PeerMon: A Peer-to-Peer Network Monitoring System

Tia Newhall, Jānis Lībeks, Ross Greenwood, Jeff Knerr
Swarthmore College Computer Science Department, Swarthmore, PA, USA

Abstract

We present PeerMon, a peer-to-peer resource monitoring system for general purpose Unix local area network (LAN) systems. PeerMon is designed to monitor system resources on a single LAN, but it also could be deployed on several LANs where some inter-LAN resource sharing is supported. Its peer-to-peer design makes PeerMon a scalable and fault tolerant monitoring system for efficiently collecting system-wide resource usage information. Experiments evaluating PeerMon's performance show that it adds little additional overhead to the system and that it scales well to large-sized LANs. PeerMon was initially designed to be used by system services that provide load balancing and job placement, however, it can be easily extended to provide monitoring data for other system-wide services. We present three tools (smarterSSH, autoMPIgen, and a dynamic DNS binding system) that use PeerMon data to pick "good" nodes for job or process placement in a LAN. Tools using PeerMon data for job placement can greatly improve the performance of applications running on general purpose LANs. We present results showing application speed-ups of up to 4.6 using our tools.

1 Introduction

General purpose LANs of workstations are systems where multiple machines (nodes) are connected by a network. Each machine runs a stand-alone operating system (OS) and typically runs a network file system and may support a few other types of networked resource sharing. These types of systems are common at universities and other organizations where machines in offices and labs are connected to allow some system-wide resource sharing, but where most of a machine's resources are under the control of its local OS. Typically, these systems do not implement any kind of centralized scheduling of networked resources; resource scheduling is done locally by the OS running on the individual nodes.

In general purpose LANs multiple users can log into individual nodes and use the networked resources to run any workload including batch, interactive, sequential and parallel applications. The workload in such systems is much more dynamic and not as well controlled as in cluster systems that typically run system-wide job scheduling software that users must use. As a result, there are often large variations in system-wide resource usage and large imbalances in the use of computational resources in general purpose LANs [3].

To perform computational tasks efficiently it is often key to have some knowledge of resource availability and resource load. For example, it would be ideal to choose the node with the lowest CPU load, the largest amount of free RAM, and the fewest number of users to run a computationally intensive sequential program. For parallel applications (such as MPI) running on a network of workstations, performance is usually determined by the slowest node. If a user had a tool that could easily identify the best nodes on which to run a parallel job, avoiding heavily loaded nodes, the result could be a dramatic improvement in execution time of the application.

Because general purpose networked systems do not provide system-wide resource scheduling, it is up to users to either guess at good placement or gather current usage information on their own to make better informed job placement options. In some cases, this can require a fair amount of effort; in others, it may not be possible. For example, a system may be set up so that individual nodes cannot be specified for remote ssh. Instead, the DNS server may use a round-robin mapping of a generic name like `lab.cs.swarthmore.edu` to one of the nodes in the system. In this case, a user can end up on a heavily loaded node, her only recourse being to log out and hope for better placement when she tries again.

A network resource monitoring system that efficiently provides system-wide usage data could be used to better distribute users and program workloads across the system. This would result in more balanced resource usage across the system, better system-wide resource utilization and, thus, better average system-wide performance.

We designed PeerMon to efficiently provide system-wide resource usage information to tools that implement load balancing functions in general purpose LAN systems. Each node in the system runs a PeerMon daemon peer that periodically collects system usage statistics about its own node and sends its information about system-wide resource usage to a fixed number of peers (currently three). The peers are chosen based on heuristics designed to maintain accurate system-wide data and a high degree of P2P network connectivity while at the same time minimizing network overheads.

PeerMon's peer-to-peer design solves problems associated with more centralized client-server monitoring systems like those based on Simple Network Management Protocol (SNMP), namely the single server bottleneck and single point of failure. Because there is no central authority for system-wide resource information, there is no central server that can become a bottleneck as systems grow to larger numbers of nodes. Applications that use PeerMon data access it locally on the nodes on which they run by interacting with their local PeerMon daemon. This ensures that system-wide resource usage data are always available and can be accessed quickly through a local service on each node. Additionally, since it is not necessary that system-wide resource usage data be consistent across all peers for the data to be useful, our system is designed to avoid costly peer data synchronization and peer data recovery.

PeerMon is also fault tolerant. Each PeerMon peer is equal and provides system-wide usage information to clients on its local node. If a node fails, PeerMon daemons on other nodes just stop receiving data about the failed node, but continue to provide system-wide resource information for non-failed resources.

To demonstrate how PeerMon resource monitoring data can be used, we implemented three tools that make use of its data. The first tool, smarterSSH, uses data collected from the peer monitor process to select the best machine to ssh into. Currently, we support selecting the "best" machines based on combinations of CPU load, RAM load, and number of CPU cores. The second tool, autoMPIgen, uses PeerMon data to automatically generate MPI host files based on system-wide resource capabilities and usage. The third tool is dynamic DNS binding based on system-wide resource usage. Using data provided by the PeerMon daemon running on the DNS server, our tool sets bindings so that a single name is mapped to the current set of "best" nodes in the system. A user who remotely ssh's into `cslab.cs.swarthmore.edu` will be logged into one of the "best" machines in our LAN. The result is that we better distribute remote logins across machines in our system.

Currently PeerMon runs on the Swarthmore Computer Science Department's LAN of about 60 Linux 2.6/x86 machines. All three tools that make use of PeerMon data are available to the users of our system.

The remaining parts of the paper are organized as follows: Section 2 discusses related work; Section 3 discusses the design of PeerMon; Section 4 discusses PeerMon's current implementation, configuration, and running details; Section 5 discusses our three example tools that we designed that make use of PeerMon data; Section 6 presents performance results of PeerMon and our example tools; and Section 7 concludes and discusses future directions for our work.

2 Related Work

Our work is most closely related to other work in network management and network resource scheduling. There has been a a lot of work on network management systems that are designed to obtain usage information and manage networked resources. Most of these are centralized systems based on the Simple Network Management Protocol (SNMP) framework [9]. The framework is based on a client-server model in which a single central server periodically sends requests to clients running on each node to send back information about the node. In addition, SNMP allows the manager to send action requests to clients to initiate management operations on individual nodes. The centralized design allows for a single central authority to easily make system-wide resource management decisions; however, it also represents a single point of failure in the system and a bottleneck to scaling to large-sized networks.

To address the fault tolerance and scalability problems associated with a centralized design, there has been work in distributing network management functionality. Some work uses a hierarchical approach to network management [8, 20, 10]. In these systems, one or more top-level managers communicate with distributed mid-level managers to perform resource management activities. Because the mid-level managers are distributed over the network, these systems scale better than centralized systems. Additionally, there is typically some support for handling failure of one or more manager processes.

There have also been systems proposed using a P2P design for networked management systems [2, 18]. In particular, Panisson et al. [15] propose a modification to the SNMP protocol whereby the network contains nodes of three different roles, two types of managers as well as an agent processes.

Our work is similar in that we use a P2P design to solve the fault tolerance and scalability problems with centralized solutions. However our work differs in two fundamental ways. First, PeerMon is designed to provide system-wide resource monitoring and resource usage data collection only. It is not a network management system, but provides lower-level monitoring and data collection. Thus, its design is much less complicated than this other work and as a result, can be better optimized for its system-wide data collection task independently of how its data may be used by higher-level services. A higher-level resource management system could be implemented as a client of PeerMon data rather than being integrated into PeerMon. The second difference is that every PeerMon peer is an equal peer. The result is a purer P2P design than this other work; one that provides a more layered and flexible architecture for designing resource management systems, and one that is more fault tolerant and scalable.

Other work related to ours is in the area of resource scheduling and load balancing tools for networked systems. There has been a lot of work in this area, most focusing on cluster and grid systems [7, 13, 11, 4, 16, 17, 19].

Condor [13] and the Now/GLUnix project [7] are two examples that are designed, in part, to run on general purpose networked systems like ours. NOW/GLUnix implements a cluster abstraction on top of a network of workstations that are simultaneously being used by individual users as a general purpose LAN. GLUnix stores the global state of the network on a single master node. This state is updated by daemon processes running on each node, which periodically send their local resource usage information to the master. The data are used to support resource allocation and parallel and sequential job placement by the master.

Condor implements a job submission and scheduling system for running parallel and sequential applications on LANs, clusters, and grids. When run on general purpose LANs, Condor discovers idle nodes on which to run jobs. When a node running a Condor job is no longer idle, Condor uses process migration to move Condor jobs to other idle nodes in the system. Condor uses a centralized manager and local daemons to collect system-wide load statistics and to perform process control.

GLUnix and Condor provide much higher-level services and abstractions than our work, but both collect system-wide resource usage data on the same types of target systems. PeerMon provides only the underlying system for data collection, but uses a P2P design instead of a centralized one. PeerMon could potentially be used to provide data to higher-level system services like Condor or GLUnix.

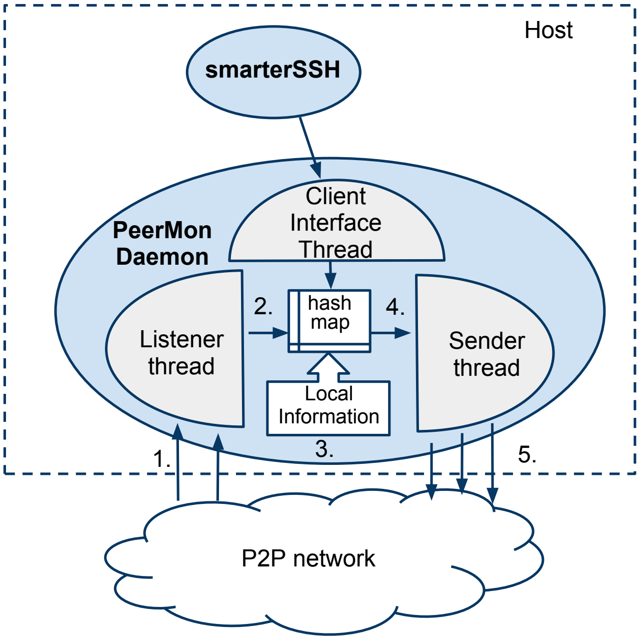


Figure 1: *PeerMon Architecture. Each host runs a PeerMon daemon . The Listener thread receives UDP messages from the P2P network (1) and updates the hashMap with the newest data (2). The Sender thread periodically wakes-up and updates the hashMap with local node resource data (3). It then selects three peers to send its hashMap data via UDP messages (4 and 5). Applications, like smarterSSH, interact with the PeerMon Client Interface thread via a TCP/IP to obtain PeerMon system-wide resource usage data.*

3 The PeerMon System

PeerMon is designed to run on a general purpose networked system where users can log into any node at any time and run any mix of parallel and sequential programs, and batch and interactive applications. The three main goals in designing PeerMon are: to efficiently provide, in real-time, system resource usage information; to scale to large-sized systems; and to be fault tolerant. The system also needs to be flexible enough to allow nodes to easily enter and leave the P2P network. Additionally, because each node continuously runs a PeerMon daemon, it is important that PeerMon uses minimal network and other system resources.

To meet these goals we chose a P2P design for PeerMon. Each node in the network runs a PeerMon daemon, which is an equal peer in the system; there is no central authority nor is there a hierarchical relationship among peers. Every node in the system provides system-wide resource usage data to its local users. Thus, users of PeerMon data need only contact their local daemon to get information about the entire system.

When a PeerMon node fails, the rest of the system continues to function; non-failed nodes continue to use PeerMon data collected from their local PeerMon daemons. Data from failed or unreachable nodes ages out of the system and will not be included as a ”best node” option by system services that use PeerMon.

Recovery from failure is easy. When a node comes up, it starts a PeerMon daemon that reconnects to the system by sending its information to three peers. Once other peers hear of the new peer, they will send it system-wide resource usage data in subsequent peer data exchanges. Our tests show that it takes on average eight rounds of message exchanges for a new peer to become fully connected into the system.

To reduce the amount of network traffic between peers, we use the observation that it is not necessary, nor is it possible, to have completely accurate system-wide resource usage information in general purpose networked systems. Even in a centralized resource monitoring system, the data do not represent an instantaneous snapshot of system-wide state [12]. PeerMon is designed so that each peer collects system-wide resource information, but individual PeerMon nodes may have slightly different information about the system. Distributed PeerMon data do not need to have the same type of consistency constraints as distributed file system and database data do. Thus, we do not need to implement expensive synchronization to support consistency of data across all peers. As long as each PeerMon peer has relatively recent resource-usage information about the system, its data is just as accurate and useful as data provided by a centralized system.

Higher-level services that use PeerMon data to implement load balancing or job-placement combine PeerMon data with accounting of their activities to make policy decisions. These higher-level services could be implemented as centralized, hierarchical or distributed independent clients of PeerMon. The constraints on higher-level service determine which PeerMon peers it would use to make policy decisions. This is no different than how such systems would use data from a centralized resource monitoring system. PeerMon, like other resource monitoring systems, does not need to account for how its data may be used by higher-level services.

3.1 System Architecture

Figure 1 shows the structure of the multi-threaded PeerMon daemon process. The Listener thread receives messages from other peers containing system-wide resource statistics. The Sender thread periodically wakes up, collects resource usage information about its local node and sends a copy of its system-wide state to three other PeerMon peers. The Client Interface thread exports the peer's collected system-wide state to local applications that want to use PeerMon data.

IP	TS	TTL	Indegree	payload
130.52.62.123	5	7	4	(char *...)

Table 1: *Structure of a hashMap entry.*

Each PeerMon daemon stores its resource usage data in a data structure called the hashMap. The Listener and Sender threads update hashMap data in response to receiving or collecting newer resource information. The Sender and Listener threads communicate using UDP/IP sockets and the Client Interface thread communicates with applications using TCP/IP.

3.1.1 Resource Usage Data

Each PeerMon daemon stores system-wide resource usage information in a data structure called the hashMap. Table 1 shows the structure of a hashMap entry. Each entry in the hashMap contains information associated with a specific machine (node) in the network. The set of information stored includes the IP and port number of the node and PeerMon Listener thread, and the payload that contains the resource usage data from that node. Currently, PeerMon is implemented to collect and store resource usage information in the payload field, but the general structure is such that it could be modified to store other types of data.

The time to live (TTL) field approximates the age of the data. Its value is decremented each time stamp (i.e. each time the Sender thread sends information to three other peers). The Indegree field counts the number of messages that a node has received in the latest interval between two time steps. The time stamp (TS) field contains the last time stamp when the node was directly contacted from this PeerMon daemon. The TTL, Indegree, and TS are used by heuristics to select the three peers to send hashMap data to at each time step. The TS field is stored locally and is not sent to other nodes. All other hashMap data are shared with peers.

3.1.2 Sender and Listener Threads

The Sender thread periodically wakes up, collects resource statistics about its local node, adds them to its hashMap, and then selects three peers to send its hashMap data. The Listener thread is responsible for receiving hashMap entry data from other peers and updating its local hashMap with all or some of these data. The Sender thread decrements the TTL field of each entry each time it performs this operation. The TTL field approximates how old the resource data are for each node. The Listener thread compares TTL fields of entries in its hashMap and entries received from peers. If the peer data has a larger TTL value, it updates the hashMap with the peer data (i.e. this is more recent data about the node). If the current hashMap entry's TTL value is larger, it does not update its hashMap with the data from the peer (i.e. the current entry represents more recent data about the node than the data the peer sent). Currently, the TTL field's value starts at 10. Experiments run on networks of sizes 25-500 show that this value works well to ensure both recent data and high connectivity.

We chose to have the Sender and Listener threads use UDP/IP sockets to communicate to avoid TCP connection overheads each time peers wanted to communicate. As long as most UDP messages are delivered, an occasional dropped packet will not affect the quality of the data in the system. Because absolute correctness of the data cannot be guaranteed, losing an occasional packet will have little or no effect on the system. When the node receives other system-state messages, the window of time during which it missed an update about a node's state is small. If packet loss is common, then the added guarantees provided by TCP may be worth its higher communication overhead.

Because UDP is used to send hashMap data, care must be taken to ensure that loss of a single packet does not cause a Listening thread to block forever waiting for a lost message. To ensure this, the Sender thread sends several independent messages containing parts of its hashMap data to each node. Each independent message fits into a single packet so that if a packet is dropped, the Listener thread will never block trying to receive it; it just receives and handles the next independent message never knowing that it received one fewer message than a Sender thread sent.

3.1.3 Heuristics used to select Peers

To ensure that all nodes have recent usage information about all other nodes in the system, care must be taken in selecting which of three peers the Sender thread sends it hashMap data to. We developed three heuristics for selecting peers that, when used in combination, do a good job of distributing new data to all peers and of maintaining a high degree of connectivity in the P2P network. Each time the Sender thread wakes up, it applies one of the three heuristics. The heuristics are cycled through in round-robin order.

The first heuristic, named "Contact New Nodes", picks peers that are relatively new to the network. Since PeerMon nodes can enter or leave the P2P network at any time (e.g. due to node failure and restart) this heuristic ensures that new nodes in the system collect system-wide information soon after they connect the network. The heuristic picks peers with the smallest value of: $MAX\_TTL - TTL + Indegree$. The heuristics ensures that nodes with a high TTL (i.e. nodes whose information is new) and a low Indegree (nodes who have not been sent to recently) are selected. The heuristic results in new peers being quickly integrated into the system; however, its use alone can lead to P2P network partitioning.

The second heuristic, "Contact Forgotten Nodes", selects the three nodes with the lowest TTL (i.e. nodes that the present node has heard from least recently). The third heuristic, "Contact Old Friends", is designed to ensure that a node cannot become permanently isolated. It uses the TS field values to choose peers that it has not sent data to recently.

The combination of three heuristics works well to prevent network fragmentation and to allow for new nodes to quickly become fully incorporated into the system.

4 Current Implementation of PeerMon

PeerMon is implemented in C++. It runs on the Swarthmore Computer Science Department network of about 60 Linux 2.6/x86 machines. Our system has some heterogeneity in that machines have different numbers of cores (we have 2, 4 and 8 core machines), different amounts of RAM, and slightly different processors. All machines are connected by a switched 1Gbit Ethernet network.

PeerMon daemons collect local resource data for CPU load, amount of free RAM, and number of users through the Linux /proc interface on the node on which they run. PeerMon can be modified to collect and distribute other data. Currently, this would require changing PeerMon code. In the future we plan to add a programming interface that would allow users to more easily change the set of data PeerMon collects and change how it collects it.

4.1 Starting Up a PeerMon Daemon

The PeerMon executable takes several command line arguments that can be used to run and configure PeerMon in different ways. Figure 2 shows the command line options that include specifying the port number for the Listener thread, running the daemon in collector-only mode, starting with a user-defined configuration file, and specifying the number of seconds the Sender thread sleeps between collecting local information and sending its hashMap data to three peers.

When a PeerMon daemon starts-up it reads information from a config file that contains addresses of three PeerMon nodes. These are the first three nodes that the Sender thread contacts to start the exchange of system-wide resource data.

If the PeerMon daemon is started in collector-only mode it will receive resource usage information about other nodes, but sends "invalid" information about itself.

```
peermon -p portnum [-h] [-c] [-f configfile] [-n secs]
   -p  portnum:   use portnum as the listen port for peermon
   -c:            run this peermon daemon in collector-only mode
   -f conf_file:  run w/conf_file (default /etc/peermon/peermon.config)
   -n secs:       how often daemon sends its info to peers (default 20)
```

Figure 2: *Command line options to peermon daemon.*

Other nodes, upon receiving ”invalid” data, will not include the collector node’s information in data it exports to its local users. This allows a collector-only node to use PeerMon data but not make itself a candidate for other node’s use. We run PeerMon in collector-only mode on our DNS server so that other nodes will not choose it as a target for ssh or spawning application processes.

Each machine in our system is configured to start a PeerMon daemon when it starts-up. Each machine also periodically runs a cron job to detect if the PeerMon daemon is still running, and if not, re-starts it.

4.2 PeerMon Data Interface

Users (clients) of PeerMon data, such as smarterSSH, obtain PeerMon data by connecting to the Client Interface thread and sending it a message requesting its hashMap data. TCP sockets are used to connect to the Client Interface thread.

In our current implementation, the PeerMon daemon also exports its hashMap data by writing it to a file on the local file system. The PeerMon Sender thread replaces the old contents of the file with updated hashMap values each time it wakes up. Clients can access PeerMon data by opening and reading this file. There is a potential race condition between the reader and writer of this file. However, because we do not plan to support the file interface in the final version of the code, we ignore handling the unlikely event of a read/write conflict to this file (in practice we rarely see it). The file interface was our initial client interface to PeerMon before adding the Client Interface thread, and is currently used to help with debugging of our system.

Although the file interface is easier for clients to use than the TCP interface, it has two problems: the first is the potential read/write race condition to the shared file that could result in clients reading garbage or incomplete data; the second, and more serious, problem is that there is non-trivial overhead associated with re-writing the file contents each time data are collected. With the TCP interface the PeerMon daemon only exports its data when they are being used by a client.

In the future we plan to implement a higher-level programming interface for PeerMon clients that will hide the underlying TCP interface in an easier to use library.

5 Example Applications that make use of PeerMon data

The initial motivation for developing PeerMon was to design tools that could make better load balancing decisions in general purpose network systems by considering system-wide resource usage data. As a demonstration of how PeerMon data can be used for such purposes, we developed three tools: smarterSSH; autoMPIgen, and dynamic DNS binding based on resource load.

5.1 smarterSSH

smarterSSH is our tool for choosing the ”best” ssh target node based on PeerMon data. It is implemented in Python and has several command line options that allow a user to specify different criteria for ordering the ”best” node(s) and to select different runtime options.

The following are the command line options to smarterSSH:

```
-c: order nodes by CPU load
-m: order nodes by free memory
-n num: print out the best num nodes
        rather than ssh into the best
-i: verbose printing mode
```

By default, smarterSSH orders nodes based on a combination of their CPU load and amount of free RAM using the function: $\frac{freeMem}{1+CPUload}$ (1 is added to prevent division by 0).

When run with no command line options, smarterSSH connects to its local PeerMon daemon to obtain its hashMap data, sorts the data based on the combination of CPU load and free RAM, randomizes the order of equivalent nodes, and ssh’s into the top node from the sorted result. Running with command line options `-c` or `-m` sorts the data by CPU load only or free RAM only. The ordering functions use small delta values to place nodes into equivalence groups so that small differences in free RAM or CPU load are not deemed significant.

Running with command line options `[-n num]` causes smarterSSH to print out an ordered list of its top `num` nodes rather than ssh’ing into the ”best” node.

As an example, Figure 3 shows output from a run of smarterSSH with the command line options: `-c -i -n 10`. This run will order nodes by CPU load only, and will

```
host      CPU load   free RAM    cores
--------------------------------------
avocado   0.000      13068052      8
pimento   0.000      15828112      8
orange    0.000      2933896       4
cucumber  0.000      6291932       4
dill      0.000      5967724       4
ginger    0.000      3170436       4
marjoram  0.000      7049804       4
molasses  0.000      6881228       4
anise     0.030      14659024      8
perilla   0.020      5597020       4
```

Figure 3: *Example output from a run of* `smarterSSH -c -n 10 -i` *(print out the top 10 "best" nodes as ordered by CPU load). Eight of the nodes are equally good with a CPU load of 0.0. anise is ranked higher than perilla because it has 8 cores vs. 4.*

print out the top 10 nodes rather than ssh'ing into the top node. In this example there are eight "best" nodes, all with CPU load 0.0. Each time smarterSSH is invoked, it randomizes the PeerMon data so that the total ordering among equal nodes varies. This means that subsequent runs of the above command could result in a different ordering of the first eight nodes. Randomization is used so that multiple invocations of smarterSSH will distribute the load over the "best" nodes while these new ssh's have not yet had time to effect system load.

5.2 Automatic MPI host file generation

autoMPIgen is another tool that uses PeerMon data to perform load balancing in general purpose LANs. It automatically generates MPI host files by choosing the best nodes based on PeerMon's system-wide resource use data. It is written in Python and is very similar to smarterSSH. When run, autoMPIgen interacts with the local PeerMon daemons to obtain system-wide resource usage information. It has command line options to allow the user to specify how to order machines and how to configure the resulting OpenMPI [5] hostfile [1] containing the "best" machines.

The following are the command line options to autoMPIgen:

```
-n num: choose total num nodes
-f filename: specify the output file.
-c: order best nodes by CPU load only
-m: order best nodes by free RAM only
    (default is combination CPU and RAM)
-i: printout results to stdout
-p: include a node's number of CPUs
    in the generated hostfile
-cpus:  interpret the num value from
        (-n num) as number of cores
```

As an example, using the PeerMon data from Figure 3, autoMPIgen run with the command line options `-n 9 -c -p` generates the following hostfile (the 9 best hosts ordered by CPU load and including the core count in the hostfile ("slots=n")):

```
avocado slots=8
pimento slots=8
orange slots=4
cucumber slots=4
dill slots=4
ginger slots=4
marjoram slots=4
molasses slots=4
anise slots=8
```

A run adding the additional command line argument `-cpus` interprets the `-n 9` value to mean CPUs rather than nodes, and generates the following hostfile (best machines with at least a total of 9 cores):

```
avocado slots=8
pimento slots=8
```

5.3 Dynamic DNS

Our third example of using PeerMon data is to incorporate it into dynamic domain name server (DNS) binding. [1] This allows a virtual host name to be mapped to one of the set of "best" physical nodes where "best" nodes are selected based on system-wide load.

Using PeerMon data to select a set of "best" nodes has several benefits over BIND's support for load distribution that selects a host to bind to using either round-robin or random selection from a fixed set of possible hosts. Our solution allows for the "best" host to be selected based on current system resource load, thus adapting to dynamic changes in system resource usage and resulting in better load distribution. Our solution is also resilient to nodes being unreachable due to temporary network partitioning, node failure, or to deliberate shut-down of nodes in order to save on energy consumption during times of low use. In BIND, if the selected host is not reachable, then ssh hangs. Using our system, unreachable or failed nodes will not be included in the set of "best" targets. When a node is reachable again, PeerMon will discover it and the node may make its way back into the set of "best" targets.

Adding support for dynamic DNS binding using PeerMon data is fairly easy if you have control over your own domain name server. In our department we run our own DNS server and control both the name-to-address and the reverse address-to-name mappings for our sub-domain (`cs.swarthmore.edu`.) The following is a summary of the steps we took to add support for dynamic binding to nodes chosen using PeerMon data:

1. Run PeerMon on our domain name server in collector-only mode.

2. Periodically (currently once per minute) update the resource records for our sub-domain so that one hostname (cslab.cs.swarthmore.edu) has n address records associated with it (we have n set to 5). These 5 machines are selected using data from the local PeerMon daemon.

3. Use the round-robin feature of BIND 9 to rotate through the 5 addresses when queries for cslab.cs.swarthmore.edu are made

The first step requires that PeerMon is running on potential target machines in our system and on the DNS server. We run PeerMon daemons on most of our machines (we exclude most servers and a few other machines that are designated for special use). The DNS server runs the PeerMon daemon in collector-only mode, which will exclude it from being a target of smarterSSH, autoMPIgen, or any other tool using PeerMon.

The second and third step for adding PeerMon data into the DNS records require that we first enable the dynamic update feature of BIND 9 by adding an ”allow-update” sub-statement to our DNS zone configuration file:

```
zone "cs.swarthmore.edu" {
 type master;
 file "cs.db";
 allow-update {127.0.0.1;130.58.68.10;};
};
```

Next, a script to update DNS records based on PeerMon data is added as a cron job that runs once per minute. When run, the script first contacts its local PeerMon daemon to obtain system-wide resource usage data to determine the 5 ”best” machines. For example, suppose these are currently the five best machines based on PeerMon data:

```
130.58.68.41
130.58.68.70
130.58.68.162
130.58.68.74
130.58.68.148
```

The script next generates a file of commands for nsupdate (part of the BIND 9 software), deleting the old records first, and then adding new A records (an example is shown in part (a) of Figure 4.) As a last step, the script runs ”nsupdate” on the generated file to change the DNS records (the results on the example are shown in part (b) of Figure 4):

The round-robin feature of BIND will map cslab.cs.swarthmore.edu to one of these 5 ”best” nodes until the cron job runs again to change the mapping to a possibly new set of the 5 ”best” nodes.

Our implementation led to a couple difficulties that we had to solve. First, every PeerMon daemon must have the same ssh host key. Otherwise, when users repeatedly ssh to cslab, each time getting a different machine from the PeerMon list, ssh would warn them that the host identification has changed for cslab.cs.swarthmore.edu. We solve this problem by giving all machines running PeerMon the same ssh host key, and distributing an ssh_known_hosts2 file that reflects this fact.

The second difficulty had to do with editing DNS data files. Because we are using dynamic DNS, a program running on our DNS server updates our domain data files every few minutes. A serial number in the domain data file is used to signal the change in the zone's data, which means that the serial number for the zone data is being changed with each dynamic update. This poses no problem until we need to manually edit the domain data file (e.g., to add a new name-to-address mapping). To solve this problem, our system administrators must first ”freeze” the zone, then make manual editing changes, and then ”unfreeze” the zone. BIND 9's rndc command makes this fairly easy:

```
$ sudo rndc freeze
 (edit the data files here, being
  sure to update the serial number)
$ sudo rndc thaw
```

Once set up, students and faculty can ssh into cslab.cs.swarthmore.edu and be automatically logged into a machine with the lowest load in the system. Because we update the mappings every minute, and because remote ssh is not a frequent system activity, the result will be good distribution of remote ssh's accross nodes in our system. Another benefit is that users do not need to remember specific machine names to log into our system; they simply ssh into cslab.cs.swarthmore.edu and are placed on a good machine.

By using PeerMon data, machines with high loads, machines that are unreachable, or machines that have been shutdown will be excluded from possible hosts. This not only means that there is better load balancing using PeerMon data, but that our approach to dynamic DNS binding is resiliant to network partitioning and node failures. No longer do users log in to machines that are already heavily loaded, or try to log into a machine, only to see their ssh process timeout. A benefit for our system administrators is less editing of the DNS data files. If a machine is taken out for service, it is automatically (within a minute or two) removed from the pool of best-available machines, requiring no manual editing of the DNS data files. When a machine is restarted, it will quickly be added back into the PeerMon network

```
(a) example generated file contents:
    --------------------------------
update delete cslab.cs.swarthmore.edu.
update add cslab.cs.swarthmore.edu. 30 IN A 130.58.68.41
update add cslab.cs.swarthmore.edu. 30 IN A 130.58.68.70
update add cslab.cs.swarthmore.edu. 30 IN A 130.58.68.162
update add cslab.cs.swarthmore.edu. 30 IN A 130.58.68.74
update add cslab.cs.swarthmore.edu. 30 IN A 130.58.68.148
<a blank line is necessary here>

(b) results after executing nsupdate:
    ---------------------------------
$ host cslab.cs.swarthmore.edu
cslab.cs.swarthmore.edu has address 130.58.68.70
cslab.cs.swarthmore.edu has address 130.58.68.74
cslab.cs.swarthmore.edu has address 130.58.68.148
cslab.cs.swarthmore.edu has address 130.58.68.162
cslab.cs.swarthmore.edu has address 130.58.68.41
```

Figure 4: *Dynamic DNS impementation details: (a) an example generated file containing update command for nsupdate; and (b) output from running* `host` *after the script runs* `nsupdate`*.*

and will automatically be a candidate target for dynamic DNS binding.

6 Performance Results

We present results mesuring the performance of PeerMon in terms of its overheads, the degree of P2P network connectivity, the age of system-wide resource data, and its scalability to larger networks. We also present results using the tools we developed that make use of PeerMon data to perform load balancing in our network. Our results show that these tools significantly improve the performance of application programs running on our system.

6.1 PeerMon P2P Network Conectivity and Age of Resouce Usage Data

To evaluate the connectivity of PeerMon peers, we simulated a network of 500 nodes by running 10 instances of a PeerMon daemon process on each of 50 machines in our lab. Each daemon was started with a time stamp value of 5 seconds [2] (how frequently the Sender thread wakes-up and collects and distributes usage data).

P2P network connectivity is computed as the average number of nodes contained in each daemon's hashMap divided by the total number of nodes in the network. A connectivity of 1 means that every node in the network has current information about every other node in the network. For networks up to size 500, we consistently saw connectivity values above 0.99.

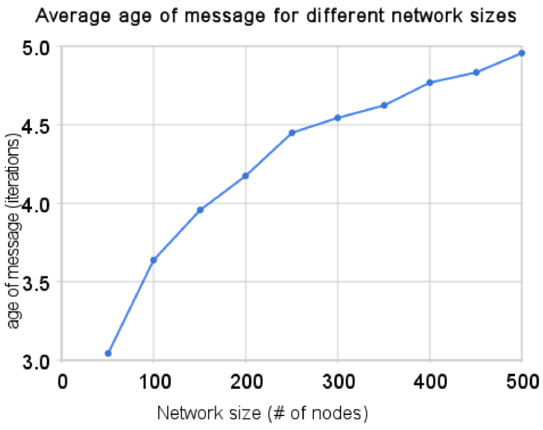
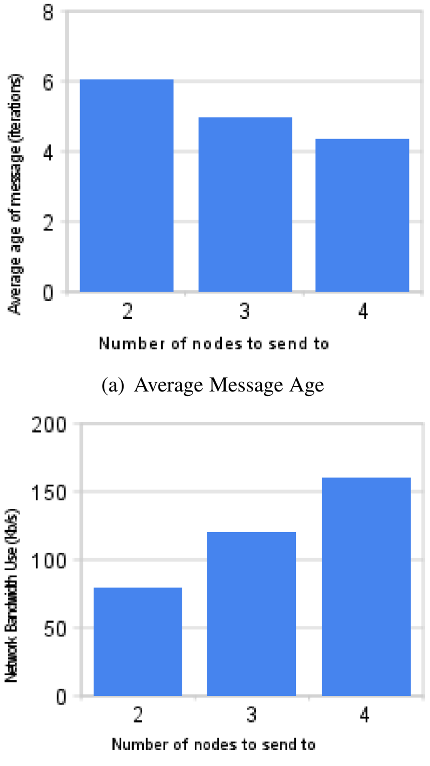


Figure 5: *Average message age across all nodes in the network for various network sizes.*

In addition to connectivity, we computed the average age of hashMap data for different sizes of networks. Figure 5 shows the results. For a network of size 50, the average age of data is about 3 iterations (roughly 15 seconds old). The average message age increases with the size of the network. For a network size of 500, the average age of a message is about 5 iterations (roughly 25 seconds old).

Additionally, we ran experiements to help us determine a good value for the number of peers that the Sender thread should send to each time it wakes-up. We ran experiements of different numbers of send-to peers on a network of 500 nodes. The results, in Figure 6(a), show that that as the number of peers increases (x-axis) the average message age decreases (y-axis). However, Figure 6(b) shows that as the number of peers increase, the PeerMon network load increases linearly.

Picking a good number of peers to send to each time step involves acheiving a good balance between maintaining newer data and good P2P network connectivity and maintaining low messaging overheads. Our results indicate that a send-to value of 2 is too small, resulting in older resouce usage data and potentially less than full connectivity. A send-to value of 4 results in an average data age of about 22 seconds with 100% connectivity; however, nearly 150 Kb/s of data are sent from each node in the network. Based on our results, we chose 3 peers as producing a good balance between achieving low data age (about 25 seconds on average), high connectivity (around 99.5%), and moderate network bandwidth usage (about 120 Kb/s).



(a) Average Message Age

(b) Network Bandwidth Use

Figure 6: *Average message age (a) and bandwidth used (in Kb/s) (b) on each node for different send-to values on a network of 500 nodes.*

6.2 PeerMon Scalability

To evaluate how well PeerMon scales to larger-sized systems, we ran multiple instances of PeerMon on each of 55 machines in our system to simulate systems of larger sizes. We ran experiements with 1, 2, 4, 10, 20, and 40 PeerMon daemons per real machine to simulate LANs of size 55 to 2,200 nodes. For these experiments we used the default 20 second rate at which the Sender thread sends to three of its peers. We ran a script on our monitoring server to periodically get MRTG [14] data to obtain a trace of five minute averages of network, memory and CPU load for every machine in our network. In order to ensure that our results for different sized systems were comparable, we ran experiments over a weekend when our system was mostly idle so that PeerMon was the primary load on the system. The data collected from each of the physical 55 machines in our network were divided by the number of PeerMon daemons running per host to obtain the per-node values for the larger systems that we simulated.

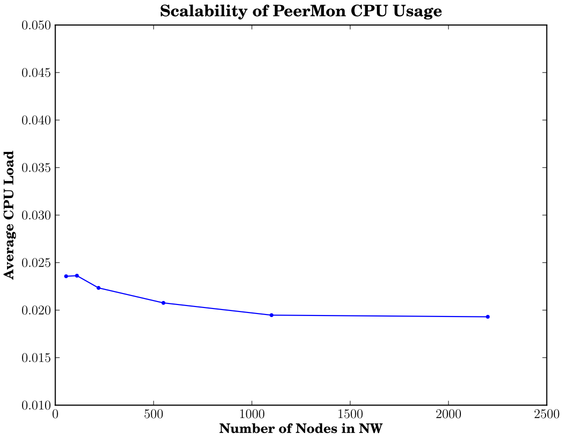


Figure 7: *The average additional CPU load per PeerMon host for different sized networks (Numbers of Nodes). The results show a basically fixed-size per-node CPU load as the PeerMon network increases.*

Figure 7 shows CPU load per PeerMon node and Figure 8 shows the amount of free RAM per PeerMon node for different sized networks. Both per-node CPU load and per-node RAM use stay relatively fixed as the network size increases. As the system-size grows, each PeerMon node has a larger hashMap data structure. However, the amount of memory storage and CPU processing that this data structure requires is so small that the overheads for a network of 2,200 nodes are equivalent to overheads for a network of 55 nodes. These results show that neither RAM nor CPU use will limit PeerMon's scalability to larger sized LANs.

Figure 9 shows the number of bytes sent and received per second per PeerMon node for different sized networks (from 55 to 2,200 nodes). The amount of network traffic remains very small as the size of the network grows. On the 2,200 node PeerMon network each

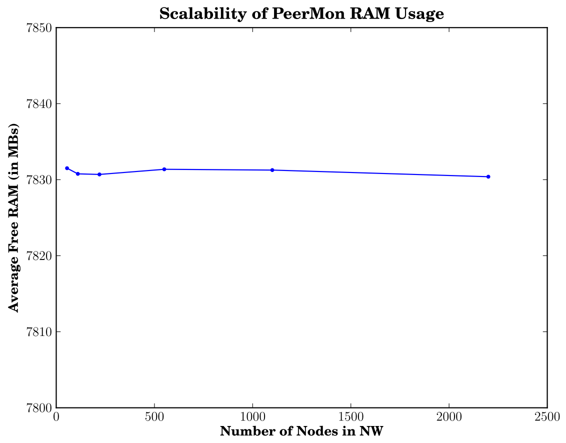


Figure 8: *The Amount of Free RAM (in MB) per PeerMon host for different sized networks. These data show that PeerMon uses little RAM space, and that the amount it uses per node stays fixed as the size of the network grows.*

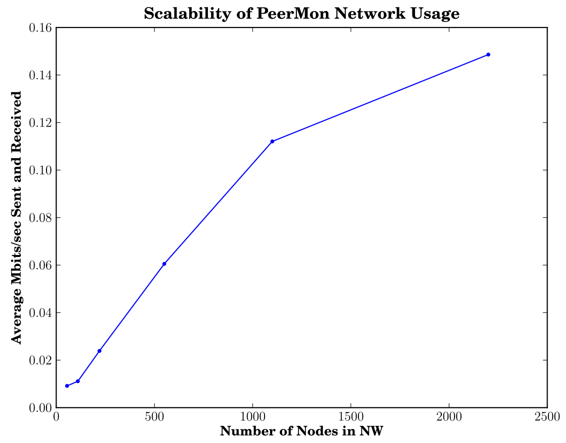


Figure 9: *The average Network load per PeerMon host for different sized networks. The data are the average Mbits/second sent and recieved per node. The data show that although there is a slight increase in network bandwidth used per node as the PeerMon network size increases, the amount used per node is still a small fraction of the total bandwith available to the node.*

PeerMon daemon uses less than 0.16 Mbits/second on its 1 Gbit connection. However, there is an increase in the amount of data each PeerMon daemon sends to its three peers as the network grows (the number of peers sent to by each PeerMon daemon is constant, but the size of each message grows with the number of nodes). On a 55 node network, each PeerMon deamon's hashMap has at most 55 entries. On a 2,200 node network, each hashMap can contain up to 2,200 entries. Each time the Sender thread wakes up and sends its hashMap contents to three peers, the total number of bytes sent to each peer grows with the size of the network.

Even for a network with 2,200 nodes, our results show that PeerMon adds very little network overhead and that its network use scales well to the types of systems for which it was designed. However, the data show some added network costs as the size of the network grows. The decision for each PeerMon daemon to send its full HashMap contents works well for the systems we are targeting, but it could become a bottleneck if PeerMon were to be deployed on a system with tens or hundreds of thousands of nodes. In this case, its design may need to be changed so that each peer exchanges only partial hashMap contents.

Our results show that PeerMon scales well to large-sized systems of the type we are targeting. It adds only negligable amounts of network, RAM, and CPU load to the system.

6.3 Results Using smarterSSH and autoMPIgen on Application Workloads

The initial motivation for developing PeerMon was to implement tools that could distribute user and program load in general purpose networked systems. Therefore, as a way to evaluate this use of PeerMon data, we ran experiments using smarterSSH and autoMPIgen to select the best nodes on which to run sequential and parallel MPI applications.

The experiments were run during a time when our system was heavily used so that there was variation in system-wide resource usage. For some experiments we additionally ran artificial workloads on some nodes to ensure more variation in resource usage across nodes. For these experiments, we needed to ensure some variation in resource usage, because if all nodes are basically equal, a randomly chosen node will be just as good as one chosen based on PeerMon's system-wide resouce usage data.

We evaluated the results of running different applications in the network using smarterSSH and autoMPIgen to select the nodes on which to run the applicaiton. We found significant improvements in application runtime when using our tools. The results were consistent across a broad range of tests.

For each experiment we compared runs of a benchmark program using smarterSSH or autoMPIgen to pick the ”best” node(s) to runs of the benchmark on randomly selected nodes (representing no use of PeerMon data). For the smarterSSH runs, we tested all three node ordering criteria (CPU only, RAM load only, and both). We ran each benchmark 200 times, doing 50 tests of random selection and 50 tests of each of the three smarterSSH ordering criteria. We interleaved the runs of each test so that the results would be equally affected by changes in system load over the course of the experiement.

Our first benchmark is a memory intensive sequential program that allocates a 2.8 GB array of integers and then iterates over the array elements ten times, modifying each array element as it goes. By reading and writing array values with each iteration, we ensure that if the entire array does not fit in available RAM, the application will trigger swapping on the node, which will significantly increase its total execution time.

The ”Memory” column in Table 2 lists speedup values of using smarterSSH over randomly chosen nodes. The results show that the run time using smarterSSH with CPU load ordering is not significantly different from random (speedup value of 0.87) [3] However, the two smarterSSH runs that use RAM load to select the ”best” node perform significantly better than randomly selected nodes (speedup values of 4.62). The speedup value of 0.87 for CPU, although not significantly different than random, does show that picking nodes based on CPU load alone for this benchmark will not necessarily result in good choices. Since this is a memory intensive benchmark, it makes sense to choose nodes based on their RAM load.

Our second experiment uses a primarily CPU intensive benchmark consisting of an openMP implementation of Conway’s Game of Life (GOL) [6]. The benchmark program runs on a single machine. It consists of a two threaded process that computes the Game of Life on a 512x512 grid for 1000 iterations. The column labeled ”OpenMP GOL” in Table 2 presents the speedup values obtained using smarterSSH vs randomly selecting a node. Our results show that speedup is significant for all three smarterSSH runs, with the combination ordering criterion performing slightly better than the others (speedup of 2.29).

The final benchmark program is an OpenMPI implementation of the Game of Life [4]. We ran the benchmark on a 10000x10000 grid for 30 iterations. The program consists of 8 MPI processes that are distributed across 8 different nodes in our system. The implementation proceeds in rounds where processes must synchronize with the others before starting the next round. As a result, the runtime is determined by the slowest process. autoMPIgen was used to automatically generate the MPI hostfiles for the runs using PeerMon data.

Node	**Benchmark**		
Ordering	Memory	OpenMP GOL	MPI GOL
CPU	*0.87*	1.63	1.27
Memory	**4.62**	2.19	1.78
Both	**4.62**	**2.29**	**1.83**

Table 2: *Speedup over random selection of machines using each heuristic on all three of the benchmarks. Cursive entries are not significantly different from random selection.*

For these experiments we ran a CPU intensive program on 9 of the 50 nodes to create imbalances in CPU load across machines in our system (18% of the machines in our network have a high CPU load). Using randomly selected nodes, there is a 85.7% chance that each trial would include one of the nine machines running our CPU intensive program. For the autoMPIgen runs, these 9 nodes should not be selected.

The speedup values are shown in the ”MPI GOL” column in Table 2. The results show autoMPIgen runs performing significantly better than random node selection. Ordering nodes based on both CPU load and RAM load results in the best performance (speedup of 1.83).

Our benchmark tests show that using PeerMon data to select good nodes based on CPU load and RAM load results in applications performing significantly better than when run on randomly selected nodes. In the worst case, ordering nodes by CPU load does not perform significantly worse than random. A knowledgeable user should be able to predict which ordering criterion is most useful for her program based on whether the program is more CPU-intensive or more memory-intensive. However, our results demonstrate that for all the benchmarks ordering nodes using the combination of CPU load and RAM load works best. This is likely due to the fact that all programs require a certain amount of both CPU time and RAM space to execute efficiently. Based on these performance results, we use a combination of CPU and RAM load as the default ordering criteria in smarterSSH and autoMPIgen.

7 Conclusions

Our results show that PeerMon is a low overhead system that quickly provides accurate system-wide resource usage data on general purpose LAN systems. Its peer-to-peer design scales well to large sized systems and is fault tolerant. Our example applications that use PeerMon data (smarterSSH, autoMPIgen, and dynamic DNS binding based on system load) demonstrate that PeerMon data can be very useful for implementing load balancing applications for systems that do not have centralized control of resource scheduling. Our benchmark studies show significant improvement in application performance using PeerMon data to make good choices about process placement in the system. PeerMon provides a system-wide data collection framework that can be used by higher-level tools that implement management, scheduling or other monitoring activities.

Future directions for our work include: investigating, collecting, and using other system-wide statistics in PeerMon; investigating scalability and security issues associated with supporting PeerMon running on multiple LANs; and further investigating ways in which PeerMon data can be used to improve individual application performance in general purpose LANs. Additionally, we plan to implement an interface to PeerMon clients that is easier to program than the current TCP interface. Our current plan is to implement a library interface that would hide the low-level TCP socket interface. We also plan to implement better support for extensibility by adding an interface to allow users to more easily change the set of system resources that are monitored by PeerMon.

Acknowledgments

We thank Tomasz Gebarowski for his OpenMPI implementation of the Game of Life that we used to benchmark our system.

References

[1] Albitz, P., and Liu, C. DNS and BIND, 4th Edition. O'REILLY, 2001.

[2] Androutsellis-Theotokis, S., and Spinellis, D. A survey of peer-to-peer content distribution technologies. *ACM Comput. Surv. 36*, 4 (2004), 335–371.

[3] Arpaci, R. H., Dusseau, A. C., Vahdat, A. M., Liu, L. T., Anderson, T. E., and Patterson, D. A. The Interaction of Parallel and Sequential Workloads on a Network of Workstations. In *Proc.ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (1995), pp. 267–278.

[4] Cluster Resources. MOAB Workload Manager. www.clusterresources.com/products/moab-cluster-suite.php.

[5] Gabriel, E., Fagg, G. E., Bosilca, G., Angskun, T., Dongarra, J. J., Squyres, J. M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R. H., Daniel, D. J., Graham, R. L., and Woodall, T. S. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting* (Budapest, Hungary, September 2004), pp. 97–104.

[6] Gardner, M. Mathematical games: The fantastic combinations of John Conway's new solitaire game life. *Scientific American 223* (1970), 120–123.

[7] Ghormley, D. P., Petrou, D., Rodrigues, S. H., Vahdat, A. M., and Anderson, T. E. Glunix: a global layer unix for a network of workstations. *Software Practice and Experience 28* (1998), 929–962.

[8] Goldszmidt, and Yemini. Distributed management by delegation. In *ICDCS '95: Proceedings of the 15th International Conference on Distributed Computing Systems* (Washington, DC, USA, 1995), IEEE Computer Society, p. 333.

[9] Harrington, D., Presuhn, R., and Wijnen, B. An architecture for describing simple network management protocol (SNMP) management frameworks. RFC Editor, 2002.

[10] Jette, M., and Grondona, M. SLURM: Simple linux utility for resource management. In *ClusterWorld Conference and Expo* (2003).

[11] Khan, M. S. A survey of open source cluster management systems. Linux.com (www.linux.com/archive/feed/57073), 2006.

[12] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (1978), 558–565.

[13] Livny, M., Basney, J., Raman, R., and Tannenbaum, T. Mechanisms for high throughput computing. In *Proceedings of the 21st SPEEDUP Workshop: Distributed Computing* (1997), vol. 11(1).

[14] Oetiker, T. MRTG Multi Router Traffic Grapher. http://oss.oetiker.ch/mrtg/.

[15] Panisson, A., da Rosa, D. M., Melchiors, C., Granville, L. Z., Almeida, M. J. B., and Tarouco, L. M. R. Designing the architecture of p2p-based network management systems. In *ISCC '06: Proceedings of the 11th IEEE Symposium on Computers and Communications* (Washington, DC, USA, 2006), IEEE Computer Society, pp. 69–75.

[16] PBS Works. OpenPBS. http://www.pbsworks.com/.

[17] Platform Computing Inc. Platform LSF. www.platform.com/workload-management/high-performance-computing.

[18] Santana, R. L., Gomes, D. G., de Souza, J. N., Andrade, R. M. C., Duarte, Jr., E. P., Granville, L. Z., and Pirmez, L. Improving network management with mobile agents in peer-to-peer networks. In *SAC '08: Proceedings of the 2008 ACM symposium on Applied computing* (New York, NY, USA, 2008), ACM, pp. 1874–1875.

[19] Skovira, S. K. M. R. P. M. D. B. J. F. Workload management with loadleveler. IBM RedBooks (www.redbooks.ibm.com/redbooks/pdfs/sg246038.pdf), 2001.

[20] Subramanyan, R., Miguel-Alonso, J., and Fortes, J. A. B. A scalable SNMP-based distibuted monitoring system for heterogeneous network computing. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)* (Washington, DC, USA, 2000), IEEE Computer Society, p. 14.

Notes

[1] autoMPIgen currently generates OpenMPI hostfiles, but could be easily changed to output hostfile formats for any MPI implementation.

[2] We think 5 seconds is too frequent for normal deployment, but it allowed us to run our experiments more quickly

[3] Significance testing was done using the Mann-Whitney U test.

[4] http://myitcorner.com/?page_id=2, by Tomasz Gebarowski