Resource-Aware Stream Management with the Customizable *dproc* Distributed Monitoring Mechanisms

Sandip Agarwala, Christian Poellabauer, Jiantao Kong, Karsten Schwan, and Matthew Wolf
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
{sandip, chris, jiantao, schwan, mwolf}@cc.gatech.edu

Abstract

Monitoring the resources of distributed systems is essential to the successful deployment and execution of grid applications, particularly when such applications have well-defined QoS requirements. The dproc *system-level monitoring mechanisms implemented for standard Linux kernels have several key components. First, utilizing the familiar /proc filesystem, dproc extends this interface with resource information collected from both local and remote hosts. Second, to predictably capture and distribute monitoring information, dproc uses a kernel-level group communication facility, termed KECho, which is based on events and event channels. Third and the focus of this paper is dproc's run-time customizability for resource monitoring, which includes the generation and deployment of monitoring functionality within remote operating system kernels. Using dproc, we show that (a) data streams can be customized according to a client's resource availabilities (dynamic stream management), (b) by dynamically varying distributed monitoring (dynamic filtering of monitoring information) appropriate balance can be maintained between monitoring overheads and application quality, and (c) by performing monitoring at kernel-level, the information captured enables decision making that takes into account the multiple resources used by applications.*

1. Introduction

Motivation. The need for flexible and extensible monitoring tools for the high-performance computing environment, particularly for management and development of dynamic resource-responsive applications, has been documented many times [15, 17, 3, 8, 1]. Shared memory architectures provide an easy way to achieve such monitoring, but distributed memory machines present a more difficult target. The management of a distributed system is based on accurate and up-to-date monitoring of the system's relevant observable quantities and events. Management activities can include the optimization of application behavior, the distribution or balancing of application tasks between hosts, or the distribution of system resources, such as disk resources, to applications. Observable quantities include available processor, network, or disk bandwidths, and observable events include system failures, or the exceeding of resource utilization thresholds.

As scalable distributed memory and cluster computers pass to the hundreds or thousands of nodes, the perturbation implied by the frequent exchange of large monitoring data is clearly unacceptable in most circumstances. However, run-time monitoring of resources within a cluster is nonetheless desirable, since it permits the application designer to optimize performance under potentially highly dynamic conditions. Furthermore, monitoring is becoming increasingly important, as traditional high-performance computing (HPC) applications (which could presume dedicated access to a given set of resources), are being brought into more dynamic shared environments (such as computational grids and enterprise work-flow). Application-level optimizations needed to support these new environments may include dynamic spawning of subtasks to make use of newly-available resources, reallocation of workers from one parallel task component to another to achieve better load balance, application-driven check-pointing and migration of tasks, and dynamic optimization of network communications to improve communication/computation overlap. In all such cases, to achieve desired levels of performance, the requirements on timeliness of data force a perturbing monitoring solution.

Many systems have been suggested to address this issue. Recent approaches, such as Supermon [19] and MAGNeT [6], have exploited the enhanced performance and low-latency data paths associated with kernel-level monitoring. The dproc system described in this paper follows a similar path for kernel-level data collection. In addition, the dproc implementation supports full peer-to-peer communications at kernel-level, thereby improving communication performance through avoiding central master collection points (scalability of communications, fault tolerance), and by using strictly kernel-kernel messaging (avoiding user/kernel context switches). The advantages of kernel-level over user-level communication appears in [11], the principal outcome being that the variation in round-trip times is much larger for user-level compared to kernel-level communications.

The main characteristic of dproc considered in this paper is its support for *application-specific, customizable remote monitoring*. Customization includes well-understood functionality such as the provision of parameters that determine monitoring frequencies or thresholds. However, dproc acknowledges the need for more powerful dynamic customizations facilities by supporting *monitoring filters*. These filters are functions, specified by an application at run-time, that can be distributed via dproc to other hosts. They are then compiled dynamically, allowing for the online deployment of new monitoring functionality. Since the compilation takes place at the host that will execute and utilize this function, heterogeneity of hardware and software is supported while maintaining the performance advantages of binary code. These filters can customize monitoring behavior as follows:

- they can implement complex relationships between monitoring results (e.g., "monitor the available memory only if disk access times exceed a critical threshold");
- they can integrate application-level information with system-level information (such as the number of open connections in server applications correlated with the round-trip times on the corresponding sockets); and
- they can dynamically deploy monitoring functionality available in the remote kernel but not directly supported in dproc (such as the monitoring of the current battery power in mobile devices).

The result is an application-aware monitoring system which propagates only the monitoring data of interest across the cluster. Coupling this with the peer-to-peer communication infrastructure and the kernel-level data capture and analysis provides an efficient, scalable monitoring platform.

Our experimental evaluation in a cluster of 8 nodes shows that *dproc* performs monitoring with small overhead and low response times, which shows that dproc is an efficient and scalable solution to cluster monitoring. In this paper, the benefits of using the dproc monitoring system are demonstrated with a large scale real-time scientific visualization application called *SmartPointer*. We show in our experiments that monitoring of the resources like CPU, network, and disk at the clients provide the server application with the information necessary to customize the data streams it sends to clients. Such customizations decrease the total lag in the system and increase stream transfer rate. We also demonstrate the importance of kernel-based monitoring support by showing how monitoring multiple resources can prevent conflicting adaptations and increase total throughput.

Related Work. Different monitoring tools operate at different levels of granularity with consequent trade-offs between the quality of the information monitored and the overhead associated with it. Cluster performance monitoring tools have been developed to allow system administrators to monitor cluster state. A typical tool consists of two major entities: a *server* that collects state information of a cluster and a *GUI-based* front-end, which provides a visualization of system activity. Parmon [2], Ganglia [7], Smile [20], and many others belong to this kind. These tools cannot deliver very frequent monitoring updates.

Paradyn [15] is a tool that does performance monitoring for long running parallel and distributed applications. It adapts the performance of these applications by dynamically instrumenting them at run-time using the monitoring information that it collects. The Pablo [17] toolkit focuses on collecting and doing statistical analysis of performance data in scalable parallel systems. Falcon [9] is an application specific on-line monitoring system that provides its own set of instrumentation libraries and controls which the developer of an application can use to tune its performance.

The Supermon [19] cluster monitoring system uses a modified version of the SunRPC remote-status *rstat* protocol [10] to collect data from remote cluster nodes. This modified protocol is based on *symbolic expressions* which allows it to operate in a heterogeneous environment. The Supermon kernel patch exports various kernel monitoring information via a *sysctl* call. Scalability can be a problem in Supermon because of the centralized data concentrator, which collects monitoring data from all cluster nodes.

HPVM's performance monitor [18] is targeted towards Windows NT clusters. Like dproc, HPVM has the ability to automatically adapt cluster applications. The SHRIMP performance monitor [13] makes a compromise between high level software monitoring and low level kernel monitoring to accurately monitor various resource information. MAGNeT [6] uses an instrumented kernel to export kernel events to user space. It maintains a circular buffer in the kernel where all events are recorded and other nodes can obtain these records by contacting a daemon, called *magnetd*. The kernel must be configured at compile-time to enable the monitoring, which increases the administrative overhead as monitoring needs change.

In comparison, *dproc* provides a low overhead, fine-grained, kernel level monitoring facility, with communication based on strict kernel-kernel messaging. Dproc is extensible, i.e., new monitoring functionality can be added dynamically, e.g., through loadable kernel modules. Further, dproc is customizable, i.e., applications can fine-tune the distributed resource monitoring via parameters and dynamically generated code, as described in detail in the following sections.

2. Architecture

Dproc is an extensible, scalable, kernel-level monitoring toolkit for Linux-based cluster systems. The toolkit provides a single uniform user interface available through /proc, which is a standard feature of the Linux operating system. The /proc mount point provides a reasonably complete set of local monitoring data, including system load, memory utilization information, and per-process system resource utilization. The dproc project, whose name comes from 'distributed /proc', extends the local /proc entries of each of the cluster machines with relevant information from the other nodes within the cluster.

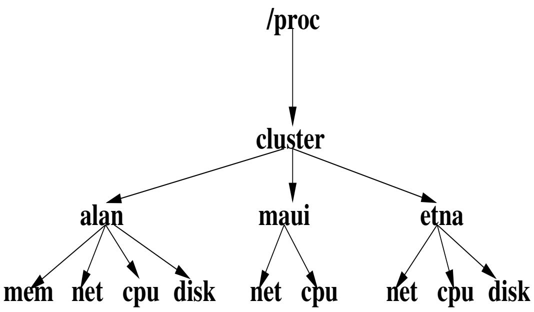


Figure 1. Dproc file hierarchy.

The toolkit adds a /proc/cluster entry into /proc, and the sub-directories contain the monitoring information for each of the registered cluster nodes. For instance, on node1, the *loadavg* information (average CPU run-queue length) from node2 would be located in /proc/cluster/node2/loadavg. For each node entry in /proc/cluster, there is also an associated control file, which a user-space application can modify to (a) specify monitoring parameters (e.g., thresholds or update periods) and (b) deploy dynamically generated filters for each individual resource or for all resources together. The associated control file for the previously mentioned node (node2) would be /proc/cluster/node2/control.

Typical hierarchies of pseudo-file entries in /proc are shown in Figure 1. Alan, maui, and etna are the names of the three nodes in the cluster. Memory, network traffic, CPU load, and disk usage are monitored on alan, network and CPU load on maui, and network, CPU load and disk usage on etna. Dproc makes this information available at each of these three nodes in the same hierarchy as shown in Figure 1.

The communications infrastructure is implemented using a kernel implementation of the ECho event channel infrastructure [5], called KECho [16]. KECho provides a publish-subscribe mechanism for direct kernel-kernel communications. Each kernel connects to both a data communication channel, or *monitoring channel*, and a *control channel*. To improve scalability, the exchange of data is triggered only when an application (either an executable or a user from a shell environment) registers interest in receiving some particular monitoring data.

The low-level implementation of dproc is described in detail in [11]. As a quick summary, it offers the following functionality:

- Selective monitoring via kernel-level publish/subscribe channels. The basic operating system construct offered by dproc is a pair of kernel-level channels, one for monitoring and one for control messages.
- Standard API. Applications need not explicitly handle monitoring channels. An application accesses dproc entries through an extension to the standard /proc interface.
- Flexible analysis and filtering. Dproc offers simple ways of dynamically varying certain parameters of monitoring actions, such as monitoring rates or desired statistics. Through simple reads and writes to control files within the pseudo-file system, an application can specify the relevant controls. We discuss this further in the following sections.
- Run-time configuration. Monitoring channels, handlers, and control attributes may be created, changed, and deleted at any time during the operation of dproc.

A simplified diagram of the dproc architecture can be seen in Figure 2. The *d-mon* (*distributed monitor*) module is the main module, coordinating the activities in dproc. It is built on top of KECho, which provides the kernel-level communication infrastructure as discussed earlier. Once the two channels are created (monitoring and control), different *monitoring modules* can register with d-mon using the *register_service* call which takes as parameter a callback function. D-mon maintains a list of all registered services and uses this callback function to retrieve monitoring information from them at regular intervals. D-mon modules use a channel registry, which is a user-level channel

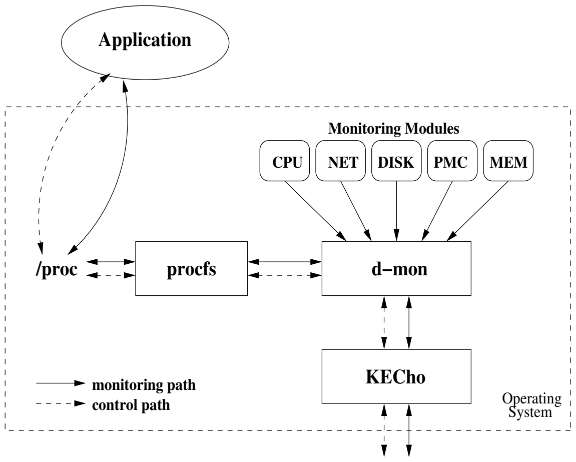


Figure 2. Dproc architecture.

directory server, to register new channels and to find existing channels. The first d-mon module to contact the registry will create the two channels. All other d-mon modules in the cluster will retrieve the *channel identifiers* from the registry and subscribe to the channels. Subsequently, applications can communicate with their local d-mon modules via the /proc interface and customize the behavior of remote d-mon modules using the dynamic filter approach discussed in this paper.

2.1. Monitoring Modules

Dproc supports a variety of monitoring modules, which will be described in this section. Further monitoring modules are under development and dproc's extension interface allows applications to dynamically deploy future monitoring modules without the need to recompile or restart the running dproc mechanisms.

CPU_MON. This keeps track of the average run-queue lengths over a period of time, which can be specified by the application. The default period is 1 minute. In a standard Linux system, */proc/loadavg* contains the average run queue lengths over 1, 5, and 15 minutes. This pre-specified time period average may not be useful in a fast system with constantly varying CPU load. Therefore, dproc's CPU monitoring module creates a kernel thread which wakes up periodically to examine the *task_list* in the kernel and computes the average of the run-queue lengths over an application-specified period.
MEM_MON. This provides information regarding the available memory. To obtain this information, the *nr_free_pages* kernel function is invoked.
DISK_MON. This measures the average number of disk writes and reads as well as the average number of sectors written and read for a certain period of time. The default period is 1s, however, as with CPU_MON, d-mon can change this value to any desired number.
NET_MON. This module monitors the round-trip times of established network connection, the used bandwidth of all connections at a node and of all individual connections, the number of re-transmissions (for TCP), the number of lost messages (for UDP), and the end-to-end delay for both TCP and UDP connections.
Performance Monitoring Counters (PMC). Most modern processors offer performance monitoring counters, which are accessible from kernel-level. The counters can be configured to track certain low-level processor events (depending on particular processor model), such as cache misses, number of operations, and other potentially interesting chip-level statistics. Many projects and products have attempted to extend this low-level monitoring to the application level. In our work, we implement a method for exposing this monitoring not only locally (through the pseudo-files and associated control of /proc) but also to the entire dproc-enabled cluster. This enables an application-level process on one machine to remotely extend the kernel of a participating cluster machine, so as to generate very fine-grained monitoring data which may be relevant to a particular application. For instance, monitoring of the number of cache lines loaded (through cache misses) may give a remote master process a way to track the amount of data that a worker process has consumed, allowing it to better tune its data distribution policies.

3. Extensibility in dproc

This paper focuses on the extensibility and filtering capabilities which have been added to the initial implementation as described in [11]. With dproc, we offer *parameters* and *dynamic filters* as means for customizing the communication and processing needs of the monitoring activities to the specific needs of applications or to the capabilities of the participating hosts.

Parameters. We distinguish between two types of parameters: (a) ones that modify the *update period* of information and (b) ones that modify *thresholds*. The update period indicates how often (expressed in seconds) an application would like to be informed about the current utilization or availability of a resource. Another option is to specify thresholds, i.e., an application indicates upper or lower bounds on resource utilization. Once d-mon recognizes that a threshold has been exceeded, the new value is distributed to the remote /proc entries. Combinations of these two are also possible, e.g., an application can specify to "update the CPU information once every 2 seconds IF the CPU utilization is above 80%". Threshold comparisons can be expressed as percentage limits (e.g., if x varies by 10% from the last measurement), or as fixed relative values (e.g., if $x < y * 1.1$ or $x > y * 0.9$), or as minimum and maximum values (e.g., if x is in the range $[y, z]$). These parameters allow us to dramatically reduce the amount of monitoring traffic, and hence increase scalability. For example, for a batch-queue scheduler, we might need load average updates only if it is less than the number of CPUs.

Dynamic Filters. An application can deploy filters by writing the filter code as string to the control file in /proc. It is d-mon's responsibility to distribute the string to the corresponding hosts via KECho's control channel. Incoming filter strings are received by d-mon, which then dynamically generates binary code. The resulting filters are executed by d-mon before any information is submitted to the channel, allowing the filters to customize (or block) the monitoring information. Dynamic filters can provide the same functionality as described in the parameters section above. However, they can provide more powerful data transformations as well, e.g., they can combine multiple decision making strategies. Note that although dynamic filters can provide the functionality of parameters, it is typically 'cheaper' to use parameters to specify simple rules because parameters require less book-keeping, and there is no dynamic code generation overhead.

To extend the previous example, imagine that the batch-queue scheduler is not interested in loadavg, but instead in the amount of free memory. However, it still wants the memory information to be updated only if there is a free CPU to run its process on. So it will tie the update period of the memory information to the load average dropping below the number of CPUs. In this way, we achieve a much more flexible application-level concept of quality of service for the monitoring data, with consequently lower perturbation.

Filter generation in dproc is based on a kernel port of a binary code generator, called E-code [4]. E-code offers a small subset of the C programming language, supporting the C operators, *for* loops, *if* statements, and *return* statements. Code written in E-code can be passed as strings between hosts via dproc's control channel and is executed at the publishing host. Unlike specialized monitoring languages like GEM [14], the general nature of E-code allows the application programmer to create arbitrarily complex subscription criteria.

Figure 3 show a sample filter code using which we can manipulate the information being sent out by a dproc node in the cluster. This manipulation reduces the amount of data sent out to different nodes in the cluster, which in turn reduces both network and CPU perturbation, thereby increasing the scalability and performance of the overall system.

```
{
    int i = 0;
    if(input[LOADAVG].value > 2){
        output[i] = input[LOADAVG];
        i = i + 1;
    }
    if(input[DISKUSAGE].value > 10000 &&
          input[FREEMEM].value < 50e6){
        output[i] = input[DISKUSAGE];
        i = i + 1;
        output[i] = input[FREEMEM];
        i = i + 1;
    }
    if(input[CACHE_MISS].value >
        input[CACHE_MISS].last_value_sent){
        output[i] = input[CACHE_MISS];
        i = i + 1;
    }
}
```

Figure 3. Filter code example.

4. Evaluation

All experiments are performed on a cluster of 8 quad-Pentium Pro 200MHz machines with 512KB cache and 512MB RAM. The cluster is connected via a switched 100Mbps Fast Ethernet and the cluster nodes run RedHat Linux 7.1 (kernel version 2.4.18). The communications infrastructure (KECho) as well as the dproc monitoring system are implemented as loadable kernel modules.

The goal of the experiments in this section is to show the following:

- application-specific filtering of monitoring information can reduce the overhead and perturbation caused by the monitoring mechanisms,
- monitoring information can be used to make intelligent decisions how to manipulate and customize data streams in order to reduce resource requirements and to adapt streams to a clients' capabilities,
- and resource monitoring information has to comprise information about multiple resources in a system to enable an application to properly identify and remove resource bottlenecks.

4.1. Microbenchmarks

We evaluate the scalability and performance of the dproc monitoring toolkit by conducting a set of experiments. For discussion purposes, we will concentrate on the following three cases:

- dproc with an update period of 1 second,
- dproc with an update period of 2 second,
- dproc with a 'differential filter': monitoring information is sent only if the utilization of a resource varies by at least 15% from the last measured result.

For this set of experiments, dproc monitors CPU load, disk usage, memory usage, and network traffic, resulting in monitoring events of about 50–100 bytes of information.

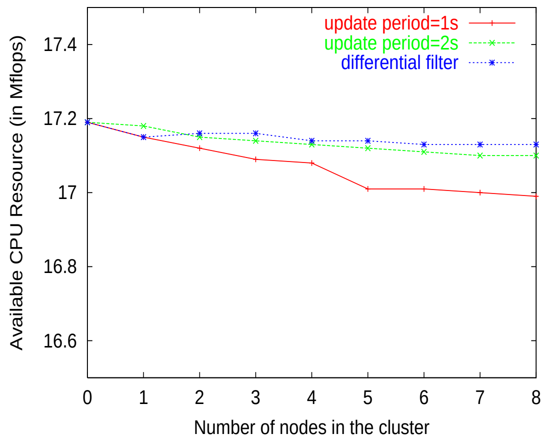


Figure 4. CPU perturbation analysis.

CPU Perturbation. The CPU overhead of dproc is evaluated in terms of performance of *linpack* [1]. Linpack is a CPU-intensive benchmark commonly used to measure the floating point computation power of CPUs in *Mflops*. We measure the change in linpack performance by running dproc on 0-8 nodes in the cluster and running linpack on one of them. Figure 4 shows that in all three cases the number of Mflops, as measured by linpack, decreases only slightly as the number of participating nodes increases in the cluster. However, the decrease in the measured Mflops is less accentuated in the case of the differential filter.

Network Perturbation. Dproc sends out monitoring information to other nodes in the cluster and therefore decreases the available bandwidth. We performed a network perturbation analysis using *Iperf* [2] *version 1.6* and found that the available bandwidth is hardly affected. We repeated the setup of the previous experiment and measured the available bandwidth between two nodes in

[1]http://www.netlib.org/linpack/

[2]http://dast.nlanr.net/Projects/Iperf/

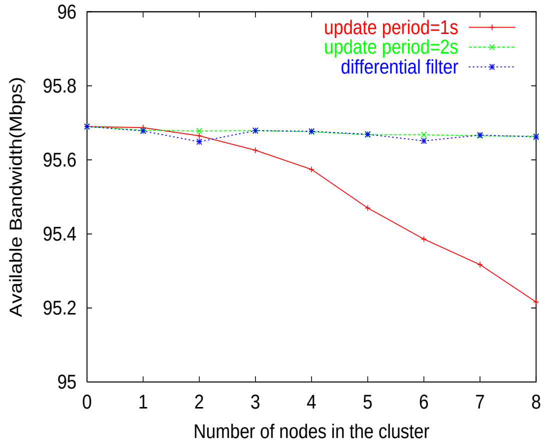


Figure 5. Network perturbation analysis.

the cluster by running Iperf in UDP mode. Figure 5 shows the results of the network perturbation analysis of dproc. The plot shows that dproc generates only small amounts of network overheads, e.g., the bandwidth drops by less than 0.5% for an update period of 1s and remains constant for update periods of 2s and the differential filter.

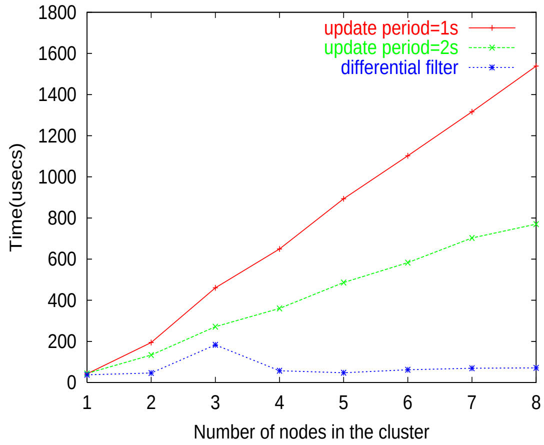


Figure 6. Event submission overhead.

Event Submission Overhead. Every second, d-mon polls each of the registered monitoring modules and sends the collected information to interested clients. Figure 6 shows the event submission overheads during one polling iteration of d-mon. Since the overhead is very small, it is measured by counting the number of CPU cycles using the *rdtsc* (Pentium time-stamp counter) instruction. The overhead is calculated by timing 100 polling iterations and taking the average. The plot shows that if the differential filter is used, the overheads remain within 100 microseconds, even for 8 nodes.

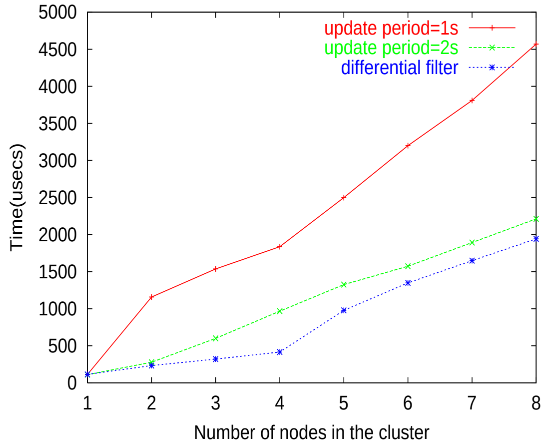


Figure 7. Submission overhead of events of larger size.

Figure 7 repeats the previous experiment, however, this time with monitoring events of average size 5KB. Although the overheads have increased, the results show a similar behavior as in Figure 6.

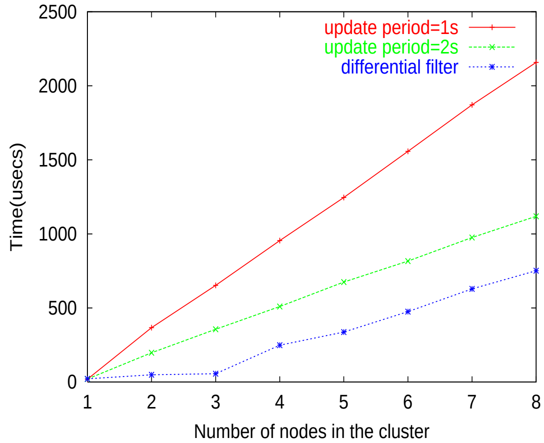


Figure 8. Overhead in receiving incoming events.

Event Receiving Overhead. Every second, d-mon polls the listening sockets to check whether an event has arrived or not. If there is an event in the *receive queue*, d-mon invokes the appropriate handler to consume the event. Figure 8 shows the overhead caused by handling these incoming events in one polling iteration of d-mon. Again, the overhead is calculated by timing 100 polling iterations and taking the average. The plot shows that even when the number of nodes in the cluster is increased to 8, the overhead remains less than 1ms in the case of an update period of 2s and the differential filter, and less than 2.2ms when the update period is 1s.

4.2. Scientific Collaboration

Application Description. The need to support scientists as they collaborate on a local, national and even international scale is a great challenge in visualization research today. Work which our group is already pursuing [21] is designed to leverage existing collaborative toolkits, such as the Access Grid[3] or the AGAVE project [12], and extend them with tools which are suitable for handling the very high data flows which the next generation of scientific computing will demand. Current scientific codes, such as the Terascale Supernova Initiative[4] funded under the DOE SciDAC program, can already generate relevant data at nearly a gigabit per second. Within the next 5 years, these sizes are expected to grow by at least a factor of 10. In order to provide a remote collaborator with an adequate and timely visualization, the data must be carefully staged and transformed to manage bandwidth, latency, and content representation. This become particularly relevant when multiple streams (such as data, video, and audio) must be synchronized. The sample application which we have designed to demonstrate the dproc infrastructure is based on a client-server model that complies with this vision.

The server delivers molecular dynamics data, similar to what might be generated by large scientific codes in Physics, Chemistry, or Mechanical Engineering (to name a few) to different clients which can range from high-end display like ImmersaDesk to smaller display like iPAQ, storage clients and fast desktop machines. The clients can subscribe to any of a number of different derivations of that data, ranging from a straight data feed, to down-sampled data (for example, removing velocity data), to a stream of images representing the full visualization. Communication is based on an event service, a server establishes an event channel and interested clients can subscribe to this channel in order to receive the data stream. Moreover, clients can customize the data stream by using *data filters*, similar to the concept of filters described earlier in the context of the monitoring data distribution. For example, resource-constrained devices such as wireless handhelds can down-sample a data stream using a filter, while other, resource-rich, devices can receive the full-quality data stream.

Without a monitoring system, a client must specify its requirements *a-priori* depending upon the expected

[3] http://www.accessgrid.org/
[4] http://www.phy.ornl.gov/tsi/

resource availability. If the availability of a particular resource changes beyond that scope, the client must specify a new filter to adapt. With dproc, we automate this process. The server can be made aware of the resource information of different clients via dproc. It uses this information to customize the data stream being sent to the clients.

Benchmark Measurements. In our experiments, dproc provides the following monitoring information from each client to the SmartPointer server: (a) CPU load, (b) available network bandwidth, and (c) disk utilization.

To measure the benefits of dproc, we compare the performance of the original SmartPointer application with a modified SmartPointer application using the dproc monitoring information. The data stream to all clients can be modified with a tunable data filter. The data filter reduces the information content of the data streams, and therefore the size of the data. The following three cases are compared:

- *SmartPointer application with no filter:* The SmartPointer server sends the original data stream to all clients without any customization.
- *SmartPointer application with static filters:* The SmartPointer server does the client-specified customization, but does not use the resource availability information from the clients. The customization criteria remains the same throughout the experiment.
- *SmartPointer application with dynamic filters using monitoring information:* The customization filters at the server use the clients' resource information from dproc to customize the data stream.

These tests were performed with three different kinds of clients: (a) a CPU loaded client, (b) a client connected to a link with highly varying network traffic, and (c) a hybrid client.

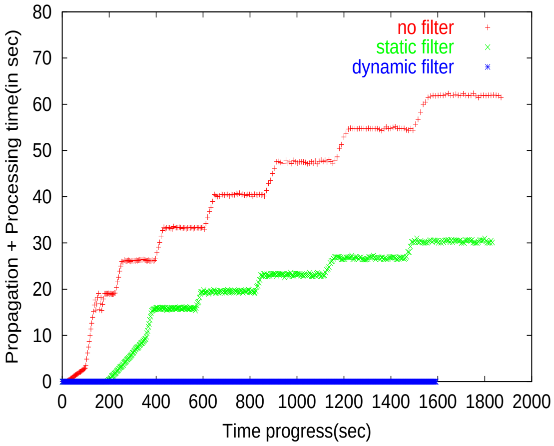
A CPU Loaded Client: The client was artificially loaded by running a CPU-intensive application, *linpack*, which was used earlier in our microbenchmark experiments. The load in the system was varied by running different instances of *linpack* processes. The experiments show that the scenario using dproc monitoring information outperforms the other two both in terms of scalability and responsiveness. Figure 9(a) shows the amount of time required for a data packet to be submitted by the server and processed by the client in a CPU loaded system. More than 99% of the time is spent in processing the event stream at the client. Every time a new linpack thread is started, the latency increases in the case where we are not using any filters or only the static filter, but it remains almost constant for the dynamic filter. The dproc toolkit informs the server about the change in load at the client and the dynamic filter uses this information to modify the size of data being sent to the client. The static filter scenario performs better than the scenario without any filter, but the delay is large in comparison to dynamic filters (which is less than 4ms). This small delay is due to events being queued when the load increases in the system.

Figure 9(b) shows the change in the event rate at a CPU loaded client. This figure shows that in the dynamic filter case, the client is able to receive and process events at the same rate at which the server sent them. Therefore the inter-event arrival delay remains almost constant. The static filter case cannot adapt itself to the increased load in the system and hence the queuing delays increase and the intervals between event arrivals get larger, although the server is sending these events at a constant rate. The case without filters shows the worst performance.

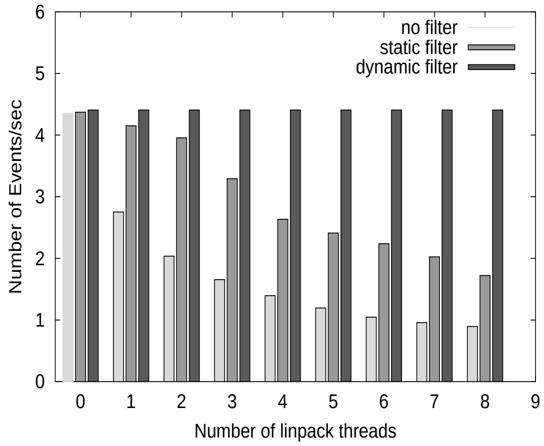
A Network-perturbed Client: In this experiment, the server sends much larger events (3MBytes) than in the previous case. To see the effect of network perturbation, the client does very little processing of incoming events. The link between the client and the server is artificially perturbed by running *Iperf* on two different nodes sharing a link between the former two and generating continuous streams of UDP packets. Figure 10 shows how latency is affected by the decrease in available bandwidth. As the network perturbation is increased, available bandwidth decreases, and latency increases. The capacity of the link is 100Mbps. When there is no perturbation, the server sends data to the client at a rate of about 30Mbps. Hence the plot remains horizontal until 70Mbps of perturbation. But as the perturbation increases beyond 70Mbps, latency drastically increases for the first two types of filters because the server is completely unaware of the state of its clients. The dynamic filter scenario, however, performs better than the others because the server reduces the data size.

A Hybrid Client: The client remains the same as in the previous experiment, but now we have both CPU and network perturbation using *linpack* and *Iperf*. We increase the number of linpack threads, thereby increasing the CPU load and the network perturbation by about 10Mbps. We compare three different types of dynamic filters in this case: (a) the first one uses CPU load information, (b) the second one uses network information and (c) the third one uses CPU, network, and disk information to customize the data streams.

Figure 11 shows that the performance is better when the filter uses more resource information to customize the data stream. Information about multiple resources allows the server to make 'smarter' decisions and prevents adaptations that could aggravate the resource shortages. When



(a) Latency variations with increasing CPU load



(b) Event rate variations with increasing CPU load.

Figure 9. SmartPointer performance in a CPU loaded system.

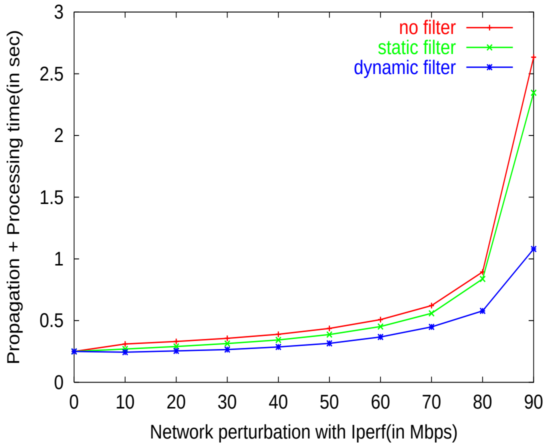


Figure 10. Change in latency with varying network traffic.

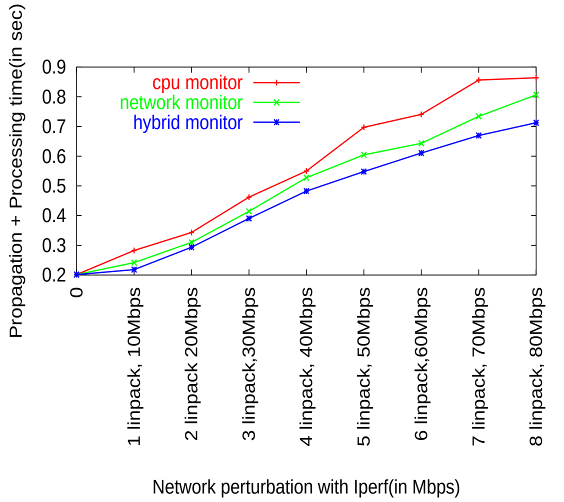


Figure 11. Change in latency with varying perturbation.

the CPU load at the client is increased, the filter will pre-process the data before sending it out so that the client requires less processing. This pre-processing increases the size of the data stream, which also increases the network requirements. Also, disk activity increases because of the increased data rate. If data is down-sampled to better fit in a congested network the client needs to do more processing before being able to render the data. Thus, we see that adaptation based on only one resource can have a negative effect on the requirements of another resource. This underlines the need of monitoring of multiple resources, in order to be able to make intelligent stream management decisions.

5. Conclusions and Future Work

This paper introduces the extensibility features of a scalable, kernel-level resource monitoring toolkit, called *dproc*. To our knowledge, dproc is the first cluster performance monitoring system designed to exploit kernel-level data capture as well as kernel-level peer-to-peer communications. Monitoring events and control information are sent to other nodes via the *KECho* publish/subscribe channels. The dynamic extensibility feature of dproc allows remote applications to specify complex subscription criteria in the form of *E-code* filters, which are compiled and deployed dynamically at run-time. Monitoring modules can also be added at run-time to extend the functionality of dproc. Our microbenchmark experiments show that the CPU and network overheads of dproc are almost negligible.

We also show the advantages of the dproc performance monitoring system by using a large-scale, distributed, scientific visualization application, called SmartPointer. The overall scalability and performance of the system is enhanced by using monitoring information provided by dproc. In particular, the use of dynamic filters using information about multiple resource can significantly improve data management decisions, e.g., by customizing the data stream according to the resource availabilities of each client.

Our future work will focus on using dproc in wide-area grids and embedded systems, which present a very dynamic and challenging environment. For example, in wireless and mobile systems, power has to be considered a first-class resource. Dproc is part of the *Q-Fabric* project, which attempts to provide large-scale distributed systems with a customizable resource management framework. The monitoring results delivered by dproc can be used by QoS management mechanisms to optimally allocate resources to applications and to integrate application adaptation with resource management.

References

[1] E. Al-Shaer, H. Abdel-Wahab, and K. Maly. HiFi: A New Monitoring Architecture for Distributed Systems Management. In *Proc. of the 19th Intl. IEEE Conf. on Distributed Computing Systems, Austin, TX*, pages 171–178, 1999.

[2] R. Buyya. PARMON: A Portable and Scalable Monitoring System for Clusters. *Software Practice and Experience journal*, 30(7):723–739, 2000.

[3] T. DeWitt, T. Gross, B. Lowekamp, N. Miller, P. Steenkiste, J. Subhlok, and D. Sutherland. ReMoS: A Resource Monitoring System for Network-Aware Applications. Technical report, Carnegie Mellon School of Computer Science, CMU-CS-97-194, 1997.

[4] G. Eisenhauer. Dynamic Code Generation with the E-Code Language. Technical Report GIT-CC-02-42, Georgia Institute of Technology, College of Computing, July 2002.

[5] G. Eisenhauer, F. Bustamante, and K. Schwan. Event Services for High Performance Computing. In *Proceedings of High Performance Distributed Computing (HPDC)*, 2000.

[6] W. Feng, M. Broxton, A. Engelhart, and G. Hurwitz. MAGNeT: A Tool for Debugging, Analysis and Reflection in Computing Systems. In *Submitted to the 3rd IEEE/ACM Intl. Symposium on Cluster Computing and the Grid*, 2003.

[7] Ganglia Toolkit: A Distributed Monitoring and Execution System. "http://ganglia.sourceforge.net/".

[8] C. Glasner, R. Huegl, B. Reitinger, D. Kranzmueller, and J. Volkert. The Monitoring and Steering Environment. In *Proc. of the Intl. Conference on Computational Science (ICCS), San Francisco, CA*, pages 781–790, 2001.

[9] W. Gu, G. Eisenhauer, K. Schwan, and J. Vetter. Falcon: On-line Monitoring and Steering of Large-Scale Parallel Programs. *Concurrency: Practice and Experience*, 6(2), April–June 1998.

[10] S. M. Inc. RPC: Remote Procedure Call Protocol Specification Version 2, 1988. http://www.ietf.org/rfc/rfc1057.txt.

[11] J. Jancic, C. Poellabauer, K. Schwan, M. Wolf, and N. Bright. dproc - Extensible Run-Time Resource Monitoring for Cluster Applications. In *Proc. of the Intl. Conference on Computational Science*, April 2002.

[12] J. Leigh, G. Dawe, J. Talandis, E. He, S. Venkataraman, J. Ge, D. Sandin, and T. DeFanti. AGAVE: Access Grid Augmented Virtual Environment. *Proc. AccessGrid Retreat, Argonne, Illinois*, January 2001.

[13] C. Liao, M. Martonosi, and D. W. Clark. Performance Monitoring in a Myrinet-connected Shrimp Cluster. In *Proceedings of 2nd SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 21–29, August 1998.

[14] M. Mansouri-Samani and M.Sloman. A Generalised Event Monitoring Lanaguage for Distributed Systems. *IEE/IOP/BCS Distributed Systems Engineering Journal*, 4(2):96–108, June 1997.

[15] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn Parallel Performance Measurement Tool. *IEEE Computer*, 28(11):37–46, November 1995.

[16] C. Poellabauer, K. Schwan, G. Eisenhauer, and J. Kong. KECho - Event Communication for Distributed Kernel Services. In *Proc. of the Intl. Conference on Architecture of Computing Systems (ARCS'02), Karlsruhe, Germany*, April 2002.

[17] D. A. Reed, R. A. Aydt, R. J. Noe, P. C. Roth, K. A. Shields, B. W. Schwartz, and L. F. Tavera. Scalable Performance Analysis: The Pablo Performance Analysis Environment. In *Proc. Scalable Parallel Libraries Conf.*, pages 104–113. IEEE Computer Society, 1993.

[18] G. Sampemane, S. Palkin, and A. Chien. Performance Monitoring on an HPVM Cluster. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '00).*, June 2000.

[19] M. Sottile and R. Minnich. Supermon: A High-Speed Cluster Monitoring System. In *Proc. of IEEE Intl. Conference on Cluster Computing*, 2002.

[20] P. Uthayopas, S. Phaisithbenchapol, and K. Chongbarirux. Building a Resources Monitoring System for SMILE Beowulf Cluster. In *Proceeding of the Third International Conference/Exhibition on High Performance Computing in Asia-Pacific Region (HPC ASIA'99), Singapore.*, 1998.

[21] M. Wolf, Z. Cai, W. Huang, and K. Schwan. SmartPointers: Personalized Scientific Data Portals in your Hand. In *Proc. of ACM Supercomputing*, November 2002.