Distop: A Low-overhead Cluster Monitoring System

Daniel Andresen, Nathan Schopf, Ethan Bowker and Timothy Bower
Department of Computing and Information Sciences
234 Nichols Hall, Kansas State University
Manhattan, KS 66506
{dan, nathan, eab9844, tim}@cis.ksu.edu
Of£ce: (785)532-6350, Fax: (785)532-7353

Abstract

Current systems for managing workload on clusters of workstations, particularly those available for Linux-based (Beowulf) clusters, are typically based on traditional process-based, coarse-grained parallel and distributed programming. The DESPOT project is building a sophisticated thread-level resource-monitoring system for computational, storage and network resources [2]. The original implementation of DESPOT was based on SGI's Performance Co-Pilot (PCP) to facilitate the collection of performance monitoring data and to provide an API for the scheduling algorithm to retrieve the data. Unfortunately, the overhead of PCP and the infrastructure required to use PCP slowed down the performance of the DESPOT scheduling algorithms. In this paper we present an alternative to PCP which we call Distop. Distop was developed speci£cally to satisfy the needs of the DESPOT project for low-overhead, £ne-grained resource-monitoring tools for per-process network and other resource usage. We also present experimental results indicating the overhead of our system is minimal while providing accurate resource utilization data.

1 Introduction

Current systems for managing workload on clusters of workstations, particularly those available for Linux-based (Beowulf) clusters, are typically based on traditional process-based, coarse-grained parallel and distributed programming [13, 3]. In addition, most systems do not address the need for dynamic process migration based on differing phases of computation. The DESPOT project is building a sophisticated, thread-level resource-monitoring system for computational, storage, and network resources. We plan to use this information in an intelligent scheduling system to perform adaptive process/thread migration within the cluster. Full details of DESPOT architecture were presented in a previous paper [2].

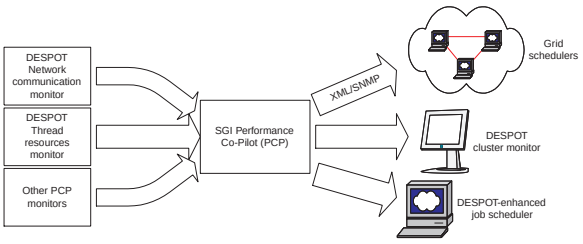

Figure 1. Original DESPOT architecture.

Of central importance to process scheduling systems such as DESPOT is the collection of system performance monitoring data which is used by the scheduling algorithm. Since DESPOT incorporates per-process and per-thread data into the scheduling algorithm, it has demanding performance monitoring requirements. The original implementation of DESPOT was based on SGI's Performance Co-Pilot (PCP) to facilitate the collection of performance monitoring data and to provide an API for the scheduling algorithm to retrieve the data. Unfortunately, the PCP proved to have too much overhead which slowed the performance of the scheduling algorithms. In this paper we present an alternative to PCP called distop, which was developed speci£cally to satisfy the needs of the DESPOT project.

Section 2 discusses related work. Section 3 covers the design of our system. Section 4 offers our initial experimental results, and £nally, Section 5 offers our conclusions and £nal thoughts.

2 Background

We £rst brie¤y discuss Beowulf clusters, and how they differ from a network of workstations (NOW) [13, 1]. We then go on to discuss various monitoring and scheduling tools such as PCP.

Beowulf clusters

The £rst Beowulf cluster computer was built by Thomas Sterling and Don Becker at CESDIS from 16 Intel 486DX4 processors connected by channel-bonded Ethernet. "The machine was an instant success and their idea of providing COTS (commodity off the shelf) base systems to satisfy speci£c computational requirements quickly spread through NASA and into the academic and research communities" [3]. Beowulf systems are marked by a reliance on COTS machines, publicly available software (in particular, the Linux operating system, the GNU compilers and programming tools, and the MPI and PVM message-passing libraries), and fast-node interconnects (typically Ethernet or Myranet). This software and hardware environment provides a robust system for today, with the ability to migrate easily to faster and better systems in the future due to standards like the Linux API, and message passing based on PVM[1] and MPI[2] [8, 5].

Performance monitoring tools

Several commercial and open-source tools have been developed to monitor the performance of a large number of computers such as a typical computing cluster. In contrast with existing systems, which usually display information only graphically, the DESPOT project integrates performance monitoring with scheduling systems. In the following sections, we discuss open-source cluster-monitoring tools.

Several tools have been developed to monitor a large number of machines as stand-alone hosts as well as hosts in a cluster. These tools can be useful because they monitor the availability of services on a host and detect if a host is overloaded, but they do not generally provide performance-monitoring information at the level of detail needed to tune the performance of a Beowulf cluster. Examples of these systems are PaRe Procps [14], BWatch [12], Mon [16], Nocol [11], and Netsaint [7].

[1] Parallel Virtual Machine
[2] Message Passing Interface

Detailed tracing of message-passing events in a cluster is afforded with the *Conch Visualization Package* from Georgia Tech [15]. This tool can be used to perform detailed traces of the execution of a distributed program, but the data reported is more relevant to tracing and debugging than measuring performance.

The *SMILE Cluster Management System* (SCMS) is an extensible management tool for Beowulf clusters [9]. SCMS provides a set of tools that help users monitor, submit commands, query system status, maintain system con£guration, and more. System monitoring is limited to heartbeat-type measurements.

The *Network Weather Service*, although not targeted at Beowulf clusters, is a distributed system that periodically monitors and dynamically forecasts the performance various network and computational resources can deliver over a given time interval [17, 18]. The service operates a distributed set of performance sensors (network monitors, CPU monitors, etc.) from which it gathers system condition information. It then uses numerical models to generate forecasts of what the conditions will be for a given time frame. NWS is used for various meta-computing systems such as Globus and APPLeS [6, 4].

MOSIX is a popular platform for supporting distributed computing. It enhances the Linux kernel with cluster computing capabilities. In a MOSIX cluster, there is no need to modify applications to run in the cluster, or to link applications with any library, or even to assign processes to different nodes. MOSIX does it automatically and transparently. The resource sharing algorithms of MOSIX attempt to equalize the processing load of the machines in the cluster. However, the scheduling algorithms only consider the total CPU load and memory usage of each machine. Per process load and network load measurements are not considered [10].

MOSIX was useful for our experiments for two reasons. First of all, it provides a framework and an API for migrating processes between machines. Thus it is a convenient platform for the development of prototype scheduling algorithms. Secondly, the built-in MOSIX scheduling algorithm offers a baseline measuring stick for comparing our own scheduling algorithms.

The *SGI Performance Co-Pilot (PCP)* provides a systems-level suite of tools that cooperate to deliver distributed, integrated performance management services. PCP provides the ability to quickly isolate and understand performance behavior, resource utilization, activity levels and performance bottlenecks. Performance data may be collected and exported from multiple sources, most notably the hardware platform, the IRIX kernel, layered services, and end-user

applications, and is returned in a structured text format.

PCP is the component of DESPOT that distop is designed to replace. While PCP has many features which may be useful in some applications, many of those features are not needed in the DESPOT project. The infrastructure associated with those additional features slows the system down enough that it is not able to adjust quickly enough to changes in resource usage. Because of this, we decided that a more ef£cient performance monitoring system was necessary.

3 System design

The distop package is intended to be used in a distributed computing environment to provide a lightweight means of collecting the overall system usage. The package is divided into three applications. There is a daemon, a server, and a client. The daemon program runs in the background collecting current system information, comparing it to previously collected system information, and storing the information as a ratio of difference over time. The server, upon request from the client, collects non-ratio system information, as well as obtaining the stored information from the daemon, and returns it to the client. The shared storage for the server and daemon is implemented using a shared memory segment along with a semaphore for mutual exclusion. For purposes of testing distop as a stand alone package, the client displays the information obtained from the server. When integrated with the despot project, the client is the scheduling algorithm.

Daemon: The daemon needs to run on any machine where system information needs to be collected. The daemon stores the calculated information in such a way that it will be easily accessible to the server. The daemon will run a continuous loop updating the stored information to ensure that the information is kept up to date.

Server: The server also runs on any machine where system information needs to be collected. The server, when started after the daemon, will wait for requests from the client. Upon receipt of a request from the client the server will collect system information that is of non- ratio type (i.e., it will collect total memory usage, current time, etc). The server will also obtain the most recent information that the daemon has stored. It will then return this information to the client.

Client: The client can run on any machine that is network enabled, and can connect to a machine that has a server and daemon running on it. The client then connects to the server machine and requests the system information from the server. The client and server communicate via RPC.

The server and daemon collect system information from £les stored in the `/proc` £le system. The `/proc` £le system is a virtual £le system containing £les which are kept up to date by the kernel.

Files from the `/proc` £le system that are of importance are:

1. `/proc/loadavg` — Load averages for the system.
2. `/proc/meminfo` — Overall system's memory information.
3. `/proc/stat` — Swapping, paging, and CPU information.
4. `/proc/net/dev` — Network device information.
5. `/proc/net/tcp` — Open tcp socket statistics.
6. `/proc/net/udp` — Open upd socket statistics.
7. `/proc/[number]/stat` — Individual process information.

All information collected by the daemon is ratio information (i.e., bandwidth, per- process CPU usage, etc). This is collected by having the daemon collect all initial information at startup. Then, every 5 seconds the daemon wakes up and collects the same information again. It calculates the differences between current information and the initial information, and then divides each quantity by the amount of time since the last reading. Finally, the per-second information is stored in the shared memory segment, the variables holding the initial information are updated with the current information, and the daemon goes back to sleep for 5 seconds. This goes on for the life of the daemon process.

Monitoring communication

Since connections are monitored at the socket/packet level, some connections are unable to be monitored completely. This would include communication through shared memory segments; or sockets to machines outside the cluster, in which case statistics are not known for the remote process.

It is not possible to monitor the amount of intra-node communication in generic Beowulf systems (i.e., those not running DSM or other distributed IPC systems) without creating several headaches. First, a large amount of kernel hacking would be required, including changes to sensitive areas of code – particularly the networking and shared-memory sections. Second, by adding monitoring code, we would end up signi£cantly impacting performance by adding latency to transmission times. And £nally, this performance-monitoring kernel code would have to be maintained as the kernel is revised.

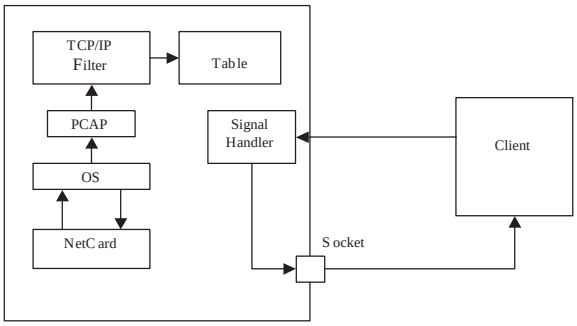

Figure 2. Traf£c capture module structure.

We could solve this intra-node communication problem by implementing a layer above the standard system library calls that records relevant information. This has the same pitfalls as a kernel modi£cation, but has the possibility of missing data through the use of static linking or nonstandard libraries in the observed applications.

The **Pcap** module of distop is responsible for collecting information about the amount of traf£c between each node in the cluster, and consists of two processes: a statistics process which gathers information from the network interface cards in promiscuous mode, and an agent process listening and honoring requests for these statistics The statistics process collects packet counts as well as total byte counts. Thus, we can also distinguish data transfer connections from interactive connections based on the ratio of bytes to packets (a low ratio probably indicates an interactive session). The module retains and reports only recent traf£c. Longterm trends should be recorded and analyzed with another module if needed.

The implementation is largely based on the PCAP library. *Tcpdump* is a front-end parser and pretty-printer of the information which libpcap gathers. The statistics process re-uses the tcpdump code, replacing pretty-printing with statistics-gathering code. Time stamps, byte counts, and packet counts are recorded in a hash table, which can then be queried through the second process. The PCAP library works through inserting an IP £lter within the kernel to promiscuously sort through all packets received by a node.

To avoid modifying libpcap, we had to use a two-process system (Figure 2). Currently, libpcap has its own event loop with no hooks to register new events for which to trigger call-backs (such as IO on a £le descriptor) except the traditional signal API. Thus, if queries from outside the machine are to be honored, a separate process is needed to listen for such queries, signal the statistics process, and relay the info to the network. This agent process can be implemented to honor INET socket-based requests, PCP requests, or requests of any other type.

Pcap gathers information about bandwidth usage by various processes, which is expressed in terms of packets per seconds and bits per second.

	Avg. CPU %	Avg. RAM (KB)	Avg. Ping (ms)
Not running	n/a	n/a	0.2
1 client (local)	1.7	2128	0.1
1 client (remote)	1.7	2135	0.2
2 clients	1.6	2133	0.2
3 clients	2	2128	0.2
4 clients	2.1	2642	0.3

Table 1. Resource consumption on 600Mhz Celeron server

	Avg. CPU %	Avg. RAM (KB)	Avg. Ping (ms)
Not running	n/a	n/a	1.2
1 client (local)	2.5	2152	0.9
1 client (remote)	2.1	2152	1.1
2 clients	2.5	2149	0.9
3 clients	2.6	2166	1
4 clients	2.57	2148	0.9

Table 2. Resource consumption on Dual 1.5Ghz Athlon server

4 Experimental results

To test the performance of distop, especially in relation to per-process bandwidth monitoring, we downloaded one of the CD images for Red Hat Linux from the mirror hosted by the University of Indiana (over Internet2) using ncftp, which down-loaded at an average rate of 2–3 MB/s. We took performance measurements for one local client, one remote client, and up to four clients (one local) up to four clients. Servers and clients ranged in capability from a 600Mhz. Celeron with 256MB RAM to a dual 1.5Ghz. Athlon with 2GB RAM. All machines were running the Linux 2.5 kernel. The performance measurements we took are average CPU usage and RAM usage on the server, and average ping time to the clients. The purpose of measuring ping time is to determine how much effect, if any, our use of libpcap has on packet latency. (As a control value, we measured ping time while the down-load was in progress without distop running.) The clients were polled at 2 s. intervals, and transmitted approximately 1KB of data per query.

Our results from Tables 3 and 3 indicate that our system, at its maximum resource consumption, consumes a small fraction of available resources on the server while providing accurate per-process resource usage data. Resource utilization on the clients and bandwidth utilization on a 100Base-T network for reporting the results was negligible, as was the additional latency imposed by the Pcap module. Peak bandwidth on the clients was also unaffected.

5 Conclusions and future work

In this paper we have presented our system for monitoring communication within a Beowulf cluster at the process-to-process level. , and shown how the system can also be used to monitor Other monitoring systems for Beowulf clusters are either methodology-speci£c (such as LAM), or present only aggregated communication results. We have also given experimental results indicating that the system has the low overhead and minor infrastructure requirements to be useful in our application domain.

We plan to extend the system and viewing application to a hierarchical organization to increase its scalability over the current, centralized system. We are also working to achieve the ability to monitor individual Java threads through identifying their mapping to kernel-level threads.

Acknowledgments This material is based in part upon work supported by the National Science Foundation under award numbers ITR-0082667 and ACS-0092839. Any opinions, £ndings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily re¤ect the views of the National Science Foundation.

References

[1] T. E. Anderson, D. E. Culler, and D. A. Patterson. A case for NOW (Networks of Workstations). *IEEE Micro*, 15(1):54–64, Feb. 1995.

[2] D. Andresen, S. Kota, M. Tera, and T. Bower. An ip-level network monitor and scheduing system for clusters. In *Proceeding of the 2002 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'02)*, Las Vegas, June 2002.

[3] D. Becker. *The Beowulf project*, July 2000. http://www.beowulf.org.

[4] F. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao. Application-level scheduling on distributed heterogeneous networks. In *Proceedings of Supercomputing'96*, Pittsburgh, PA, November 1996.

[5] M. P. I. Forum. MPI: A message-passing interface standard. Technical Report UT-CS-94-230, Department of Computer Science, University of Tennessee, Apr. 1994. Wed, 7 Jul 99 23:57:06 GMT.

[6] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.

[7] E. Galstad. *Netsaint Network Monitor*. http://www.netsaint.org/.

[8] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM 3 User's Guide*.

[9] P. R. Group. *SCMS Web Page*. Kasetsart University. http://smile.cpe.ku.ac.th/.

[10] *The MOSIX Project Homepage*. http://www.mosix.cs.huji.ac.il/.

[11] Netplex Technologies Inc. *Nocol System Monitoring Tool*. http://www.netplex-tech.com/software/nocol.

[12] J. Radajewski. *bWatch - Beowulf Monitoring System*. University of Southern Queensland, Apr. 1999. http://www.sci.usq.edu.au/staff/jacek/bWatch/.

[13] T. Sterling, D. Savarese, D. J. Becker, J. E. Dorband, U. A. Ranawake, and C. V. Packer. BEOWULF: A parallel workstation for scienti£c computation. In *Proceedings of the 24th International Conference on Parallel Processing*, pages I:11–14, Oconomowoc, WI, Aug. 1995.

[14] F. Strauss and O. Wellnitz. *Procps Monitoring Tools*. Technical University Braunschweig, June 1998. http://www.sc.cs.tu-bs.de/pare/results/procps.html.

[15] B. Topol. *Conch Visualization Package*. Graphics, Visualization and Usability Center; Georgia Institute of Technology. http://www.cc.gatech.edu/gvu/people/ Undergrad/Brad.Topol/conchviz.html.

[16] J. Trocki. *Mon System Monitoring Tool*. Transmeta Corporation. http://www.kernel.org/software/mon/.

[17] University of California San Diego. *The Network Weather Service Homepage*. http://nws.npaci.edu/NWS.

[18] R. Wolski. Dynamically forecasting network performance using the network weather service. *Cluster Computing*, April 1998.