LoadLeveler: Workload Management for Parallel and Distributed Computing Environments

Anton A. Prenneis, Jr.

IBM POWERparallel Development, Poughkeepsie, N.Y., U.S.A.
email: prenneis@vnet.ibm.com

Abstract

LoadLeveler is a job scheduling system that allows users to run serial and parallel batch jobs alongside interactive work in a homogeneous cluster of workstations or a heterogeneous cluster that may or may not include a RISC System/6000 Scalable POWERparallel (SP) System. LoadLeveler is currently supported on the RISC System/6000 platform running AIX as well as on the HP-UX, SGI IRIX, SunOS and Solaris platforms. It is a powerful and flexible system in use worldwide by educational, industrial and government research centers and product development laboratories. Among the disciplines in which LoadLeveler is used to run large scale simulations and to optimize site resource usage are the aerospace, automotive, finance, petroleum, pharmaceutical, retail and semiconductor industries, and research labs running experiments in astronomy, biology, computer science, economics, environmental and weather studies, high-energy particle physics and physical chemistry, among others. This paper will provide a general description of the features and functionality of the latest release of LoadLeveler (1.3 at the time of this writing).

1 Overview of LoadLeveler

1.1 Administration

Administering a LoadLeveler cluster is very easy. Policies governing machines, users and jobs are maintained by the administrator in a central administration file and a central configuration file. All administrative tasks can be accomplished and all cluster statistics can be obtained via commands that can be issued from the command line or from a GUI monitor on any workstation sharing the network with the cluster.

One of the most attractive features of LoadLeveler is this ease of management. One person can work part time managing a number of clusters spanning hundreds of machines. Administrative responsibilities can be delegated among users, leaving day to day tasks, such as job class management, to the users most directly affected. This frees senior administrators to work on such broader issues as cluster architecture, user account management, and performance.

It is the primary responsibility of the LoadLeveler administrator to ensure that a site's resources, namely cpu cycles, are optimized at all hours of each day, and that all users have fair access to those resources. This implies an ongoing responsibility on the part of the administrator, calling for constant planning and reevaluation of cluster configurations as computing power is added to and taken away from a site, and as the processing needs of users become more sophisticated. This is where the true challenge of LoadLeveler administration lies. And this is where the flexibility of the LoadLeveler system shines as a vital asset.

1.2 Machines

LoadLeveler runs on a cluster of client machines which report to a *central manager* machine via udp sockets at a globally defined time interval (this time interval, like other global time intervals in LoadLeveler, is easily configured by the administrator). The information reported to the central manager by each client includes the types of jobs the client is configured to run, the number and types of jobs currently running on the client, as well as the client's load average, available paging space, keyboard idle time, and other resource information. The central manager will use the information reported by each client in concert with administrator defined client properties, such as machine priority, to select an optimal client on which to run a job submitted by a user.

To prevent the central manager from becoming a central point of failure for a LoadLeveler cluster, the administrator can define one or more *alternate central managers* to take over for the primary if the primary does not respond to client connections for greater than a globally defined time interval. Since all job and machine state information is persistently stored on client machines, an alternate central manager is guaranteed to take over the cluster with no loss of information as soon as all the clients have established contact with it.

There are three types of LoadLeveler clients - *submit-only client, scheduling client, executing client* - and any machine in a cluster, including the central manager, can play multiple client roles.

- A submit-only client functions as a "dumb" LoadLeveler terminal. Its only function is to allow users to submit jobs to the cluster and to query the state of clients and jobs in the cluster. This type of client does not run any jobs, nor does it handle the scheduling of jobs. No LoadLeveler daemons run on a submit-only machine. No machine acting as a submit-only client can play any other client role.
- A scheduling client (also referred to as a *scheduler*) maintains a local database of jobs that have been submitted by users to run somewhere in the cluster. These jobs may have been submitted by someone logged into the scheduling client machine or, if this is a *public scheduler*, some jobs may have been submitted from a submit-only machine which were then forwarded to this scheduler. The scheduler will send its job information to the central manager which will then queue the jobs until they are ready to run. When the central manager finds an executing client to run a job on, it notifies the scheduler which establishes a tcp socket connection with the executing client, sends the job's executable to the executing client, and maintains contact with the executing client for the duration of the job. During a job's life, the scheduler will send periodic job status to the central manager, gather accounting data, and maintain a record of execution checkpoints. The scheduler runs as the **LoadL_schedd** daemon. Any machine acting as a scheduler can also function as an executing client.
- An executing client runs LoadLeveler jobs. It receives the job's executable from the scheduler and runs the job locally with the permissions (and AFS authentication if applicable) of the user who submitted the job. The executing client runs as the **LoadL_startd** daemon, which spawns a **LoadL_starter** process to manage each of the jobs it runs. All resource limits imposed on a job by LoadLeveler, such as cpu time, are monitored and enforced by the LoadL_startd daemon. If a job exceeds a resource limit, the LoadL_startd will remove it from the executing machine and write the reason the job was removed to stderr, optionally sending mail notification to the user who submitted the job. An executing machine can be configured to be *dedicated* or *interactive* (Note: This interactive designation is not related to Interactive Session Support, which is discussed in section 4). A dedicated machine will accept all batch jobs dispatched to it by the central manager and run them to completion with no interruption. An interactive machine will give priority to the interactive user by suspending, vacating or not accepting batch jobs under certain conditions. Such conditions could include keyboard/mouse activity, load average thresholds, times of day, and others. Any machine acting as an executing client can function as a scheduling client as well.

Whether a machine will be a submit-only, scheduling or executing client is determined in its *local configuration* file. This file is located in the machine's *local LoadLeveler file system* which could exist in the machine's journaled file system (JFS) or in an NFS mounted file system. The settings in this file override defaults set by the administrator in the global configuration file, allowing for machine customization. The local LoadLeveler file system is owned by the *loadl* userid, or some other *primary LoadLeveler userid*. This ensures that only the LoadLeveler administrator, who is the owner of this userid, will be able to modify the local configuration file. Of course, anyone with superuser authority on a machine can edit the file if it is located in the client's JFS.

Scheduling and executing clients can be in any of a number of different *daemon states* at a given point in time.

Daemon states applying to the LoadL_schedd are:

- *Available* - This state indicates that the machine is available to schedule jobs.
- *Draining/Drained* - This state indicates that the LoadL_schedd is not available to schedule new jobs but that it will continue to schedule any jobs that have already started running on an executing machine. The LoadL_schedd will remain in this state until *resumed*.
- *Down* - This state indicates that the central manager has not heard from the LoadL_schedd within an administrator defined time interval. The cause of this state could be a network failure between the scheduling client and the central manager, the scheduling client machine being down or in a hanging state, or the LoadL_schedd actually being down. The latter will always be the case for a submit-only client.

Daemon states applying to the LoadL_startd are:

- *Idle* - This state indicates that the LoadL_startd is not running any jobs and is available to do so.
- *Running* - This state indicates that the LoadL_startd is running one or more jobs and is available to run at least one more.
- *Busy* - This state indicates that the LoadL_startd is running the maximum number of jobs it is configured to be able to run at one time.
- *Draining/Drained* - This state indicates that the LoadL_startd is not available to run new jobs but that it will run to completion any jobs that have already started. The LoadL_startd will remain in this stated until resumed.
- *Draining/Drained Classes* - This state is the same as Draining/Drained, but only applies to the job classes listed. Drained job classes will remain that way until the LoadL_startd is told to resume them.
- *Flushed* - This state indicates that the administrator has immediately emptied the executing client of all running jobs, removing them whether complete or not, and vacating the *restartable* ones to the central job queue to be run again. The LoadL_startd will remain in this state until resumed.
- *Down* - This is identical to the LoadL_schedd down state.

In addition to the LoadL_schedd and LoadL_startd daemons, the other daemons that make up LoadLeveler are the **LoadL_negotiator** daemon, which runs only on the central manager and which is responsible for maintaining the central job queue and the central list of client machines in the cluster; data which it uses to inform its scheduling decisions, the **LoadL_kbdd** daemon, which monitors keyboard and mouse activity on AIX clients, and the **LoadL_master** daemon, which runs as the parent LoadLeveler daemon on each client in the cluster.

1.3 Jobs

A LoadLeveler job is a group of one or more executables defined with characteristics that tell LoadLeveler about the environment in which the executables need to run. These environments can include workstation clusters, dedicated workstations, SP machines or even NQS machines existing outside the LoadLeveler cluster.

Central to LoadLeveler's job scheduling scheme is the concept of *job class*. Job classes allow the administrator to create job categories based on resource consumption (such as memory, cpu time and wallclock time), as well as job characteristics like queue priority and nice value. The administrator can restrict the use of a job class to a list of users and can appoint class administrators who are allowed to issue administrative commands for jobs of a given class.

Job classes ultimately allow the administrator to configure individual client machines to be able to run a specific mix of job types. For example, an administrator could configure a powerful workstation to be able to run up to three memory intensive jobs at a time while configuring a less powerful workstation to be able to run two low memory jobs at a time.

To submit a job, a user enters job information into a *job command file*. Included in the job command file is the location of one or more executables to run, and any *job requirements* such as arguments to be passed to the executables at run time, shell environment variables, job class, required resource limits beyond those defined for the job class, required machine architecture and operating system level, what kind of execution checkpointing, if any, will be used, and whether or not the job should be vacated to the central job queue if it is removed from an executing machine prior to completion.

The format of a job command file is that of a shell script with all LoadLeveler directives having the syntax of shell script comments. This gives the user the option of submitting a shell script to LoadLeveler as part of a command file with no executable specified. When this type of job is dispatched to run, the command file itself is sent from the scheduler to the executing client to be run as the job executable.

A job command file can contain the location of a *prologue* program to be run on the executing machine prior to the job's execution, and an *epilogue* to be run directly following execution. A prologue can be used for setup tasks such as acquiring temporary disk space and obtaining a license, and an epilogue can be used for clean up tasks such as deleting data and scratch files.

Additional mechanisms exist within a command file to allow users to select machines based on custom features such as the availability of node-locked licenses, special run-time environments, machine model numbers, available amount of virtual memory, relative machine speeds, etc. If any of the above mentioned criteria are omitted from a job command file, LoadLeveler will queue the job with default requirement values where needed.

A job command file can contain a *single-step job* or a *multi-step job*. Up until now, no distinction has been made between a job and a job step. A job step is a subset of a job and contains one executable and the various requirements which apply to it. A single-step job runs one job step once. A multi-step job runs multiple steps on one or more executing clients. Multi-step jobs can have *dependencies* defined so that certain job steps can be made to run only if a preceding step completes with a particular outcome. This gives users the ability to serialize job steps. For instance, a job could be defined with two steps: step1 runs a desired executable and step2 is defined with a dependency telling it to only run if the executable in step1 completes with an error code. The executable in step2 could then send mail alerting the submitting user of the first step's failure. A parallel job is a type of multi-step job.

At various stages throughout its life, a job step will be in one of the following *job states* that are of interest to the user (there are a few internal job states that are not mentioned here):

- **o** *Not Queued* - The job step was submitted, but it is currently being ignored by the LoadL_negotiator. This is because the user who submitted the job has reached the limit of either running or queued jobs allowed for that user by the administrator. Once the number of jobs submitted by that user decreases below the user's limit, the job will enter the Idle state.
- **o** *Idle* - The job step was submitted and is in the queue, waiting for the LoadL_negotiator to dispatch it to an executing client.
- **o** *Deferred* - This state applies to a parallel job that the LoadL_negotiator was unable to schedule enough nodes for. Such a job will reenter the queue one or more times until it either gathers enough nodes to run or until it surpasses the maximum number of times it is allowed to enter the deferred state, at which point it will be removed from the job queue.
- **o** *Hold* - The job step is either in user or system (administrator) hold in the queue. It will be ignored by the LoadL_negotiator until it is manually released from hold or until a release date and time specified in its job command file elapses.
- **o** *Starting* - The job step has been dispatched to an executing client and it is being set up to run. Set up includes the forking of a LoadL_starter parent by the LoadL_startd and the copying of the executable file over a tcp socket from the LoadL_schedd that is scheduling the step. In the case of a parallel job, this state will persist until the job step running the master task reports back to the scheduler that set up has completed for all its slave tasks and that they have begun to run.
- **o** *Pending* - The job step is in a suspended state on its executing host. One possible cause of suspension is keyboard/mouse activity by an interactive user on the executing client machine.
- **o** *Running* - The job step is running on its executing host.
- **o** *Removed* - The job step was cancelled while it was in the job queue or running on an executing client. In the latter case, it may or may not be vacated back to the job queue, depending on whether the job was marked restartable by the user.
- **o** *Vacating* - The job step is being vacated from its executing host back to the central job queue. Interactive executing clients are often configured to vacate a job if it has been suspended due to keyboard/mouse activity over a certain period of time.
- **o** *Completed* - The job step has run to completion. A job is only complete once all its steps are.

1.4 Users

In the same way the administrator is able to manage groups of machines and groups of jobs, the administrator can also manage groups of *users*. Users can be categorized and identified according to their UNIX group, or according to an administrator created LoadLeveler group. The administrator can assign a number of properties to any user, either on an individual basis or as part of a user group.

User properties include the maximum number of job steps a user can have running or queued at any given time, the maximum number of processors a user can request for a parallel job, which job classes a user is and is not allowed to use, and a user's default job class. Like jobs and machines, users can be given queue priorities by the administrator. These priorities are figured into the overall queue priority of the job steps submitted by each user. User priorities can be dynamic in the sense that a user's queue priority can be configured to decrease as the number of job steps the user has running or queued increases.

As mentioned previously, a user can serve as a *job class administrator*. In this capacity, the user has the ability to issue a subset of administrative commands to control jobs of a particular class. A user with class administrator authority can cancel, reprioritize, hold and query detailed information about jobs of that class, even if those jobs were submitted by a different user. This has the benefit of decentralizing job administration by allowing a user to have “ownership” of a job class while still denying the user full privileges of a LoadLeveler administrator.

1.5 Accounting

As a job runs, the executing client sends information to the scheduling client that the LoadL_schedd will store in its history file once the job completes or is removed. This information can be gathered by the administrator at any time either locally from a scheduling client or remotely from any machine in the cluster. An accounting summary command is provided that can be used to generate reports. The level of detail of these reports can vary widely to suit the administrator’s needs, and the information reported can be collated in a great variety of ways.

Reports can be generated on jobs submitted by a particular user, from a particular host, during a specified date range, for a particular job class, by a user group or user account, or any combination of these and many other criteria. An *accounting API* also exists that provides all accounting data in raw form, allowing the administrator to write custom accounting summary programs.

The information collected is thorough, ranging in granularity from job step start time to the number of page swaps performed by a step’s LoadL_starter parent. Ranging in between is information about the user who submitted a job, an account number if the administrator has set up chargeback accounts, which client the job was submitted from and which ones its steps ran on, the amount of system resources consumed both by all the steps and by their LoadL_starter parents, the name of the executables and arguments passed, whether the job was serial or parallel, and much more.

1.6 Clusters

As defined above, a cluster is a collection of client machines which all report to the same central manager machine. LoadLeveler gives users the ability to submit jobs to different clusters from the same machine, regardless of what cluster, if any, that machine happens to be in. This is accomplished with the use of *cluster files*. By defining the path to the appropriate cluster file as an environment variable, a user can submit, query and cancel jobs as well as query client information in more than one LoadLeveler cluster.

One benefit provided by this cross-cluster function is the ability of a user to query the workload of various clusters and submit jobs to the cluster with the largest number of available machines. Another benefit is the ease of submitting different types of jobs to the appropriate cluster; for instance serial jobs may be submitted to a heterogeneous cluster of workstations and parallel jobs may be submitted to a cluster containing an SP machine - all from the same workstation.

The only case where a machine can be prevented from communicating with a cluster is when the cluster runs with *machine authentication*. With this option the administrator has the ability to maintain a machine list in the central administration file, telling the central manager to refuse to talk to any machine not on that list.

1.7 Checkpointing

When a non-checkpointed job step is vacated (removed from an executing client and placed back in the central job queue), it will start to execute again from the beginning once it is dispatched to run on a new executing client, no matter how far execution progressed on the original executing client. This situation is particularly unfavorable after a job step has run a number of hours prior to being vacated.

Checkpointing will allow a job to pick up from its last execution checkpoint when it begins to run on its new executing host, so that the only execution time lost is the time that passed between the job's last checkpoint and the time at which it was vacated.

Users have the option of selecting *system initiated checkpointing* for their jobs, where a checkpoint is automatically generated by the system at an interval which decreases in frequency after each checkpoint is taken, or *user initiated checkpointing*. User initiated checkpoints occur each time a call is made to LoadLeveler's *checkpointing API* from within the user's application. This mechanism enables users to force their jobs to take checkpoints after critical regions of execution.

In order to enable an executable for checkpointing, it must be linked with modified C++, C or FORTRAN libraries provided with LoadLeveler.

1.8 GUI

LoadLeveler provides a *graphical user interface* consisting of three paned windows for the monitoring of both the job queue and the list of client machines running in the cluster, for the building and submission of jobs, and for the issuance of user and administrative commands.

The top window provides a view of the job queue. This display can be customized to list only the jobs submitted by a particular user, only jobs in a particular state, all jobs in the queue and other combinations.

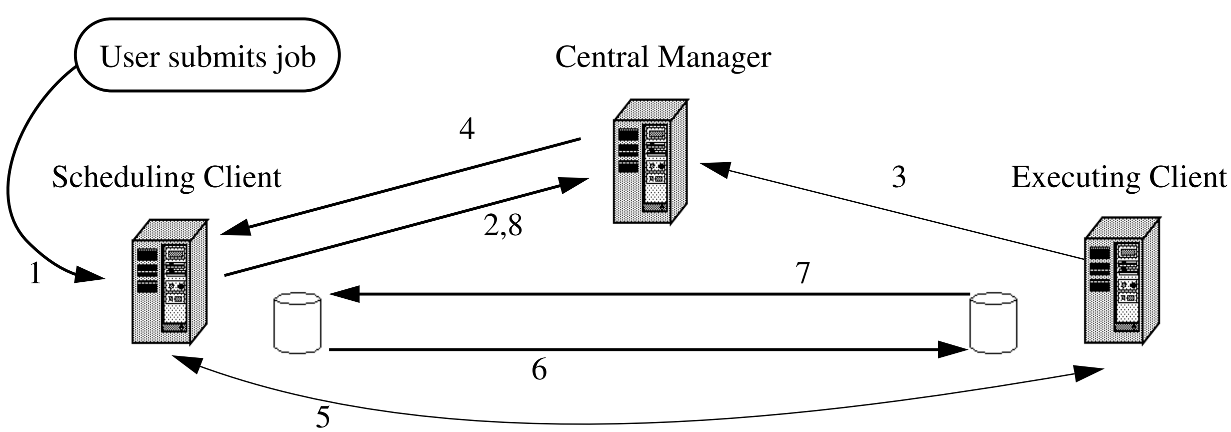
The middle window contains a view of the central manager and all the LoadLeveler clients reporting to it. As with the job window, this display can also be customized to list a subset of all the clients in the cluster.

By highlighting one or more entries in the job or machine windows and clicking on menu bar options, the user and administrator can view more detailed job and machine data and can issue any LoadLeveler command that is provided on the UNIX command line. The job and machine listings in can be ordered with primary and secondary sorts.

The bottom window displays output and error messages resulting from LoadLeveler commands issued via the GUI and it also alerts the user when contact cannot be made with the central manager.

Additional windows are available to provide templates for users when they build LoadLeveler job command files from the GUI. There are also windows to display help text.

2 Serial Job Scheduling



The basic steps for serial job scheduling are as follows. For simplicity, we'll assume that the job submitted here is a single-step job. The numbers in the list below correspond to the numbered arrows in the above diagram.

1. The user submits a job and it is queued locally on a scheduling client machine by its LoadL_schedd daemon.
2. The LoadL_schedd on the scheduling client sends the job's requirements to the LoadL_negotiator daemon on the central manager. The LoadL_negotiator, in turn, queues the job on the central job queue.
3. LoadL_startd daemons on executing clients continuously report data to the central manager. Eventually, a client reports state and configuration information that is compatible with the user's job.
4. Once the central manager finds an executing client to run the job on, it sends the scheduling client the name of the executing client.
5. The LoadL_startd on the executing client spawns a LoadL_starter process to manage the job and the LoadL_schedd on the scheduler makes a tcp socket connection with the LoadL_starter over which files and status information will be passed.
6. The scheduler machine sends the job executable (located either in its JFS or in an NFS mounted directory) to the executing client, which stores it on disk and executes it.
7. The LoadL_starter reports job status to both the LoadL_startd and the LoadL_schedd. Both daemons store job state information locally to disk. If applicable, the LoadL_starter sends checkpointing data back to the LoadL_schedd. The scheduler collects accounting information and stores it to a local history file. For checkpointing jobs, the scheduler uses the checkpointing data sent from the executing client at each checkpoint to build a new executable. If such a job is vacated from the executing client, the most recently built executable will be sent to the next executing client that the job is redispatched to.
8. The scheduler periodically updates the central manager with job status, ultimately reporting its completion or removal.

3 Parallel Job Scheduling

LoadLeveler schedules parallel jobs written for any parallel environment that uses LoadLeveler's *parallel API* package. Parallel operating environments that are already instrumented to make use of these APIs include Parallel Virtual Machine (PVM) 3.3, IBM's AIX Parallel Operating Environment (POE) 1.2.1, and IBM's Parallel Virtual Machine Extended (PVMe) 1.3.1. LoadLeveler will schedule parallel jobs to run on either the SP or on a heterogeneous cluster of workstations.

The APIs comprising the parallel API package are used by a machine designated as a parallel *master starter* (which runs the job's master task) to obtain a list of parallel nodes from LoadLeveler and to start slave tasks on the allocated nodes. One of the benefits of using LoadLeveler's parallel APIs on the SP is that it gives the master starter the ability to allocate parallel nodes through the SP *resource manager*. The resource manager can, in turn, allow jobs to use the SP's high performance switch in *user mode*. User mode provides the fastest communication possible between nodes running parallel tasks on an SP.

The LoadL_negotiator uses a *temporary reservation* scheme to select nodes for a parallel job. This means that the LoadL_negotiator evaluates executing clients as they report their status, marking any that match the parallel job's requirements as reserved. As soon as enough nodes have been reserved, LoadLeveler will start the parallel job. If enough nodes do not become available in a configurable period of time, the nodes are released from temporary reserve and the parallel job is deferred. A deferred job will eventually reenter the queue at which point the LoadL_negotiator will renew the job's temporary reservation process.

For users who want to use their own parallel scheduling algorithm, LoadLeveler provides a *scheduling API*. This API library provides calls that allow an external scheduler program to query LoadLeveler about the status of nodes and jobs. The scheduler program can then use that data in its own scheduling algorithm to decide which job to run next and on which nodes, eventually making a call to LoadLeveler telling it to start the job on the nodes it selected. An additional API call allows the external scheduler to cancel jobs.

The convenience of this scheme is that it relies on LoadLeveler to do all the work of collecting machine and job status while providing the hooks for an external scheduler to use that information to make its scheduling decisions. This API is being used with great success by the Extensible Argonne Scheduling sYstem (EASY) on the Cornell Theory Center's 512-way SP system.

4 Interactive Session Support (ISS)

ISS is a utility packaged with LoadLeveler that can be run on a cluster of machines whether or not they are part of an active LoadLeveler cluster. The function of ISS is to balance the interactive load of a cluster (called an ISS *pool*) by making the pool's least loaded node available for the next interactive login. ISS accomplishes this by altering the nameserver's mapping of the pool's domain name (also called the pool's *alias*) to the IP address of the least loaded node. For instance, if an existing pool is called *iss2*, a user would connect to it as if it were an actual host name (e.g. "rlogin iss2"). At the time of the user's connection attempt, the nameserver would resolve iss2 to the IP address of the node selected most recently by ISS as the least loaded node in the iss2 pool.

ISS makes its balancing decisions at a set time interval using one of three *metrics*: the *LoadLeveler* metric, the *round-robin* metric, or the *custom* metric. When the LoadLeveler metric is used, ISS submits a "dummy" job to LoadLeveler and lets the central manager choose the least loaded node for it. The node selected by LoadLeveler becomes the next available ISS node. The round-robin metric is trivial; ISS makes the next sequential node in the pool's host list available after a user logs into the currently available node. The custom metric allows the administrator to define the method of node selection. Two examples of a custom metric would be running "ps -ef | wc -l" at every time interval to identify the node with the fewest number of running processes, or running the "rup" command to identify the node with the lowest load average. The chosen custom metric string will be run on each node in the pool's machine list by a program called *issagent*, which will notify the ISS server whenever a node's load changes.

The ISS server, called *issmonitor*, runs on a nameserver machine. An issmonitor can manage any number of pools using any variety of metrics. Similar to the LoadLeveler alternate central manager, one or more *standby issmonitor* servers can be defined. They share the restriction of having to run on a nameserver machine. If configured to do so, the issmonitor will issue an alarm to the administrator if it ever loses contact will all the client machines in a pool.

5 Conclusion

This paper provides a very high level view of the LoadLeveler product. Hopefully it demonstrates why LoadLeveler is one of the leading commercial workload management products available today. Development continues apace, with work currently underway on serial and parallel scheduling enhancements, checkpointing improvements, performance and scalability optimization, expanded ISS functionality, ease of use, ease of migration and national language support.

6 References

[1] Using and Administering LoadLeveler: IBM Document Number SH26-7220-04: Available via anonymous ftp from lscftp.kgn.ibm.com, /pub/pps/loadleveler/lluadmin.ps.Z

[2] CTC's LoadLeveler Web Page: http://ibm.tc.cornell.edu/ibm/pps/loadl/index.html

[3] CTC's LoadLeveler-EASY Web Page: http://www.tc.cornell.edu/~jap/easy_ll.html

[4] LoadLeveler and Batch Newsgroup: http://spud-web.tc.cornell.edu/HyperNews/get/LL.html

[5] SP Discussion Group: http://spud-web.tc.cornell.edu/HyperNews/get/SPUserGroup.html