In proceedings of the 17th International Parallel and Distributed Processing Symposium, Nice France, April 22-26 2003.
http://www.lanl.gov/radiant/pubs/magnet/ipdps2003.{ps|pdf}

MUSE: A Software Oscilloscope for Clusters and Grids*

Mark K. Gardner, Michael Broxton, Adam Engelhart, Wu-chun Feng
{mkg, mbroxton, adame, feng}@lanl.gov
Los Alamos National Laboratory
Los Alamos, NM

Abstract

Oscilloscopes and their cousins, logic analyzers, are the tools of choice for difficult electronic hardware problems. In the hands of a skilled engineer or technician, these tools can be used to solve stubborn problems. The key to the utility of oscilloscopes is the depth of detail they provide and their flexibility, which allows the level of detail to be adjusted to fit the task at hand.

Distributed applications, which run on computing clusters and computational grids, are also complex and difficult to tame. We need tools to understand their complexities and the ability to choose the level of detail to fit the task, whether the task be debugging, tuning, monitoring or controlling.

The MAGNET User-Space Environment (MUSE) has been designed as a "software oscilloscope" for computing clusters and computational grids. It is a toolkit for applications and developers to obtain detailed information about the environment on the host. The information can be used on-line or saved for off-line analysis. It has low overhead and allows the level of detail to be adjusted. Furthermore, MUSE monitors without requiring the modification or re-linking of applications. It has been designed to make it easy to develop "adaptive applications" — applications that are aware of their environment and can adapt to changes.

1 Introduction

Many contemporary architectures for high-performance computing are built from commercial off-the-shelf (COTS) components in order to leverage the rapidly increasing performance and decreasing cost of consumer hardware. Clusters of (high-end) consumer components connected with (near-) commodity networks and programmed in a message-passing style are the scientific workhorses of today. The next, or meta, level in high-performance computing is to view powerful clusters connected by a wide-area network, such as the Internet, as a computational resource. Such systems are called computational grids. Yet in spite of dramatic increases in computational power afforded by such architectures, writing, debugging and tuning parallel applications remains a painful task.

A large part of the problem stems from the highly asynchronous nature of distributed computations. Applications written to take advantage of large numbers of CPUs must overlap computation and communication in order to effectively use available resources. Not surprisingly, the causes of performance problems in distributed applications are often distributed and hence extremely difficult to identify without a global knowledge of execution history. Furthermore, the causes are often subtle issues of timing which make accurate global histories more important. Yet accurate global histories are very difficult to obtain. A means for monitoring distributed applications and the hosts on which they run is needed. Furthermore, global histories will need to be filtered to only contain events of interest, or they are likely to become unmanageable.

Monitoring frameworks for collecting and presenting significant events in the life of a distributed computation are being developed as part of, or in conjunction with, frameworks for writing such applications. (This is particularly true of computational grids due to their increased complexity.) As an example, NetLogger [9, 19, 20] and Autopilot [14, 15] are both monitoring frameworks which work well with the Globus [7] computational grid toolkit.

One of the challenges faced by monitoring frameworks is the selection of an appropriate level of detail at which to identify a problem. For example, it is sufficient to collect a periodic heartbeat from nodes in order to monitor the availability of computing resources in a computational grid. If, however, the problem is low performance caused by poorly timed arrivals of messages, details concerning message arrival times are required. Perhaps the performance problem is due to increased latency caused by messages arriving while the process for which they are intended is waiting for its next timeslice. In general, it is difficult to know beforehand what level of detail is required to diagnose and solve a particular problem. Not only is sufficient detail re-

*This work was supported by the U.S. Dept. of Energy's Laboratory-Directed Research & Development Program and the Los Alamos Computer Science Institute through Los Alamos National Laboratory contract W-7405-ENG-36. Any opinions, findings, and conclusions, or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DOE, Los Alamos National Laboratory, or the Los Alamos Computer Science Institute. Los Alamos Unclassified Report (LA-UR) 02-7169.

quired, but problems could be identified and solved much more quickly if the amount of detail could be varied on the fly.

We have developed a tool, which we call MUSE—MAGNET User-Space Environment — that allows the on-line monitoring of nodes of a computing cluster or computational grid. The information available through MUSE can be tuned to provide just the right level of detail for the task at hand, whether that task be debugging, tuning or status monitoring.[1] Furthermore, MUSE can be used to support the development of "adaptive applications" — applications that are aware of the environment in which they execute and can adapt their behavior based on that awareness.

As an example of an adaptive application that MUSE facilitates, consider a distributed visualization tool which steers through a large data set. The application consists of a renderer which is co-located with the stored data at a remote site and a user interface which executes on the scientist's workstation. When the available bandwidth is plentiful, the renderer can send raw frames to the user interface for display. This provides the maximum resolution to the scientist. If the network becomes congested, however, the renderer can reduce the frame rate or compress the data to provide better response times. The key capability needed to respond appropriately is for the application to know what bandwidth is available from the network. MUSE is designed to provide such information to applications.

2 Design of MUSE

As the name "MAGNET User-Space Environment" implies, MUSE provides an environment for user-space applications to make convenient use of the wealth of information which the MAGNET toolkit [5, 6, 10] exports from the operating system kernel. It was designed to consolidate the functions of event filtering and information synthesis into one component, `magnetd`. The resulting information can be saved to disk for later analysis or sent to (multiple) applications for immediate use. Thus, MUSE provides the infrastructure for debugging or tuning system performance, monitoring computing clusters or computational grids, or building adaptive applications.

In the following sub-sections, we present the architecture of MUSE and its main component, `magnetd`, then describe how `magnetd` filters events and synthesizes information. Finally, we describe how `magnetd` supports user-defined extensions through the use of dynamically linked data handlers.

2.1 `magnetd`

Figure 1 shows the architecture of MUSE. The main component of MUSE, `magnetd`, is designed as a multi-threaded daemon process with two main threads of execution. The first, called the data collection thread, is responsible for extracting event records from the MAGNET kernel buffer and processing them. The second, the master server thread, listens for command connections from clients interested in obtaining information from `magnetd` and creating server threads to service their requests.

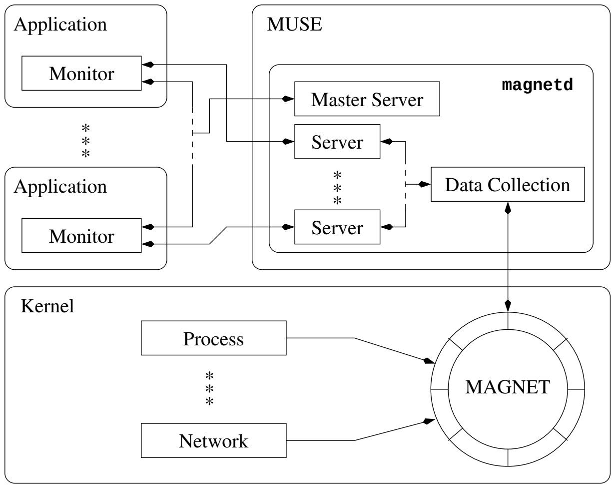

Figure 1. Architecture of MUSE

Although clients can request a copy of the MAGNET event stream, it is more efficient for most applications to request the subset of events that they find interesting. This greatly reduces the amount of information applications need to process and also minimizes the amount of communication required.

To emphasize the need for filtering, we have seen MAGNET trace rates as high as 1.76 million event records per second (33.6 MBps, where MBps = 2^{20} bytes per second) depending on the host and configuration. A single copy of the event stream sent to a remote application could easily consume a 100 Mbps Ethernet connection. Filtering the event stream before sending it to applications reduces the cost of communication and consolidates the filtering code for all applications. Functions which perform this service are called *filters*.

After the event stream is filtered, the remaining events can be synthesized into application-specific information through the use of data *handlers*. Handlers may perform arbitrary computations but should be as lightweight as possible since time-consuming computations are likely to perturb the phenomena being monitored and will reduce the time available for other handlers. Used judiciously, however, handlers further reduce the amount of communication and simplify application processing.

Clients communicate with `magnetd` by sending requests to the master server thread using UNIX or TCP/IP sockets. Local (UNIX) connections are useful for applications, running on the same host as `magnetd`, to query the state of the host. Remote (TCP/IP) connections are useful for the monitoring of computing clusters and computational grids by middleware or by the distributed application itself.

[1] Initial feedback from end users indicates that cluster environments require finer granularity than grid environments.

```
typedef struct magnet_data RECORD;
typedef struct data {int count;} DATA;

int help(int rsize, char *rstr) {
  strncpy(rstr, "counts events", rsize);
  return NO_ERROR;
}

int create(void *args, void **data,
           void **filters) {
  *data = malloc(sizeof(DATA));
  if (!*data) {return MEM_ERR;}
  ((DATA *) *data)->count = 0;
  return NO_ERROR;
}

int destroy(void **data) {
  free(*data);
  return NO_ERROR;
}

int process(void **data, RECORD *record) {
  ((DATA *) *data)->count++;
  return NO_ERROR;
}

int query(void **data,
          int size, char *result) {
  snprintf(result, size, "event count %d",
    ((DATA *) *data)->count);
  return NO_ERROR;
}
```

Figure 2. A Handler that Counts Events

Using the command connection, a client creates a handler to synthesize parameters of interest, such as measured bandwidth. It also adds filters to restrict which events are used to compute the parameters. The client also starts, stops, resets and queries the handler through the command connection.

Depending on how the handler is written, the synthesized parameters may be sent periodically ("push" model) or in response to a query from the client ("pull" model).

2.2 User-Defined Handlers

`magnetd` supports a plug-in architecture that makes it very easy for developers to write handlers to suit their application without recompiling `magnetd`. The object code for a handler is contained in a dynamically linked library. Figure 2 shows a simple handler that counts the number of events that match its filters.

Handlers are required to implement five functions: `create`, `destroy`, `process`, `query` and `help`. Each handler function is expected to return zero on success or a non-zero error code on failure. All of the functions except `process` are called in response to a command from the client, so the error code is propagated back to the client. Errors in the `process` function are logged to the console and `magnetd` stops the handler to prevent further errors.

The `create` function performs initialization, including allocating storage for the handler's internal state. Although the example does not show it, filters can be added within `create`, if appropriate. Newly created handlers are placed in a "stopped" state and must be activated by a separate "start" request from the client. When a client is finished with a handler, it sends a "destroy" request, which results in the handler's `destroy` function being called to perform appropriate clean-up activities, including deallocating the internal state.

The `process` function analyzes, stores, or streams events as appropriate. Updates to the handler's internal state also occur in `process`. Filters ensure that records received by the handler only contain relevant events. Handlers should be designed to minimize the execution time of the `process` function to minimize perturbing the quantities being measured. In the example, the count is incremented for each event received.

In response to "query" requests from clients, the `query` function prepares a string reflecting the values of the parameters computed by the handler. In the example, the response string reports the number of events that the handler has seen. For more details about handlers, see the documentation at `http://www.lanl.gov/~radiant/`.

3 Performance

The process of monitoring a computation has the potential to perturb the very parameters being measured. If the effects of the perturbation are small enough, they can safely be ignored. In this section, we bound the effects of using MUSE to monitor parallel scientific computations from the NAS Parallel Benchmark suite.

The two metrics we consider are the reduction in CPU cycles available due to the increased load caused by monitoring and the latency between when an event occurs and when the monitoring application receives the event. The former is important since fewer CPU cycles translates into slower execution of applications on the host. The latter is important for on-line monitoring purposes since stale information makes good decisions difficult.

3.1 Load Increase

We use the NAS Parallel Benchmark suite version 2.2 IS Class A kernel [16] as the workload. The IS kernel is a distributed bucket sort algorithm in which each processor sends the keys which fall in its range to all the other processors. Because IS communicates heavily, `magnetd` must process a large number of network events. The CPU cycles used by `magnetd` are unavailable for computing and hence reduce the operations per second reported by the benchmark. Each socket send of B bytes generates $1 + 3 * \lceil B/1448 \rceil$ events when MAGNET is configured to monitor throughout the network stack.[2] We also test the case where only socket send and receive calls are monitored.

[2] One event comes from the socket call itself. Three additional events occur for each fragment that traverses the network stack. Because Linux makes use of TCP options, the maximum fragment size is 1448 bytes.

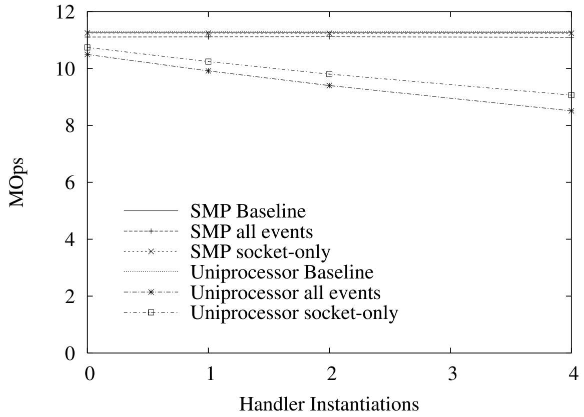

Figure 3. MOps vs. Number of Handlers

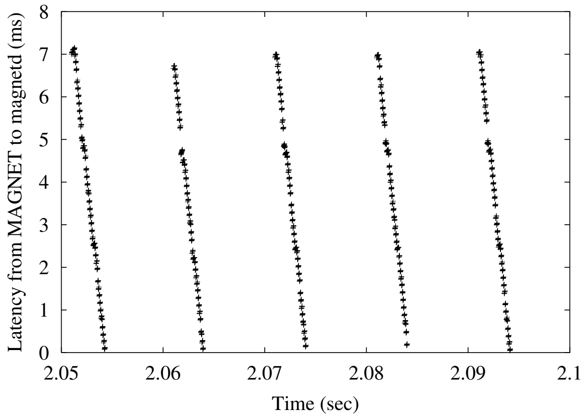

Figure 4. Latency Between MAGNET and `magnetd` with Sleeping on Idle

The test equipment consists of two identical machines with dual 933 MHz Pentium III processors and 512 MB of RAM connected by MPICH running over Gigabit Ethernet. Five configurations are presented. The first or baseline runs stock Linux 2.4.18 kernels. The second runs MAGNET-ized 2.4.18 kernels with `magnetd` executing on the monitoring host without handlers. The final three tests are the same as the second but with one, two and four handlers which count the number of events seen. The filters are configured to accept any network event.

The number of IS kernel operations per second as a function of the number of handlers is presented for both uniprocessor and SMP kernels in Figure 3. In all cases, the 95% confidence intervals are less than $\pm 0.03\%$.

First, we note that the performance of the stock kernel on multiple processors is nearly 0.5% lower than the stock kernel on a single processor due to locking overhead. Next, the number of IS kernel operations per second is reduced by only 1.3–1.4% in the SMP configuration with 1–4 handlers. The gradual increase in overhead indicates that SMP systems are not sensitive to the number of handlers. There is a higher impact on uniprocessor systems, however. The performance is reduced by 7.2–24.7% when `magnetd` competes with the application for processor cycles.

Due to the fact that the IS kernel communicates heavily and MAGNET is configured to collect events throughout the networking stack in these tests, the results are representative of worst-case behavior. The performance impact is less if MAGNET is configured to only collect the events needed. For example, if we collect only socket events, the rate at which monitored events occur is reduced by 24.6%. The overhead is also reduced by a similar amount. Although the overhead is still 19.8% for a single processor, scientific applications with a balanced compute to communication ratio will see less overhead.

Clearly, excess processing capacity must exist for the effect of monitoring to be negligible. This is not surprising since extensive monitoring does perturb the system. The effect can be minimized by configuring MAGNET to export only the events necessary for handlers to perform their computations and by running multiple processors.

3.2 Data Path Latency

We now turn our attention to measuring the latency from when an event occurs until the time an application receives it. There are four components to the latency. The first is the time it takes MAGNET to register the event. The second is the time it takes `magnetd` to receive an event record from MAGNET. The third is the time it takes a handler to receive an event record from `magnetd`. The fourth is the time it takes to transmit results from the handler to the application.

It takes MAGNET an average of 371 ns to register an event on the test hosts. The next two components of latency depend upon the performance of `magnetd`, while the fourth is independent. We quantify `magnetd`'s performance by comparing the timestamp when an event record arrives with the timestamp when it was sent.

In the results that follow, the monitoring application is run on the same host as `magnetd`. Components of the distributed application use either Fast Ethernet or Gigabit Ethernet to communicate. The results are for runs which used Fast Ethernet except where noted.

Figure 4 shows the time it takes for `magnetd` to see an event exported by MAGNET. The latency repeatedly decreases to near zero indicating that `magnetd` processes events faster than they are generated. The latency is the smallest for the last event to arrive before `magnetd` sleeps and greatest for events which arrive immediately after `magnetd` goes to sleep. The linear shape of the curves confirms that `magnetd` removes events from the kernel buffer in FIFO order.

The average latency from MAGNET to `magnetd` is 4.01 ms. Since a timeslice under Linux is 10 ms,[3] `magnetd` is operating at 40% capacity in this test. Con-

[3]The spacing between lines in Figure 4 corresponds to a timeslice.

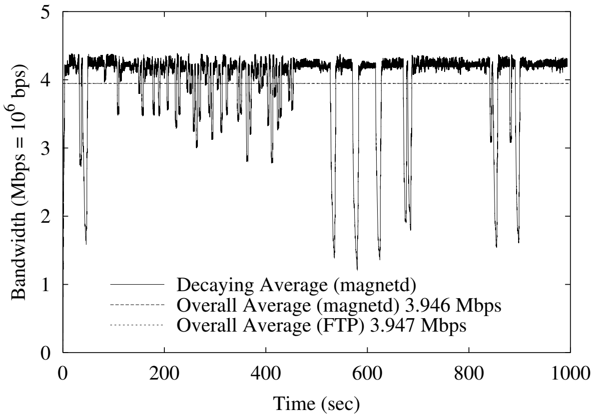

Figure 5. Verification of Bandwidth Handler Accuracy

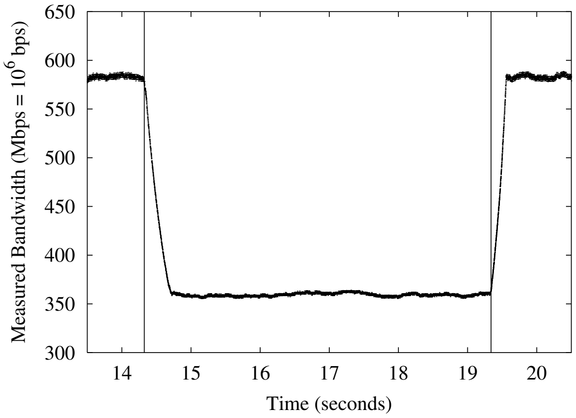

Figure 6. Bandwidth Handler for Two `netperf` Connections

figuring `magnetd` to poll for new events continuously reduces the average latency to 38 μs at the cost of greater CPU load.

We note that the latencies between MAGNET and `magnetd` are higher for Gigabit Ethernet than for Fast Ethernet (4.98 ms vs. 4.01 ms). This is caused by interrupt coalescing which increases the average latency substantially. The average latency from `magnetd` to a handler is also higher for Gigabit Ethernet than for Fast Ethernet (1.273 ms vs. 0.248 ms). The cause is still unknown.

In summary, the biggest cause of latency is the process scheduler. The average latency is less than 5 ms even for Gigabit Ethernet with interrupt coalescing. Thus, the latency should be acceptable for many monitoring tasks, particularly in computational grids, where round-trip times are on the order of 100 ms.

4 Handler Validation

The bandwidth handler supplied with MUSE computes the average bandwidth over the last N socket send events. In this section, we show that an accurate bandwidth can indeed be computed in a handler from an event stream. We validate the handler by comparing its results with the results obtained through independent means.

As a first check, we monitor an FTP application transferring a 467 MB file from ftp.debian.org and compare the average transfer rate reported by FTP with the overall average computed by the bandwidth handler. FTP reports an average transfer rate of 3.947 Mbps, where Mbps = 10^6 bits per second. The bandwidth handler reports an average transfer rate of 3.946 Mbps, a difference of 0.025%. Figure 5 shows the "instantaneous" transfer rate, windowed over $N = 1000$ events, along with the two average rates. Times when the network was congested are clearly visible.

The next check is also performed between the same two hosts as in Section 3. First, a `netperf` connection is started and allowed to reach streaming state. Next, another `netperf` connection is started 14.33 s after the first connection. Five seconds later, the second `netperf` is terminated. Figure 6 shows the computed bandwidth as a function of time. The starting and ending times of the second connection are delimited by vertical lines. The figure clearly shows that the bandwidth of the first connection falls off exponentially as the second connection goes through slow start. After about one third of a second, both connections reach steady state once more. Finally, the bandwidth of the first connection increases after the second connection has terminated until it reaches streaming state again. (The slight curvature in the line during linear increase comes from computing the average over the last N events.)

5 Applications of MUSE

MUSE provides many benefits to the parallel and distributed computing community. Its overhead is low enough that it can be used on-line for many tasks which used to be done off-line. This ability has many practical applications, some of which we discuss in this section.

5.1 Event Visualization in Distributed Systems

Distributed systems, by nature, are very complex with plenty of opportunities for subtle bugs and performance problems. The ability to visualize the execution history of a distributed application could potentially save large amounts of time and speed up the development and deployment of such applications. Although MUSE does not contain tools for visualization, it can easily serve as an data source for existing tools.

As a proof of concept, we have developed a translator from MAGNET event records to the Universal Log Message (ULM) format used by the NetLogger toolkit [9, 19, 20]. This allows one to use the NetLogger Visualization tool (NLV) to view an event stream graphically. The transla-

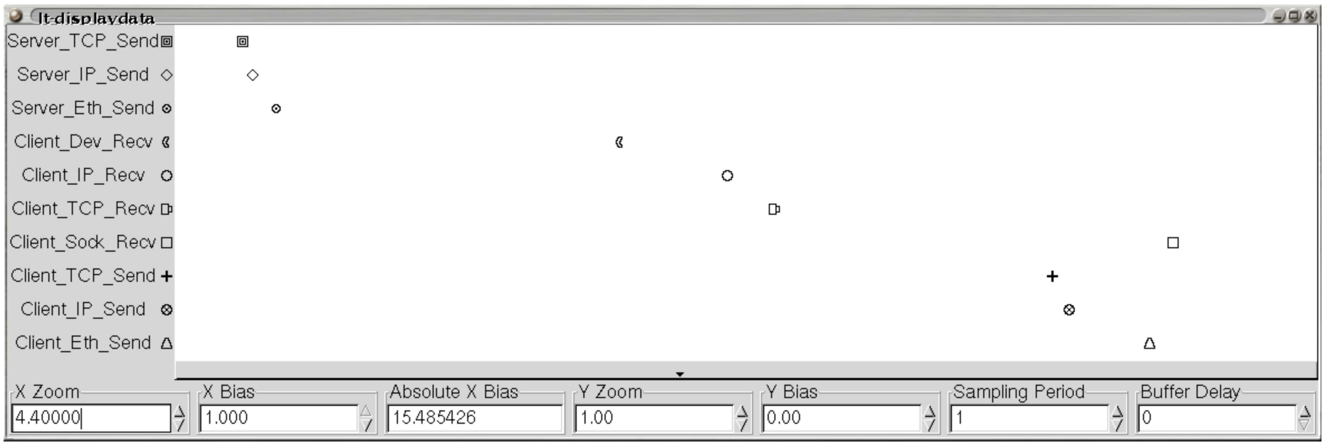

Figure 7. Visualizing the Transfer of a Packet in FTP

tor establishes a connection with `magnetd`, requests some subset of the event records, translates them to ULM format and appends them onto the end of a log file. NLV watches the tail of the file and updates the display as new events appear. Also as a proof of concept, we have developed a translator from MAGNET event records to the format expected by GScope [8], an open-source software oscilloscope library.

Bulk data transfers, via FTP or some other protocol, are an important operation in grid computing. In this test, a monitoring application connects to `magnetd` on the FTP server node using TCP/IP. It also connects, via TCP/IP, to `magnetd` on the FTP client node. Events are collected from the MAGNET daemons on both nodes, collated and displayed.

Figure 7 shows a GScope screen capture of a segment of a FTP transfer with MUSE monitoring both the client and the server. The first three data points show the server sending a packet down the network stack and out onto the network. The second three data points shows the same packet climbing up the network stack on the client. Of the remaining four points, the first three show an TCP acknowledgment being sent back to the server, while the fourth point shows the socket on the client receiving the packet that was sent.

From the figure, we can see that the packet traveled down the server's network stack at nearly constant speed, i.e., each layer took similar amounts of time to process the packet. On the client side, however, the packet sat in a buffer in the device driver until the IP layer was ready for it. The IP layer, on the other hand, passed the packet on to the TCP layer fairly quickly. Finally, a significant amount of time passed before the client actually read the packet from the socket.

With just a short look at the graph we are able to tell a lot about the behavior of the FTP transfer. We were also able to identify two places where time is potentially being wasted and where the transfer might be sped up.

5.2 Distributed Application Monitoring

One of the big challenges in developing a distributed application is acquiring insight into the operation of the application in order to debug, tune, monitor or control the application. Several frameworks, such as NetLogger [9, 19, 20], Autopilot [14, 15], Remos [2, 3] and CODE [17], have been proposed which accomplish these goals.

Each of these frameworks has some mechanism, called a *sensor*, for acquiring information about a distributed computation. Some, like NetLogger, require the application to be modified or relinked in order to collect information. Some obtain information about the behavior of the operating system through daemons such as `rstatd`, through the `/proc` file system [12, 18], through SNMP [2, 3], through active probing [2, 3] or through CPU hardware counters [4, 11].

MUSE is another way for these frameworks to obtain information. It makes available, in a convenient form, the wealth of extremely detailed information which MAGNET exports. It adds value because it provides information that the types of sensors discussed above do not and because it can filter and synthesize specific information needed by the framework. Furthermore, MUSE is able, through MAGNET, to export internal operating system variables which are important to understand the behavior of the system.

As described in Section 5.1, we have implemented a translator from MAGNET event records to ULM format so MAGNET events can be used in the NetLogger framework. We are working with the University of Illinois to integrate MUSE into the Autopilot framework.

5.3 Adaptive Applications

Adaptive applications are aware of the environment in which they execute and can adapt to changing conditions. Within the context of MUSE, an adaptive application enters into a dialog with `magnetd` to receive pertinent information about the current state of the system.

Going back to the distributed visualization example in Section 1, MUSE provides accurate bandwidth measurements that the renderer can use to reduce the frame rate or increase the compression ratio of the data.

We note that the two validation experiments in Section 4 are a first-order models of a distributed visualization application in which the network suffers congestion. Thus, Figure 6 is a graph of the bandwidth measurements which the renderer would use to decide what the frame rate or compression ratio should be.

6 Related Work

As discussed earlier, there are quite a few frameworks for debugging, tuning, monitoring and controlling distributed applications. Some examples are NetLogger [9, 19, 20], Autopilot [14, 15], Remos [2, 3] and CODE [17] Each of these has at least one way to obtain the information they act upon. For example, Autopilot obtains information by manually or automatically instrumenting the object code. Other tools, such as Supermon [12, 18] and Cluster Performance Monitor [1], collect operating system performance by exporting data through the `/proc` file system or by sending the data to a centralized monitor via the network.

Another distributed monitoring framework which deserves special mention is the Network Weather Service (NWS) [13, 21, 22]. It is a distributed monitoring system which periodically samples network bandwidth and latency, CPU utilization and available non-paged memory. It includes the ability to forecast resource availability.

MUSE, through MAGNET instrumentation, provides a different set of information than the sensors in the above tools. The information it provides is complementary. Since multiple sets of sensors may give a more complete view of the behavior of a distributed system, we intend to make MUSE compatible with those frameworks.

Another tool for fine-grained monitoring of applications is `pfmon` [4]. It is an IA-64/Linux-specific tool for CPU performance monitoring on Intel Itanium and Itanium II processors. Like MUSE, it provides fine-grained information without requiring applications to be modified or relinked. Also like MUSE, `pfmon` requires support to be compiled into the kernel. Unlike MUSE, the application to be monitored must be started from within `pfmon`. Furthermore, there are no provisions for `pfmon` to export events to remote hosts.

Finally, the tool most like MAGNET-MUSE is the Linux Trace Toolkit (LTT) [23]. Both tools originated at about the same time but with different initial purposes. Over time, the tools have evolved until now they are remarkably similar. The emphasis in LTT has been breadth of instrumentation, while the emphasis in MAGNET-MUSE has been depth. Both LTT and MAGNET-MUSE have low overhead, but it appears that MAGNET-MUSE is more efficient in certain circumstances.[4] Unlike MUSE, LTT has its own graphical event visualization tool. Merging the two open-source projects would leverage the strengths of each tool.

7 Future Work

MUSE is a very new tool and as such has plenty of room to grow. One area of future work is the development of handlers for `magnetd`. Some handlers will be general enough that they can be used in many different monitoring domains. For example, a handler which utilizes CPU performance monitoring hardware, like `pfmon` does, should be developed. Most handlers, however, will likely be very application specific and hence will probably be written along with the application which uses them.

So far, all the handlers have been designed to "pull" the data they require. For some tasks, it would be much more natural for `magnetd` to "push" the data to the application. Early in the life of `magnetd`, code existed to push data to NLV for display. In implementing the current handler mechanism, that code was removed. The code should be added back and an example of a push-style handler written.

One area which we have not addressed is security. Although the information exported from kernel space by MAGNET has no obvious security issues that we are aware of, there are likely to be several subtle ones. Unauthorized users should be prevented from accessing the MAGNET event stream. Right now, however, any user on the system can read the MAGNET device file. This problem can easily be solved by creating a "MAGNET" user or group and appropriately setting file permissions on the MAGNET device file used by `magnetd` to read events from the kernel.

Another way MUSE could be more secure is by authenticating clients before responding to requests. A Kerberos or PKI system for authentication could be added to `magnetd`. In certain contexts, such as when MUSE is used within a framework which provides authentication in the middleware, MUSE could rely on the security mechanisms of the framework.

We are collaborating with colleagues to develop the adaptive distributed visualization application alluded to in Section 5.3. According to the current design, the resolution of the frames generated by the renderer is adjusted based on the available bandwidth.

We have received feedback which suggests that it is impractical for some users to install MAGNET in order to use MUSE. Assuming that this would be the case, we are nearing completion of a user-space solution for monitoring applications without installing a MAGNET-ized kernel. This tool will become part of the MUSE toolkit. The MUSE and MAGNET toolkits are available from `http://www.lanl.gov/radiant/software.html`.

[4] The overhead for LTT is reported to be $< 2.5\%$ (Figure 3, configuration 6 of [23]) while the overhead of MAGNET alone is $< 0.8\%$ [6] under as similar conditions as we have tested. On SMP machines, the worst-case overhead of MAGNET-MUSE is $< 1.4\%$, while the worst-case overhead is $< 24.7\%$ on uniprocessor machines.

8 Conclusion

MUSE is a tool for applications to obtain information about the hosts on which they execute without the need to know the gory details of MAGNET. For many uses, it has sufficiently low overhead as to ensure very little perturbation of the phenomena being measured. We have shown that MUSE causes less than a 0.8% reduction in performance of the NAS IS benchmark if sufficient computing capacity exists.

Thanks to the fine level of detail provided by MAGNET, MUSE can synthesize information from MAGNET exported events with great accuracy. We have shown that the handler which synthesizes bandwidth from socket send events was within 0.025% of the bandwidth reported by FTP. By selectively processing MAGNET events, MUSE can tailor the level of detail for any task.

Finally, we gave examples of several ways in which one can use MUSE. The first was to visualize the events of a distributed application using NetLogger's NLV or GScope. The second illustrated how MUSE could provide the events needed by computing cluster and computational grid monitoring frameworks such as CODE [17], NetLogger [9, 19, 20] or Autopilot [14, 15]. The last example showed how an application can take advantage of the information provided by MUSE to become adaptable.

Acknowledgements

This work was funded in part by the U.S. Department of Energy Laboratory Directed Research program and the U.S. Department of Energy Office of Science.

References

[1] D. Anderson and J. Chase. Cluster performance monitor. `http://www.cs.duke.edu/~anderson/freebsd/cluster_mon/project.html`.

[2] A. 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 CMU-CS-97-194, Carnegie Mellon School of Computer Science, 1997.

[3] P. Dinda, T. Gross, R. Karrer, B. Lowekamp, N. Miller, P. Steenkiste, and D. Sutherland. The architecture of the Remos system. In *Proceedings of the 10th IEEE Symposium on High-Performance Distributed Computing (HPDC'01)*, Aug 2001.

[4] S. Eranian. The Perfmon Project. `http://www.hpl.hp.com/research/linux/perfmon/`.

[5] W. Feng, J. R. Hay, and M. K. Gardner. MAGNeT: Monitor for application-generated network traffic. In *Proceedings of the 10th International Conference on Computer Communication and Networking (IC3N'01)*, Oct 2001. (This paper gives details about an early implementation of MAGNET).

[6] M. K. Gardner, W. Feng, M. Broxton, A. Engelhart, and G. Hurwitz. MAGNET: A tool for debugging, analysis and reflection in computing systems. In *Proceedings of the 3rd IEEE/ACM International International Symposium on Cluster Computing and the Grid (CCGrid'2003)*, May 2003. (This paper gives details about the current implementation of MAGNET).

[7] The Globus Project. `http://www.globus.org/`.

[8] A. Goel. Gscope: A software oscilloscope library. `http://gscope.sourceforge.net`.

[9] D. Gunter, B. Tierney, B. Crowley, M. Holding, and J. Lee. NetLogger: A toolkit for distributed system performance analysis. In *Proceedings of the IEEE Mascots 2000 Conference (Mascots 2000)*, aug 2000.

[10] J. Hay, W. Feng, and M. K. Gardner. Capturing network traffic with a MAGNeT. In *Proceedings of the 5th Annual Linux Showcase and Conference (ALS'01)*, Nov 2001. (This paper gives details about an early implementation of MAGNET).

[11] J. Mellor-Crummey, R. J. Fowler, G. Marin, and N. Tallent. HPCView: A tool for top-down analysis of node performance. *Journal of Supercomputing*, 23:81–104, Aug 2002.

[12] R. Minnich and K. Reid. Supermon: High performance monitoring for linux clusters. In *Proceedings of the 5th Anual Linux Showcase and Conference*, Nov 2001.

[13] Network Weather Service. `http://nws.cs.ucsb.edu/`.

[14] R. L. Ribler, H. Simitci, and D. A. Reed. The Autopilot performance-directed adaptive control system. In *Future Generation Computer Systems, special issue on performance Data Mining*, sep 2001.

[15] R. L. Ribler, J. S. Vetter, H. Simitci, and D. A. Reed. Autopilot: Adaptive control of distributed applications. In *Proceedings of the 7th IEEE Symposium on High-Performance Distributed Computing (HPDC-7)*, Jul 1998.

[16] W. Saphir, R. V. der Wijngaart, A. Woo, and M. Yarrow. New implementations and results for the nas parallel benchmarks 2. `http://www.nas.nasa.gov/NAS/NPB/Specs/npb2\_report.ps`.

[17] W. Smith. A framework for control and observation in distributed environments. Technical Report NAS-01-006, NASA Advanced Supercomputing Division, NASA Ames Research Center, Jul 2001.

[18] M. Sottile and R. Minnich. Supermon: A high-speed cluster monitoring system. In *Proceedings of Cluster 2002*, Sep 2002.

[19] B. Tierney, D. Gunter, J. Becla, B. Jacobsen, and D. Quarrie. Using NetLogger for distributed systems performance analysis of the BaBar data analysis system. In *Proceedings of Computers in High Energy Physics 2000 (CHEP 2000)*, feb 2000.

[20] B. Tierney, W. Johnston, B. Crowley, G. Hoo, C. Brooks, and D. Gunter. The NetLogger methodology for high performance distributed systems performance analysis. In *Proceedings of IEEE the High Performance Distributed Computing Conference (HPDC-7)*, Jul 1998.

[21] R. Wolski. Dynamically forecasting network performance using the Network Weather Service. *Journal of Cluster Computing*, 1:119–132, Jan 1998.

[22] R. Wolski, N. Spring, and J. Hayes. The Network Weather Service: A distributed resource performance forecasting service for metacomputing. *Journal of Future Generation Computing Systems*, 15(5–6):757–768, Oct 1999.

[23] K. Yaghmour and M. R. Dagenais. Measuring and characterizing system behavior using kernel-level event logging. In *Proceedings of Usenix Annual Technical Conference 2000*, Jun 2000.