriately.

Strong mobility uses state saving techniques to provide transparent process migration or persistence functionalities. Furthermore, strong mobility has the ability to store and retrieve computations as variables (continuations) and passes these to the other agents (remote continuations). To support these things, transparent location function is required. Strong mobility also usually communicates in an asynchronous fashion which one agent sends messages to other agents and do not wait for the answers. As a fault tolerance, whenever one of the communication partners of a given agent dies, the agent will not stop working correctly even if it is waiting for some action of dead partner.

The existing languages that support strong mobility are Telescript, Tycoon, Agent TCL and Emerald. In Agent TCL, an executing TCL script can move from one host to another with single jump instruction. A jump freezes the program execution context and transmits it to a different host which resumes the script execution from the instruction that follow the jump.

One of application areas used in strong mobility is load balancing. Load balancing requires that a running application be restored exactly as it was before the movement of agents because it must be transparent to the application itself. This seems to require a strong mobility mechanism, which grants that also the execution state is transferred and resumed at the destination node.

Features	Strong Mobility	Software Tools
Code	Required	BPROC / CONDOR
Data State	Required	BPROC / CONDOR
Execution State	Required	CONDOR
Transparent migration	Required	CONDOR (Limited Support)
Agent Migration (e.g., Itinerary schedule policy (Sequence / Selection / Split / Split-Join))	Implicit	BPROC / CONDOR (Limited Support)
Inter-Agent Communication and Synchronization	Required	MPI
Collective Agent Communication	Optional	MPI
Agent Monitor and Control	Optional	CONDOR (Limited Support)
Agent Fault Recovery	Required	CONDOR (Limited Support)
Agent Identification	Required	BPROC (Limited Support)
Security	Required	NONE

4.2 Analysis of Existing Softwares

In this section, we will describe the way to apply the existing softwares to the strong mobility presented in section 4.1. The existing softwares are Bproc, MPI, CONDOR, and PVFS that can run in the Beowulf system environment.

4.2.1 MPI

The goal of MPI is to write our own parallel programs using the powerful and general message-passing model of parallel computation. Therefore, MPI can be only used on the clustering systems not general heterogenous systems connected to Internet. There are several implementation version of MPI such as MPICH and LAM (http://www.cs.nd.edu/lam).

In parallel programs, there are two important questions. First, how many processes are participating in this computation? Second, Which one am I? MPI provides functions to answer these questions by providing the follows:

- ❖ MPI_Comm_size - reports the number of processes.
- ❖ MPI_Comm_rank - reports the rank, a number of between 0 and size –1, identifying the calling process.

The MPI-1 standard doest not specify how to run an MPI program. In genereal, starting an MPI program is dependent on the implementation of MPI we are using, and might require various scripts, program arguments, and/or environment variables. *mpiexec <args>* is part of MPI-2, as a recommendation, but not a requirement. However, we can write my MPI implementor.

There is no mechanism for loading code onto processors, or assigning processes to processors or creating/destroying processes in MPI. Besides, the MPI does not provide remote memory transfers, multithreading and virtual shared memory but it is designed to be thread-safe.

In MPI, a process is defined in a group and a rank (a unique integer for labeling each process in the group). Process groups can be created and destroyed. A message label is specified by a context and a corresponding tag. In a point-to-point communication, it is a communication between pairs of processes. The Message selectivity is by rank and tag. In Collective communication, it involves all processes in the scope of the communication specified by the communicator. The collective communication routines do not take message tag arguments.

The feature of MPI includes six basic functions and 125 extension functions. The point-to-point communication functions are

- MPI_SEND
- MPI_RECV

The collective communication functions are

- MPI_Bcast – sends data from one process to all others.
- MPI_Reduce - combines data from all processes and returning the result to a single process.

4.2.1.1 Strong Mobility Features

As described above, MPI is good solution for data transportation across heterogeneous systems. Therefore, MPI successfully satisfies *Inter-Agent Communication* and *Collective Agent Communication* in strong mobility features. On the other hand, we can use socket API based on TCP/IP as alternatives of MPI. The socket API is also another good candidate for agent communication method. However, the performance of socket API is much slower than that of MPI.

Strong Mobility Features	Requirement	**MPI**	**Socket API**
Inter-Agent Communication and Synchronization	Strongly Required	**Support**	**Support**
		Functions: (MPI_SEND(), MPI_RECV())	Functions: (sendto()/recvfrom(), send()/recv())
Collective Agent Communication	Optional	**Support**	**Support**
		Functions: (MPI_REDUCE(), MPI_BCAST())	Functions: (send()/recv() using multicast or broadcast address.

Limitation		Applied only clustering systems within small local networks	Applied all gobal networks based on TCP/IP protocol but performance is much slower
System Platform		Supported on Beowulf	Supported on the systems with TCP/IP

4.2.2 BPROC

BPROC introduces a distributed process ID (PID) space. This allows a node to run processes which appear in its process tree even though the processes are physically present on other nodes. The remote processes also appear to be part of the PID space of the front end node and not the node which they are running on. The node which is distributing its pid space is called the master and other nodes running processes for the master are the slaves. Each PID space has exactly one master and zero or more slaves. Each PID space corresponds to a real PID space on some machine. Therefore, each machine can be the master of only one PID space. A single machine can be a slave in more than one PID space.

4.2.2.1 PID Masquerading

The PID masquerading modifications make it possible for a process to appear as though a process exists in a different PID space. Processes are still part of the single PID space that we're used to, but the PID related syscalls (getpid, getppid, kill, fork, wait) have been modified to treat their arguments differently and to give different responses for processes that have been tagged as masqueraded. PID masquerading also introduces a user space daemon to control some of the PID related operations normally done by the kernel. Each daemon will define a new "PID space" and can create new processes in that space. Operations such as new PID allocation are handled by this daemon. (In the case of a masqueraded process forking a new masqueraded PID will be needed for the child process. This request gets sent out to the user space daemon which will forward it to the master node. The ghost process there will fork and return the child PID it gets back to the slave on the node. The child's new masqueraded PID is set to that PID.)

The PID related syscalls will only operate on other masqueraded processes that are in the same PID space (that is they're under control of the same daemon.) Other non-masqueraded processes on the local system become effectively invisible as a result. Signals (kill(2)) that cannot be delivered locally are bounced out to the user space daemon for delivery.

This is used in conjunction with ghosts to make it appear as though a piece of the master node's PID space has been moved onto the slave node. When creating remote processes, there are really 2 processes created, a ghost on the master to represent the remote process and the real process on the slave node. The (masqueraded) process on the slave gets the same PID as the ghost on the master node. The daemon controlling the masqueraded process space in forwards requests from the real process back to the master node. This way any operations the real process performs (fork, kill, wait, etc) will performed in the context of the master node's process space. Most requests will be satisfied by the ghost thread.

4.2.2.2 Starting processes

There are two basic ways to start a process in this scheme. The simpler one is the rexec (remote execute) which takes the same arguments as execve plus a node number. This inteface also has roughly the same semantics as the execve system call. This doesn't involve transfering much data, but it does require that all binaries and any libraries they require be installed on remote nodes.

The other interface which bproc provides is a "move" or "rfork" interface. This works by saving a process' memory region and recreating it on the remote node. This has the advantage that it can transport the binary and anything mmap'ed (such as the dynmaically linked libraries) to the remote node. This could allow a great reduction in the size of the software required to be installed on a node.

4.2.2.3 C Interface Library

Programs using bproc should include the bproc header file sys/bproc.h and be linked with -lbproc. This package builds both static and dynamic versions of libbproc.

4.2.2.3.1 System Information

- void bproc_init(void) : This initializes the bproc library. It reads the current machine state from /var/run/bproc. (This machine state is only available on the master node.) It also reads an initial node mapping from $HOME/.bprocnodes if it exists.
- int bproc_numnodes(void) : Returns the number of nodes in the system. This is the number of slave nodes (not including the front end). The nodes are numbered 0 though n-1.
- int bproc_nodeup(int node) : Returns true if node is up.
- int bproc_nodeaddr(int node, struct sockaddr *s, int size) : Saves the IP address of node in the structure pointed to by s. Note that bproc_init has to be called on the master node in order for this information to be available.

4.2.2.3.2 Node mapping

The library allows the user to create a node mapping that sits on top of the real node numbers. This allows the user to always see the nodes he is using as nodes 0 through n-1 regardless of the physical nodes in use. Mappings are presented as an array of integers. The number in element zero is the real node number node zero will map on to and so on. bproc_init() reads an initial node mapping.

- int bproc_set_node_map(int *map, int numnodes) : This sets the node mapping being used by libbproc. map is a pointer to an array which lists the real node numbers that node numbers 0 through numnodes should map onto.
- void bproc_clear_node_map(void) : This clears any node mapping that might be present. After this call, all node numbers will be treated as physical node numbers.

4.2.2.3.3 Creating processes on remote nodes

Bproc provides a number of mechanisms for creating processes on remote nodes. It is probably better to think of these mechanisms as moving processes from the front end to the remote node. The rexec mechanism is like doing a move then exec with lower overhead. The rfork mechanism is implemented as an ordinary fork on the front end and then a move to the remote node before the system call returns. Execmove does an exec and then move before the exec returns to the new process.

Movement to another machine on the system is voluntary and is not transparent. Once a process has been moved all its open files are lost except for STDOUT and STDERR. These two are replaced with a single socket. (Their output is combined.) There is an IO daemon what will forward between the other end of that connection and whatever the original STDOUT was connected to. No pseudo tty operations are done.

The move is completely visible to the process after it has moved except for process ID space operations. Process ID space operations include fork(),wait,kill, etc. All file operations will operate on files local to the node that the process has been moved to. Memory that was shared on the front end will no longer be shared.

Processes currently cannot move twice. The process movement API is only provided on the master node.

Bug: Any child processes that a process had before moving will no longer be visible to it after moving. SIGCHLD's will be delivered when they exit but it will be impossible to pick up their exits status with wait().

- ❖ int bproc_rexec(int node, char *cmd, char **argv, char **envp) : This call is like execve in that it replaces the current process with a new one. The new process is created on node and the local process becomes the ghost representing it. All arguments are interpreted on the remote machine. The binary and all libraries it needs must be present on the remote machine. This function returns -1 on failure and does not return on success.
- ❖ int bproc_move(int node, int flags) : This call will move the current process to the remote node number given by node. The flags argument determines the details of the memory space move. See the VMADump for details on the flags argument. Returns 0 on success, -1 on failure.
- ❖ int bproc_rfork(int node, int flags) : The semantics of this function are designed to minic fork() except that the child process created will end up on the node given by the node argument. What happens behind the scenes is the process forks a child and

that child performs a bproc_move() to move itself to the remote node. By combining these two operations in a system call, we can prevent zombies and SIGCHLD's in the case that the fork is successful but the move is not. On success, this function returns the process ID of the new child process, on failure it returns -1.

- int bproc_execmove(int node, char *cmd, char **argv, char **envp) : This function allows migration of ordinary binaries by allowing you to exec a new process and move the new process before it "wakes up". Returns -1 on failure, does not return on success.

4.2.2.3.4 VMADump: Dumping and restoring processes

VMADump is a kernel module distributed with bproc which will dump a process's state to or from a file descriptor. VMADump is short for Virtual Memory Area Dumper. It will read or write to pipes, sockets, etc. as well as ordinary files. These functions are used internally by bproc to move processes around. The saved state includes:

- All the processes memory regions. The date for all writable regions is saved. Read-only regions that are mmap'ed from files (i.e. glibc code) can be stored as file references to reduce the size of dumps.
- Other information about memory mmap'ed regions like where the bss and stacks here. This allows stacks to grow and setbrk (malloc) to work after restoring the memory space.
- The process's registers including FPU state.
- The process's signal handlers.

The following interface is provided for vmadump in libbproc:

- int bproc_vmadump(int fd, int flags) : This takes the current process and dumps it to the file fd. It returns the number of bytes written to fd. When the process is undumped, this function will return 0. The flags argument determines what memory regions will have their data dumped and which ones will be stored as file references. Writable memory regions are never stored as file references.

VMAD_DUMP_LIBS : If given, read only mmaps from files in /lib and /usr/lib will not be stored as file references.

VMAD_DUMP_EXEC : If given, read only mmaps from the executable file will not be stored as file references.

VMAD_DUMP_OTHER : If given, other read only mmaps not falling into the categories above will not be stored as file references.

VMAD_DUMP_ALL : If given, no read only mmaps will be stored as file references. This is the safest option if in doubt. This is the logical OR of the other flags.

- int bproc_vmaundump(int fd) : This attempts to undump an image from fd. This function is not very error tolerant. If something goes wrong half way through undumping, it will return with a half-undumped process. If successful, the current process is replaced with the image from the dump. (much like exec)

4.2.2.3.5 C Library Reference

The following shows the C library reference for Bproc.

bproc_access — determine whether a node can be accessed

bproc_chgrp — change node ownership

bproc_chmod — change the permissions on a node

bproc_chown — change node ownership

bproc_currnode — return the node the calling process is running on

bproc_execmove — move a newly execed process to another node

bproc_masteraddr — return the address of the master node

bproc_move — move the calling process to another node

bproc_nodeaddr — return the address of a node

bproc_nodecachepurgefail — flush file cache failure list

bproc_nodecachepurgeok — flush successful downloads from file cache

bproc_nodechroot — ask a slave daemon to perform a chroot

bproc_nodehalt — ask a slave daemon to halt the machine

bproc_nodeinfo — get status information for single node

bproc_nodelist — get status information for single node

bproc_nodenumber — get the node number corresponding to an address

bproc_nodepwroff — ask a slave daemon to power off the machine

bproc_nodereboot — ask a slave daemon to reboot the machine bproc_nodesetstatus — set status of a node

bproc_nodestatus — get the status of a move the calling process to another node

bproc_notifier — get BProc notifier file descriptor

bproc_numnodes — get the number of nodes in the system

bproc_pidnode — return the node that a process exists on

bproc_proclist — get status information for single node

bproc_requestfile — get status information for single node

bproc_rfork — fork a child onto remote node

bproc_version — get BProc version information

4.2.2.4 Strong Mobility Features

As described above, BPROC is the solution for golobal process ID and limited process scheduler and migration that move the process to the remote node at only one time. Therefore, MPI successfully satisfies *Agent Identification* and *Collective Agent Communication* in strong mobility features. On the other hand, we can use socket API based on TCP/IP as alternatives of MPI. The socket API is also another good candidate for agent communication method. However, the performance of socket API is much slower than that of MPI.

Strong Mobility Features	Requirement	**BPROC**
Agent Identification	Required	**Support**
		Functions: (bproc_proclist(), bproc_pidnode())
Agent Migration	Implicit	**Limited Support**
		Processes decide if, when and where they will migrate
Code	Required	**Support**
		Functions: (bproc_rexec(),bproc_move(),bproc_rfork(), bproc_execmove())

Data State	Required	**Support**
		Functions: (bproc_rexec(),bproc_move(),bproc_rfork(), bproc_execmove())
Limitation		- Limited Agent Location: Processes decide if, when and where they will migrate - System image is not preserved. (Limited Weak Mobility) - Open files are lost.
System Platform		Supported on Beowulf
Others		MPI and Bproc can work together

4.2.3 CONDOR

There are several limitations on CONDOR

4.2.3.1 Current Limitations

Although Condor can schedule and run any type of process, Condor does have some limitations on jobs that it can transparently checkpoint and migrate:

- ❖ Multi-process jobs are not allowed. This includes system calls such as fork(), exec(), and system().
- ❖ Interprocess communication is not allowed. This includes pipes, semaphores, and shared memory.
- ❖ Network communication must be brief. A job may make network connections using system calls such as socket(), but a network connection left open for long periods will delay checkpointing and migration.
- ❖ Sending or receiving the SIGUSR2 or SIGTSTP signals is not allowed. Condor reserves these signals for its own use. Sending or receiving all other signals is allowed.

- Alarms, timers, and sleeping are not allowed. This includes system calls such as alarm(), getitimer(), and sleep().
- Multiple kernel-level threads are not allowed. However, multiple user-level threads are allowed.
- Memory mapped files are not allowed. This includes system calls such as mmap() and munmap().
- File locks are allowed, but not retained between checkpoints.
- All files must be opened read-only or write-only. A file opened for both reading and writing will cause trouble if a job must be rolled back to an old checkpoint image. For compatibility reasons, a file opened for both reading and writing will result in a warning but not an error.
- A fair amount of disk space must be available on the submitting machine for storing a job's checkpoint images. A checkpoint image is approximately equal to the virtual memory consumed by a job while it runs. If disk space is short, a special checkpoint server can be designated for storing all the checkpoint images for a pool.
- On Digital Unix (OSF/1), HP-UX, and Linux, your job must be statically linked. Dynamic linking is allowed on all other platforms. (Note: these limitations only apply to jobs which Condor has been asked to transparently checkpoint. If job check pointing is not desired, the limitations above do not apply.)
- **Security Implications:** Condor does a significant amount of work to prevent security hazards, but loopholes are known to exist. Condor can be instructed to run user programs only as the UNIX user nobody, a user login which traditionally has very restricted access. But even with access solely as user nobody, a sufficiently malicious individual could do such things as fill up /tmp (which is world writable) and/or gain read access to world readable files. Furthermore, where the security of machines in the pool is a high concern, only machines where the UNIX user root on that machine can be trusted should be admitted into the pool. Condor provides the administrator with IP-based security mechanisms to enforce this.
- **Jobs Need to be Re-linked to get Check pointing and Remote System Calls:** Although typically no source code changes are required, Condor requires that the jobs be re-linked with the Condor libraries to take advantage of check pointing and

remote system calls. This often precludes commercial software binaries from taking advantage of these services because commercial packages rarely make their object code available. Condor's other services are still available for these commercial packages.

4.2.3.2 Condor Daemons

The following list describes all the daemons and programs that could be started under Condor and what they do:

- condor_master

This daemon is responsible for keeping all the rest of the Condor daemons running on each machine in your pool.

- condor_startd

This daemon represents a given resource (namely, a machine capable of running jobs) to the Condor pool. It advertises certain attributes about that resource that are used to match it with pending resource requests.

- condor_starter

This program is the entity that actually spawns the remote Condor job on a given machine.

- condor_schedd

This daemon represents resources requests to the Condor pool. Any machine that you wish to allow users to submit jobs from needs to have a *condor_schedd* running..

- condor_shadow

This program runs on the machine where a given request was submitted and acts as the resource manager for the request.

- condor_collector

This daemon is responsible for collecting all the information about the status of a Condor pool.

- condor_negotiator

This daemon is responsible for all the match-making within the Condor system. Periodically, the negotiator begins a negotiation cycle, where it queries the collector for the current state of all the resources in the pool.

- condor_kbdd

  This daemon is only needed on Digital Unix and IRIX. On these platforms, the *condor_startd* cannot determine console (keyboard or mouse) activity directly from the system.

- condor_ckpt_server

  This is the checkpoint server. It services requests to store and retrieve checkpoint files.

4.2.3.3 Submitting Different Types of Jobs: Alternative Universes

A Universe in condor defines an execution environment. Condor supports the following Universes:

- Vanilla
- MPI: The MPI Universe allows parallel program written with MPI to be managed by Condor
- PVM
- Globus
- Scheduler: DAGMan Scheduler
- Standard:
  - Transparent process checkpoint and restart
  - Transparent process migration
  - Remote system calls
  - Configurable file I/O buffering
  - Toconvert our program into a standard universe job, we must use condor_compile to relink it with the condor libraries.
    - Example:

```
% cc main.o –o program
% condor_compile cc main.o –0 program
```

4.2.3.4 Strong Mobility Features

Strong Mobility Features	Requirement	CONDOR
Code / Data State / Execution State	Required	**Support** Commands: (*condor_checkpoint, condor_submit)*
Transparent Migration	Required	**Limited Support** Commands: (*condor_reschedule, condor_submit)*
Agent Migration	Required	**Limited Support** Commands: (*condor_findhost)*
Agent Fault Recovery	Required	**Support** Commands: (*condor_checkpoint, condor_reconfig)*
Agent Identification	Required	**Limited Support** Commands: (*condor_q)*
Agent Monitor and Control	Optional	**Limited Support** Commands: (*condor_q, condor_status)*
Security	Required	**Support** X.509 Certificates for Authentication
Limitation		- See Section 4.2.3.1 Current Limitations - Multi-Universe Problem (e.g., need to check if “MPI” and “check point” can work together using different universe) – see Section Section 4.2.3.3 - No library interfaces for developer. Limitted support with command.
System Platform		Supported on Beowulf
Others		Need to check the real performance of CONDOR

4.2.4 MAUI

Maui is not the right tool to emulate the strong mobility because it is a batch scheduling in the user level via external commands. The MAUI itself is not a resource manager but it makes decisions by querying and controlling a resource management system such as PBS or Loadleveler.

4.2.5 PVFS

The PVFS is a good candidate tool to share the information or perform the file I/O by multiple processes which can access this file simultaneously and transparently. Also the PVFS maintains a consistent file name space across the cluster.

The PVFS libraries can be used either directly via the native PVFS calls or indirectly through the ROMIO MPI-IO interface or the MDBI interface.

4.2.5.1 Direct access via PVFS function calls

All normal UNIX I/O like read( ) or write( ) will work fine with PVFS without any changes. Files created this way will be striped according to the file system defaults set at compile time. To determine the physical distribution when the file is first created, the PVFS provides the function `pvfs_open()`with the parameters as shown below.

```
pvfs_open(char *pathname, int flag, mode_t mode);
pvfs_open(char *pathname, int flag, mode_t mode, struct pvfs_filestat *dist);
```

The pvfs_filestat structure is described below.

```
struct pvfs_filestat {
   int base;   /* The first iod node to be used */
   int pcount; /* The number of iod nodes for the file */
   int ssize;  /* stripe size */
   int soff;   /* NOT USED */
   int bsize;  /* NOT USED */
}
```

To obtain information on the physical distribution of a file, use `pvfs_ioctl()` on an open file descriptor:

```
pvfs_ioctl(int fd, GETMETA, struct pvfs_filestat *dist);
```

Besides, the PVFS provides multi-dimensional block interface (MDBI) which is a slightly higher-level view of file data than the native PVFS interface. With the MDBI, file data is considered as an N dimensional array of records. This array is divided into ``blocks" of records by specifying the dimensions of the array and the size of the blocks in each dimension.

There are five basic calls used for accessing files with MDBI:

int open_blk(char *path, int flags, int mode);
int set_blk(int fd, int *D*, int *rs*, int ne_1, int nb_1, ..., int ne_n, int nb_b);
int read_blk(int fd, char *buf, int $index_1$, ..., int $index_n$);
int write_blk(int fd, char *buf, int $index_1$, ..., int $index_n$);
int close_blk(int fd);

4.2.5.2 Indirect access via ROMIO MPI-IO

The ROMIO MPI-IO interface implements the MPI-2 I/O calls in a portable library. It allows parallel programmers using MPI to access PVFS files through the MPI-IO interface. The MPI-IP functions provide basic functions performed on (parallel) Files including File opening, File closing, File deleting, File resizing, File space-pre-allocating, File size/parameter querying and File Info setting/getting. The example of functions and their parameter are shown below. The full detail is available in the MPI-2 (I/O chapter).

- MPI_FILE_OPEN : open the file identified by the file name filename on all processes in the comm communicator group.

MPI_FILE_OPEN(comm, filename, amode, info, fh)

IN comm	communicator (handle)
IN filename	name of file to open (string)
IN amode	file access mode (integer)
IN info	info object (handle)
OUT fh	new file handle (handle)

- MPI_FILE_CLOSE : synchronizes file state then closes the file associated with fh.

MPI_FILE_CLOSE(fh)
INOUT fh file handle (handle)

- MPI_FILE_DELETE : delete the file identified by the file name filename.

  MPI_FILE_DELETE(filename, info)

IN filename	name of file to delete (string)
IN info	info object (handle)

The MPI-IO is able to implement in various styles e.g., Noncontiguous Accesses, Collective I/O, Nonblocking I/O, Split Collective I/O and Shared File Pointers.

References

[1] Scyld Computing Corporation, *"Scyld Beowulf Clustering for High Performance"*, *white paper*, available at http://www.scyld.com/products/wpaper.pdf

[2] Beowulf Software, available at http://www.beowulf.org/software/software.html

[3] MPICH (Argonne National Laboratory's implementation of MPI), available at http://www-unix.mcs.anl.gov/mpi/mpich/index.html

[4] LAM/MPI (Local Area Multicomputer MPI), available at http://www.mpi.nd.edu/lam/

[5] Liang Cheng, Ajay Wanchoo, and Ivan Marsic. Hybrid Cluster Computing with Mobile Objects. Proceedings. The Fourth International Conference/Exhibition on Volume: 2 , 2000 , Page(s): 909 -914 vol.2. Available at http://www.caip.rutgers.edu/disciple/Publications/HPCAsia2000.pdf

[6] Holger Pals, Stefan Petri, and Claus Grewe. FANTOMAS Fault Tolerance for Mobile Agents in Clusters. Available at http://ipdps.eece.unm.edu/2000/ftpds/18001241.pdf

[7] Sherif A. Elfayoumy and James H. Graham An Agent-based Architecture for Tuning Parallel and Distributed Applications Performance. 2nd International Workshop on Cluster-Based Computing (WCBC'2000), Santa Fe, NM, May 2000. Available at www.crhc.uiuc.edu/~steve/wcbc00/wcbc-00-elg.pdf

[8] Putchong Uthayopas, Sugree Phatanapherom, Thara Angskun, Somsak Sriprayoonsakul. "SCE : A Fully Integrated Software Tool for Beowulf Cluster System." Proceedings of Linux Clusters: the HPC Revolution, National Center for Supercomputing Applications (NCSA), University of Illinois, Urbana, IL, 2001 June 25-27. Available at http://prg.cpe.ku.ac.th/publications/sce_linuxhpc.pdf

[9] Yanyan Yang, Omer F. Rana, David W. Walker, Christ Georgousopoulos, Roy Williams. "Mobile Agent on the SARA Digital Library." Center for Advanced Computing Research (CACR) Technical Reports in 2000 (CACR-186). Available at http://www.cacr.caltech.edu/Publications/techpubs/

[10] Montri Sapapipatpong and Putchong Uthayopas. A Prototype Implementation of Mobile Agent System on SMILE Beowulf Cluster. Proceeding of The 22nd

Electrical Engineering Conference (EECON-22), Bangkok, Thailand, Dec 1999. Available at http://prg.cpe.ku.ac.th/publications/ and http://prg.cpe.ku.ac.th/~pu/.

[11] MAUI, available at http://www.supercluster.org/documentation/

[12] Thomas Sterling, Beowulf Cluster Computing with Linux, MIT Press, 2002. MOSIX. Available at http://www.mosix.org

[13] J. Drake. Linux Clusters without the Pain. Available at http://softwaredev.earthweb.com/sdopen/article/0,,12077_630211,00.html

[14] J. Vernooij. Using Mosix with LTSP. Available at http://people.nl.linux.org/~jelmer/ltsp-mosix.html

[15] S. McClure and R. Wheeler. HOW LINUX CLUSTERS SOLVE REAL WORLD PROBLEMS. *Proceedings of FREENIX Track: 2000 USENIX Annual Technical Conference*. San Diego, CA, 2000. Available at http://www.usenix.org/events/usenix2000/freenix/full_papers/mcclure/mcclure.pdf

[16] A. Barak and O. La'adan. The MOSIX Multicomputer Operating System for High Performance Cluster Computing. *Journal of Future Generation Computer Systems*, Vol. 13, No. 4-5, pp. 361-372, March 1998. Available at http://citeseer.nj.nec.com/barak98mosix.html

[17] Sprite (University of California, Berkeley). Available at http://www.cs.berkeley.edu/projects/sprite/sprite.papers.html

[18] F. Douglis and J.K. Outerhout. Process Migration in the Sprite Operating System. In *Proceedings of the 7th International Conference on Distributed Computing Systems*, pp. 18-25, September 1987.

[19] F. Douglis and J.K. Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation, Software-Practice & Experience, 21(8):757--785, August 1991. Available at http://citeseer.nj.nec.com/douglis91transparent.html

[20] J.H. Hartman and J.K. Ousterhout. Performance Measurements of a multiprocessor Sprite Kernel. Available at http://citeseer.nj.nec.com/hartman90performance.html

[21] K. Shirriff. An Implementation of Memory Sharing and File Mapping.

[22] K. Shirriff. Sprite papers. Available at http://www.cs.berkeley.edu/projects/sprite/sprite.papers.html

[23] M.J.M. Ma, C.L. Wang and F.C.M. Lau. Delta Execution: A preemptive Java thread migration mechanism. Available at http://www.csis.hku.hk/~fcmlau/papers/cluster00.pdf

[24] http://parlweb.parl.clemson.edu/pvfs