Large-scale Persistent Numerical Data Source Monitoring System Experiences

J. Brandt*, A. Gentile*, M. Showerman†, J. Enos†, J. Fullop† and G. Bauer†
*Sandia National Laboratories
Albuquerque, NM.
Email: (brandt|gentile)@sandia.gov
†National Center for Supercomputing Applications
Champaign, IL.
Email: (mshow|jenos|jfullop|gbauer)@ncsa.illinois.edu

***Abstract*—Issues of High Performance Computer (HPC) system diagnosis, automated system management, and resource-aware computing, are all dependent on high fidelity, system wide, persistent monitoring. Development and deployment of an effective persistent system wide monitoring service at large-scale presents a number of challenges, particularly when collecting data at the granularities needed to resolve features of interest and obtain early indication of significant events on the system.**

In this paper we provide experiences from our developments on and two-year deployment of our Lightweight Distributed Metric Service (LDMS) monitoring system on NCSA's 27,648 node Blue Waters system. We present monitoring related challenges and issues and their effects on the major functional components of general monitoring infrastructures and deployments: Data Sampling, Data Aggregation, Data Storage, Analysis Support, Operations, and Data Stewardship. Based on these experiences, we provide recommendations for effective development and deployment of HPC monitoring systems.

***Keywords*-resource management, resource monitoring**

I. Introduction

The operational environment of HPC systems continues to grow in complexity, becoming larger and incorporating a more diverse set of subsystems and support components. Issues of interest in system management have expanded beyond troubleshooting to encompass automated feedback loops to enable more efficient and cost effective operation. Such feedback loops can be diverse, including maintaining operation within a specified power envelope, topology and traffic aware resource scheduling and allocation, and enabling applications to be aware of and respond to impacts of congestion. While all of the examples cited are currently in research phase, they are all dependent on high fidelity, system wide, persistent monitoring.

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

This research is part of the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (award number ACI 1238993) and the state of Illinois. Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign and its National Center for Supercomputing Applications.

The National Center for Supercomputing Applications (NCSA) has been building an integrated infrastructure (ISC) [1] for monitoring and analysis of its Blue Waters [2] system. With its 27,648 nodes, Blue Waters is the largest Cray XE/XK platform in the world. The ISC utilizes both text and numeric data from the platform and support components including facilities and storage. While access to some numerical platform information is enabled by the vendor, a substantial amount of crucial information (e.g., High Speed Network (HSN) bandwidth and congestion data, Lustre traffic, and hardware performance counter data (e.g., flops, memory bus bandwidth utilzation)) is not. This can be a significant amount of data, particularly at the granularities needed to resolve features of interest and obtain early indication of significant events on the system.

Development and deployment of an effective persistent system wide data monitoring service at this scale has presented a number of challenges. Some of these were anticipated from the outset (e.g., nodes running data sampling and/or aggregation services may unexpectedly fail and the monitoring system must be robust to this). Others were completely unexpected (e.g., attempts to communicate with nodes in certain states could generate system level errors). The monitoring infrastructure we have developed, deployed, and modified to address problems as they emerged is the Lightweight Distributed Metric Service (LDMS) [3].

This paper documents our experiences, including problems addressed/encountered, in the development and deployment of effective monitoring tools and methodologies on NCSA's Blue Waters system. While some of the experiences presented here are platform specific, they are representative of the kinds of problems that can emerge when applying a generic monitoring system to any platform with specialized hardware. Thus, understanding the reasons for the artifacts, the ways in which they can adversely affect a system wide persistent monitoring system, and possible mitigating approaches can provide valuable insight to anyone attempting to apply such methods to new platforms. Additionally, these experiences are pertinent to system designers and integrators as they plan future platform architectures.

In particular we present monitoring related challenges and issues that we have encountered over the past 2 years and their effects on the major functional components of our monitoring infrastructure and deployment methodologies. We have categorized these as follows: Data Sampling, Aggregation, Storage, Analysis Support, Operations, and Data Stewardship, as well as Platform Specific issues that impact each of those. Each of these is subdivided into generic categories (e.g., Scale Issues, Usability Features, Environmental Issues) where appropriate.

This paper is organized as follows. In Section II we present background on Blue Waters and highlights of the LDMS architecture and deployment. Sections III through IX present our experiences as they impact the major components of the monitoring system. Recommendations, based on our experiences, are provided in Section X.

II. Background

Blue Waters is a Cray XE/XK system consisting of 27,648 hosts. Of these, there are 22,640 XE compute nodes with 2 AMD 6,276 Interlagos Processors and 64GB of RAM, 4,228 XK compute nodes with 1 Interlagos Processor, 32GB of RAM, and 1 NVIDIA GK110 (K20X) GPU processor. The balance of 780 service nodes utilize a single AMD Istanbul 6 core processor with 16GB of RAM. Some compute nodes may be repurposed as service nodes. All hosts utilize a Cray Gemini [4] interconnect configured in a 3D torus topology.

The vendor-supplied monitoring and operational support is based around the Cray System Management Workstation (SMW). The SMW is responsible for collecting log files from all hosts and System Environmental Data Collections (SEDC) [5] data via an out-of-band network from all platform components (e.g., hosts, blades, chassis).

There are, however, some important data sources which are not made readily available, such as HSN bandwidth and congestion data, Lustre traffic, and hardware performance counter data (e.g., flops, memory bus bandwidth utilization). For this reason, we have deployed an in-band monitoring system, LDMS, to provide system wide access to this data.

In this role, LDMS runs as a persistent system service, concurrent with applications and existing system software, and must be robust to platform node, network, and support component failures, reboots, or questionable production states. Architectural details of the LDMS architecture and installation can be found in [3]. Highlights and additional information about the operating and deployment environment necessary to enable correct and effective operation in our large scale HPC environment are provided below.

A. *LDMS Architecture and Configuration Highlights*

The Lightweight Distributed Metric Service (LDMS) is a plugin based infrastructure which utilizes a core daemon, *ldmsd*, to sample, transmit, and store numeric data. LDMS does not include large-scale analysis and visualization, although the data collected by LDMS is intended to be used by such tools.

The LDMS daemon functionality is defined by what plugins are loaded and configured. LDMS plugins are compiled C code. Thus, on-the-fly changes to the sample set are not possible beyond stop, start and sampling rate changes. Flexibility, if desired, of what metrics to collect must be coded to be configurable in the plugin configuration step. Sampler plugins support three functions which must be executed in order. These are *load*, *configure*, and *start/stop*.

The system configuration is shown in Figure 1 [3]. LDMS daemons run on each compute node with collection plugins. LDMS daemons configured for aggregation run on 4 service nodes. While the platform support for fan-in (Section IV-B) would enable aggregation from the whole system by only 2 aggregators, we use 4 with redundant connections for failover.

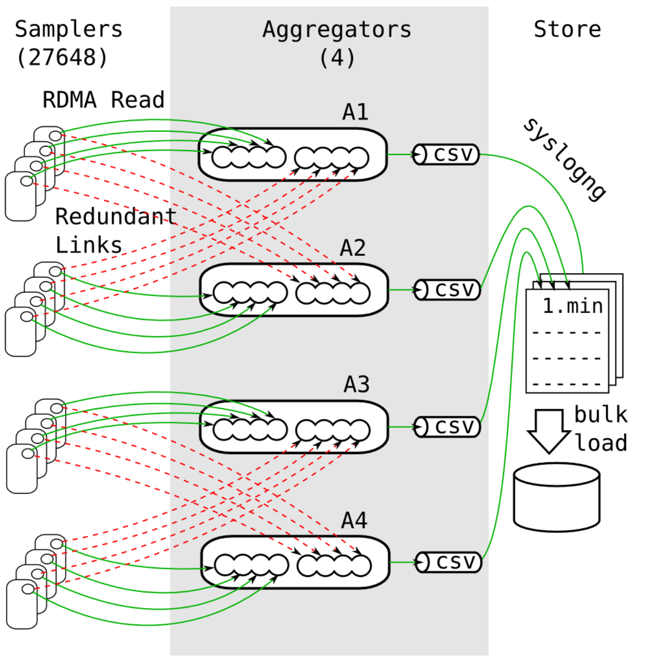

Figure 1. *Initial* LDMS configuration on Blue Waters (from [3]). LDMS collector daemons run on each node. LDMS aggregator daemons pull data from the collector daemons; these run on 4 service (non-compute) nodes. Redundant connections (dashed arrows) exist to each sampler *ldmsd* for fast failover capability. Initially, the aggregators each wrote a CSV file to a local named pipe; Data from this pipe was forwarded by syslog-ng to the ISC where it was bulk loaded into a database. Revisions to the storage configuration are discussed in this paper.

The LDMS aggregators use a pull model to periodically fetch data from the collectors. This was a design choice in order to minimize the impact and complexity of the samplers running on compute nodes and for better support of load balance and failover. In a pull model using the Remote Distributed Memory Access (RDMA) protocol, the compute nodes do not have to include functionality and the related overhead of sending messages and knowing and maintaining the locations of the aggregators and failover paths. With RDMA there is no additional compute node

CPU involvement in the data transport. We chose to place this additional complexity on the aggregators which run on service nodes. Additional CPU and/or memory overhead on the aggregators has no adverse affect on running applications. Reconfiguration of the overall collection topology in the event of an aggregator failure is done entirely at the aggregator level. In our initial deployment, rather than writing directly to local stable storage, the aggregators each wrote a CSV file to a local named pipe. Data from this pipe was forwarded by syslog-ng to the ISC where it was bulk loaded into a database. Due to observed inadequacies of this approach, revisions have since have been made to the storage configuration and capabilities. These are discussed in Sections V and VI.

B. Size Considerations

Cray XE/XK systems utilize a "diskless" model where the root image is pushed out to the platform hosts at boot time and resides in the hosts' memory. The Blue Waters image is about 400MB in size. Since this must be sent across the HSN to every host, maintaining a small boot image is essential to maintaining reasonable boot times.

Because we want to monitor host-related data of interest from the time each host boots, the monitoring programs, including support libraries, must be in the boot image. Thus an important consideration in the deployment of LDMS on Blue Waters was the additional size of the boot image.

Memory on a compute host is a shared and scarce resource, thus it is important to minimize the memory footprint of the monitoring infrastructure component that is deployed on the compute hosts. While the base *ldmsd* only occupies $\sim$2.3MB of space, there are also considerations to be taken into account with respect to data metric set sizes.

The current metric sets being collected from Blue Waters compute node hosts are comprised of 1.4KB data and 12KB of meta-data. The reason for the size difference is that the meta-data contains an 8 byte pointer and string labels per datum as well as various other information about the set itself. The data is typically just an 8 byte value per datum. Because of storage and processing constraints, the sampling is currently being performed at one minute intervals with corresponding pulls of the data by an aggregator.

As mentioned previously, CPU overhead and hence OS Noise on the compute nodes and network contention of monitor data traffic with user applications must also be kept as low as possible. OS Noise testing results are given in [3]. In this testing, no statistically significant adverse impact to application run-time was observed.

III. Numeric Data Sampling

In this section we present experiences with our sampling methodology, giving detail on what was successful, what was not, and how we addressed problems encountered. While some of these experiences are related to platform type and/or sampling mechanisms and methodologies, many are generally applicable and all are relevant to those undertaking a similar endeavor.

The Cray XE/XK platform provides a variety of interfaces for collection of pertinent numeric data. Among these are the `/proc` and `/sys` pseudo file systems, NVIDIA's `nvml` interface [6], and the Gemini network ASIC which are only accessible from the host (in-band).

A. Source Related

Cache and Clock Skew Effects: While using our data to analyze the characteristics of HSN congestion, we discovered cases in which we had multiple sets of data for a given time interval for some components. One series of redundant data had unchanging values. Root cause analysis showed that this would occur following a host clock reset performed due to significant forward clock skew (e.g., host clock ahead by more than a sample interval). In this case, the problem was that the timestamp that the kernel module (*gpcdr* [7]) supplying the HSN data used to determine if the cache time had expired was not also set back in time. The effect was that the cached values of the counters were used until the node's time exceeded the valid cache time. The time handling of cache in the kernel module has since been fixed by Cray. However significant clock skew will still cause erroneous analysis as we currently do not check data timestamps against aggregator clocks. Additionally clock reset after significant clock skew still has ramifications with respect to use of the apparent double data as there is no way to know which came from the erroneous vs. the reset time from a historic processing perspective (see Section IX).

Data Availability: Even with Cray providing data about bandwidth counts for network links, some pertinent data with respect to link bandwidth capacity was missing. We discovered, after conferring with Cray engineers, that the pertinent data was only available from the SMW and the file size for this data on Blue Waters was $\sim$ 40MB (or 10% of the image). Since this was deemed unacceptable, we developed an algorithm that exploited the regularity of the interconnect to device mapping to parse this file and extract and encode the pertinent information; the result required only 30KB and was well within our memory budget.

Data Source Component Failure: The LDMS design is robust to components failing, rebooting, etc. However, in general, within the samplers we chose to keep filehandles open when possible to avoid the overhead of opening and closing the filehandles each time. For sources such as in `/proc/meminfo`, the sources and the associated filehandle remain viable as long as the node is up. However, for some sources, such as file system mounts (e.g., Lustre), if the resource is unmounted the filehandle becomes invalid. For such sources, we had to implement a more robust error path in order to enable them to come and go and have the sampler re-establish the filehandle correctly.

When User Space and System Space Collide: There is a growing interest in system wide collection of hardware performance counters which have traditionally been used by application profilers to provide fine grained insight into application performance. We collect data (e.g., memory bandwidth utilization) from the Interlagos Model-Specific Registers (MSRs) [8]. Writing control registers defines which counters are being accumulated in the corresponding data registers.

Unlike the majority of our `/proc` data sources, these are dynamic sources, which may be modified by a user via application profiling tools. To handle this case, and give preferential use of these registers to the user we included additional functionality for checking if the counters have been reprogrammed prior to collection of each sample, and for changing and re-enabling the desired counters once a user job which reprogrammed our registers completes. We further implemented a command-line interface to be triggered by the epilog script to reset the counters to the desired value.

B. Environment

Boot time environment: The system administrators of Blue Waters wanted the monitoring system to come up as early as possible using an init script just after a node was booted. Because of this all dependencies for the monitoring infrastructure had to be included in the image. Interestingly, bash is not included in the system image. While this wasn't a show stopper, it was an inconvenience as we had written the init.d script in bash and there were some functionalities that would not work and had to be changed.

System Management Processes: The Out Of Memory (OOM) killer process is designed to kill non-system processes if the host memory utilization becomes too high. We had assumed that since we were running LDMS as a root process it would not be targeted by the OOM killer. However, over tens of thousands of nodes and a few days, we started having LDMS daemons unexpectedly die. Upon performing a root cause analysis it turned out that the OOM killer was killing our monitoring process even though it was owned by root and occupied <3MB of memory. To prevent this we set the `oom_score_adj` to -1000 for the *ldmsd* in the init script that starts it.

C. Platform Specific Considerations

In the Cray XE/XK environment, when performing small RDMA data transfers, a process must use a Fast Memory Access (FMA) descriptor. While there are only 256 FMAs available per node, a process can use a floating descriptor which a cursory read of the documentation indicates does not consume an FMA. However, upon a more careful read, this is only true within a process group. A particular application running on the Blue Waters machine was trying to use all 256 FMAs with the result being that if LDMS were running, then the application would fail and if the application were running, then LDMS could not be started. The resolution for this was to talk to the conflicting code user and have them change how they were utilizing FMAs. This resolved the problem but points out how engineering a system without planning for such a system service can cause resource problems when you suddenly need one. Thus encouraging vendors to provision future systems to natively host a system wide monitoring service that can utilize an RDMA transport and appropriate access protections would be of enormous benefit.

IV. Data Aggregation

This section presents our data aggregation methodology, impediments of scale, problems encountered, and solutions.

A. Scale Issues

Configuration Time: The first scale related issue we ran into in deployment of LDMS on Blue Waters was the time it initially took to configure an aggregator. The systems we had run LDMS on previous to deployment on Blue Waters were thousand node clusters which ran a first level aggregator for every 128 hosts. Our aggregator configuration init scripts executed seemingly instantaneously. However, when we first deployed on Blue Waters it took multiple minutes to issue 13,324 *add_host* commands to an LDMS aggregator. We solved the problem by streamlining the configuration related activities being performed in the init scripts and were able to reduce the time to ~45 seconds.

Component Death: The second scale related issue we encountered was collector thread starvation. LDMS originally had a single thread pool that supported both connection maintenance and periodic data collection from sampler LDMS daemons. On a system the size of Blue Waters it is possible to have thousands of nodes down concurrently. Thus, even setting the connection timeout to one second and retrying to establish a connection every 20 seconds resulted in all threads in the thread pool spending the majority of their time waiting on connection timeouts and very little in data collection when there were significant host outages. We solved this by creating a separate thread pool dedicated to connection setup and dedicating the previous thread pool solely to data collection.

B. Platform Specific Considerations

In this section we present problems encountered that are particular to the Cray XE/XK platform and the kernel version being run.

Inadequate Support for 3rd Party System Software: Cray never envisioned the possibility of a customer-owned system wide program running and communicating across their Gemini HSN fabrics other than user applications. An associated expectation was that a failure in a node included in an application resource pool would trigger application failure and preclude further communication with the node until it was repaired and rebooted. Since our system wide monitoring application runs as root and aggregators periodically poll sampler daemons running on compute nodes for information, this expectation is no longer valid. We discovered that communication with a compute node in the process of booting can trigger annoying *"b2b"* warning messages in system log files. While we were at first inclined to just ignore these messages as there was no evidence of harmful effects, we were informed by Cray that in their follow-on Aries network router chip the effects can be much more catastrophic as network quiesce events will result. The fix for this was to utilize a Cray library and query for node state before each communication with a host. If a node is not in an up state then we don't attempt any communication with that node.

Constraints on the Architecture: When initially testing fan-in ratios (i.e., how many sampler hosts an aggregator can aggregate from) we were unable to go above 16,000:1 even though we had set our open file descriptor limit to be unlimited. We were told by Cray that this was a known bug/limitation of the particular kernel they were running and that newer kernels had a patch for it. The fix was to utilize four daemons to support the system (with redundancy) instead of two.

C. Usability Features

Timing: We originally envisioned that the data collection on each node would occur independently. Because our target collection frequency was 1Hz, We reasoned that a few lost data points here and there would be inconsequential. When we started collaborating with NCSA staff on LDMS deployment on Blue Waters, their target sampling rate was once a minute. Figure 2 (top) shows a set of one minute collection samples across 10,000 nodes of Blue Waters. The y-axis in the figure indicates a unique node id, the x-axis indicates time. Due to staggering in the start time of each *ldmsd* on each node, a "wave-like" structure is seen in the sample times. What became readily apparent was that analysis of the data would be impossible as there was no time coherence across the resources. From an analysis perspective, we required a coherent system "snapshot". for the data to be of much use. Thus we implemented a synchronized collection mechanism whereby all nodes would attempt to collect data at as close to the same time relative to their own system clocks as possible. With synchronized collection implemented, Figure 2(bottom) [9], all nodes' *ldmsd*'s collect at the same time (empirically within 4ms) once the processes have been started. (There is data for a shorter time range in the top figure, however the data for both figures has been plotted on the same range to enable visual comparison.)

Error Diagnosis: Due to the number of hosts on Blue Waters, sampler *ldmsd*'s are run in *quiet* mode (i.e., no log

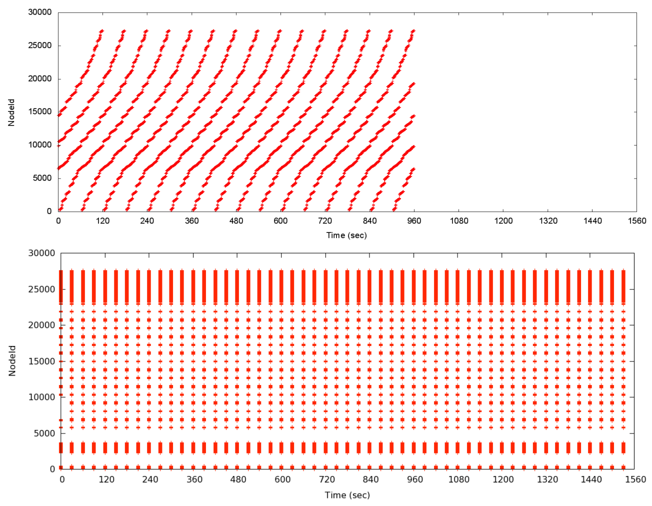

Figure 2. Collection occurrences on each node (y-axis) through time (x-axis) Top: unsynchronized collection results in varying collection times for the *ldmsd*'s on each node. Bottom: with synchronization, *ldmsd*'s are scheduled to collect at the same time on all nodes (relative to their own system clock), enabling analysis of coherent system "snapshots".

file output) to preclude the possibility of widespread errors causing a glut of log messages on the SMW or filling up host memory. Due to the additional complexity of aggregator operation, we want to be able to diagnose problems when they arise. Thus they are run with logging turned on. Early in our deployment of LDMS we had included extensive instrumentation at the *debug* level in order to ensure things were running properly. The log level defined when starting an *ldmsd* could not be modified without restarting the daemon. However, due to the large number of connections, and logging of even normal activity about them, we experienced unacceptable log file growth. To minimize log file growth while still enabling troubleshooting we incorporated the ability to perform dynamic log level modification. We now normally run with logging set to the *critical* level (only log issues resulting in daemon failure) and dynamically change to *debug* (log everything that is instrumented) or another appropriate level when problems arise.

V. Data Storage Methodologies and Considerations

In this section we present our experiences with Data Storage. Note that storage includes both active storage in support of run time analysis as well as longer term storage for historical analyses and records.

A. Scale Issues

Database Insertion Implications: Sandia's previous database schemas supporting monitoring data on smaller systems (e.g., ~1000's nodes) used one table per metric in order to flexibly support multi-metric analyses [10]. Therefore, in LDMS we initially implemented a data storage system that could write out a single file per-metric and/or insert single or multiple values of individual metrics into a database.

On Blue Waters, due to the volume of data, which was initially 55GB/day, we found it to be more practical to write the data to a file and then bulk load the file into the database. In order to do this as a single file load the database schema must then be one table with all associated metrics defined (including NULL defines) in the file. We had initially implemented a separate collector per data source. In order to support the file load, we modified the collector to produce a single metric set for the system, regardless of source (e.g., the single metric set contains all the HSN, GPU, CPU load, etc, data). Further, we modified the store to write a single file per metric set (this has additional benefits for the timestamp, discussed in Section VI-A).

However, there was an additional ramification to this approach. Not all nodes produce the same data, for example not all Lustre filesystems are mounted everywhere; there are some different metrics for service and compute nodes; and only about $1/7$ of the system produce GPU-based metrics. For efficiency's sake we have found it expedient to implement a consistent data set for all nodes, with the ability to specify non-existent data sources for a host which then are populated with 0-valued data. This is an insignificant additional amount of data to transport relative to the overall HSN and storage bandwidth. However, as our desire for new sources and rates increases, and the storage necessary to support non-existent data sources correspondingly increases, this priority may change.

ISC Database: One major goal of the monitoring system is to support run time operational analysis (in addition to supporting longer-term historical analysis and retaining a comprehensive data archive).

For many of the operational questions we focus on either job-centric time series or a drill down on full system behavior to isolate the cause of unique features. For these analyses, the monitoring data needs close integration with run time job and log data and rarely involves consideration for very long time series data. In order to support a variety of datatypes and support a wide variety of queries, the ISC utilizes a relational database. This has limits in fast response time for large datasets and is impractical for storage of the entire dataset. We have settled on storing a few days of data for run time queries. Further, we reduce the latency to meaningful analysis by transforming the raw data into rate or difference information before being stored. (Features that we have implemented within LDMS to support computing derived data and considerations with respect to processing location are described in Section VI.) This type of storage and retrieval targets answers in the seconds to multiple minute timeframe.

Forwarding Data to Storage: The full metric set without MSR data is about 55 GB/day. Writing that to syslog-ng [11] over a socket resulted in data losses, despite significant configuration tuning. We have identified a subset of derived metrics as those of greater first-order interest, and we write only this derived data, which is about 15 GB/day, to syslog-ng for insertion into the ISC database. This has eliminated the transfer-related data loss.

The full dataset is written in a separate stream via syslog-ng to a mounted Lustre filesystem. Since there are 4 aggregators, there are 4 separate output files, which are combined via a cron job at the end of the day. We intend for the MSR data, which is 93 GB/day to be handled similarly. The full dataset is currently written to the Lustre filesystem. We are currently identifying the first-order derived data and functions of interest, such as a subset that enables computation of full machine and per-job flop rates, to be stored in the ISC database.

For the full dataset, a database, particularly a relational database, is not appropriate for long-term storage. Binary flatfiles may be useful for this, however without a reasonable investment in a binary format with guaranteed long-term support, text based CSV is still the safest, universal option. For these, we have found that human-readability is less of a concern, since they are typically processed by code, and reducing filesize by removing extra whitespace (such as after commas) is desirable.

VI. Numeric Data Analysis Support

LDMS does not inherently include numerical analysis capabilities, however we intend that our data collection framework support the needs of the data analysis.

We had several initial design philosophies that had implications for the support of analysis. We initially envisioned that the sampling could occur as fast as would result in no adverse impact and that occasionally missed data samples would be immaterial. We further believed that high fidelity resolution (e.g., microsecond resolution) in the timestamps was desired, since the sample time was of order of 100 microseconds per set. Finally, in order to minimize the processing on the compute nodes, we transported all raw data, with the intent that all analysis would be done later in a pipeline approach and/or off-cluster.

As we collected data and performed analysis, we realized that for our data size our original philosophy put too much burden on the backend analysis and storage components. In this section, we discuss enhancements that we made to the infrastructure in order to better support analysis.

A. Scale Issues

Timestamp Accuracy and Resolution: As described in Section V-A, we initially implemented a per-metric focused data storage system. This not only had poor insertion rates into a relational database, but also required discovery of concurrent samples in either multiple files or multiple database tables, which was time intensive for large systems.

In practice we found post-processing to be greatly simplified by grouping all concurrent data samples into a single set with a single timestamp independent of source (e.g, network counter data and `/proc/meminfo` data all get put into one LDMS set). The time between the first and last sample of our current set is ~0.5ms or ~0.0008% of our one minute snapshot time. We consider this difference negligible.

Also, though we retain the high resolution `timeval` time stamp, we store the second and microsecond parts as unsigned integer values as well. This enables faster searches for times based on comparison with the integer value for seconds, rather than searching for values which include microseconds within a time range.

Processing Location Tradeoffs: Our initial approach was to collect and store only the raw data. This has the least impact on node performance and provides the most flexibility for analysis. However, computations using the raw data require multiple queries to get the data of interest and may not produce timely results especially with large data volumes. Analysis on rates of changes of raw counter data, for example, require multiple passes in post-processing. In order to enable lower latency to results, we insert a subset of derived data into the ISC database (Section V-A). We implemented a derived store plugin for the aggregator, which performs derived computations on the data *in transit* The derived store supports simple functional forms, where we capitalize on the fact that the store can most easily retain the previous timestamp's data for rate calculations and that computing it on the aggregator means that we do not negatively impact compute nodes for processing.

Here we summarize processing tradeoffs with respect to *in situ* vs. *in transit* vs. post-processing. Post-processing gives the greatest flexibility, but may be hardest to implement, since it may require multiple passes through the data, and therefore have the highest latency to solution. *In situ*, which in our case is at the point of collection, incurs CPU overhead which takes away from useful cycles on compute hosts. In addition, while such filtering can reduce the amount of data to handle at the end point, it is at the expense of losing data which might prove to be useful later. The savings in network bandwidth isn't necessarily significant, since the required bandwidth is very small relative to the total available for the HSN links. *In transit* simple analysis is very useful for calculating simple forms and also splitting off some data for more immediate processing. We support both the *in transit* filtering and post-processing on the full dataset.

Long-term Analyses: Doing any system based analysis that may span many days or years requires the processing of large volumes of data. A recent calculation required processing in excess of 60 TB of numeric data (in conjunction with additional job and log data).

For long term analysis, the full dataset with raw counter data is stored in a large parallel filesystem. For answering questions that are not immediate, this is more practical because we can split the access and processing of the data over many compute nodes. The types of questions we solve with file archive data are long term memory trends by application or code team, and investigations of system behavior as a function of job location within the system. Parallel analysis and data reduction tools using the compute system itself is effective for answering queries that produce results in the hours to days timeframe.

B. Usability Features

Functional Forms and Resolution: In transit implementation of a small subset of functions can reduce the latency to getting usable data. In most cases, the main functional forms we have to represent are: raw, rates, sums, ratios, threshold comparisons. These may be for a given component, or across the set of cores of a node, or across nodes in a job. Currently, we perform rate calculations and apply scale factors to these and the raw data within LDMS store plugins. Additional analyses are performed using the database or flat files.

Note that writing a general analysis library even for these small set of functions is an endeavor unto itself. Computations have to be able to resolve fidelity of interest and overflow and rollover have to be handled. As we primarily store data in the database in integer form for efficiency, some rate data requires multipliers to avoid roundoff loss. Initially we failed to implement a multiplier for low rate events such as file open/close/seek events and thus recorded those as zero valued occurrences.

VII. Cray XE/XK Specific Issues

In this section we discuss issues not addressed in other sections that are specific to the target Cray XE/XK platform and not HPC in general.

Inadequate Support for 3rd Party System Software: Protection Domains [12] are the mechanism Cray has implemented on their platforms to ensure that one user cannot gain access to another user's data. In practice, the Application Level Placement Scheduler (ALPS) entity allocates resources to an application and also assigns to it an identifier pair, called a "pTag" and a "cookie"; processes are prevented from communicating outside their "pTag" domain. This pair is set up in the user's program environment and on every resource allocated to a job. When the job finishes the pTag is reclaimed by the system.

Since LDMS was to be run by root as a system service, we were initially told by Cray that we needed to set up a "system" pTag that we could then have dedicated to the LDMS application. While this seemed to work in our first testing, we found that when we started some testing just after a system reboot we were not able to run using our system pTag. Because of a bug in how system pTags are managed by the system and in particular, after a boot, a "system" pTag may not be recognized any longer. The fix was to generate a "user" pTag from a privileged account (e.g., system administrator) to be used instead which can then never also be used by a standard user.

Quiesce Events: After we had been collecting data for some months, during an analysis we discovered that some data points timestamps were tens of seconds later than expected. Upon discussion with Cray engineers we found out that in periods of high HSN congestion the network, or portions thereof, can be issued a "quiesce" which prevents any traffic from being injected into the network. In such cases, if a processor tries to send data it will also be quiesced and LDMS sampling will stop until the congestion has passed. While this explained our displaced data points, there is no fix or workaround for this behavior.

VIII. Operations

In this section we discuss some issues in HPC operations that impact support for a persistent monitoring service.

Agile Changes in a Static Environment: Cray XE/XK systems utilize a "diskless" model where the root image is pushed out to the compute node hosts at boot time and resides in the hosts memory.

While there was a well developed set of requirements at the outset of this deployment, initial testing and deployment success rapidly resulted in additional desired enhancements. Those that came up during the testing phase (e.g., synchronized sampling) were incorporated into the initial deployment image. However, subsequent enhancements requiring modifications to the in-image code could take months to be included into a new image and for that image to be booted in production. In contrast, changes to the aggregator, which were largely additional functionality to support data management (Section VI) could be installed more quickly, since they involved service and not compute nodes.

Our approach to this has been to test changes to our aggregation code and deploy as soon as it is tested. Initially, this put an additional burden on the design as we had to be prepared to guarantee interaction of mixed-versions of code. Our current approach is to, after testing and validation on a smaller test system, push the new sampler code to a directory in `/tmp` on all hosts. We then stop all the samplers being run out of the image and start them running the new code. This accomplishes two things: 1) if problems of scale arise, it is simple to roll back and 2) the code stays in sync between host samplers and aggregators. The new validated code is then rolled into the working image at the next opportunity.

Sampler Configuration: For a particular data source one may not know *apriori* what data will eventually be wanted. For the hardware performance counters, we started off with an expected set of counters of interest and, for implementation ease, initially hardwired those into the code. We quickly realized that we wanted a more flexible way to investigate which counters to collect.

Although code changes for different counters were simple to implement, instantiating code changes in our environment is non-trivial (Section VIII). Thus we re-implemented the sampler to take counter specification and associated register address data from a configuration file and expanded the sampler plugin command set to support dynamic changes to the counters being collected. More configuration files and options incurs complexity but the overall flexibility has provided substantial benefits.

IX. Data Stewardship

In this section we discuss the complexities of long term maintenance data from an evolving system.

While we print out a header with the output data (either in the same file or as separate files), we currently have built no facility for recording meta-data that would allow us to track changes in the system, the datasource, or our collection implementation that might be relevant. For example, data from before the fix for the cached *gpcdr* data values (Section III-A), has different and erroneous reporting of values of which a user seeking to compare these values should be aware. Units of a metric have changed over time. For example, the change in scale factor in order to resolve low rate events such as file open/close/seek events (Section VI-B), resulted in a discontinuity in the meaning of the values as they are now stored with respect to the multiplier.

System operating changes have occurred that have ramifications on later interpretation of the data. For instance, a topologically-aware scheduling algorithm was instantiated on Blue Waters over various periods of time during the course of this work. The goal of that allocation assignment was to alleviate cross-application congestion in the HSN; thus characterizations drawn from the data during these different time periods are expected to be different and the data should not be blindly combined.

In general we seek to develop an approach for defining metadata and/or quality metrics for raw and derived data. This also requires an implementation for tagging both at the time of collection as well as afterwards for later discovered events and a means of propagating them into subsequent analyses.

Currently, in the *in transit* derived store, where rates and deltas between timesteps are the primary analyses, we have implemented an additional flag in the output to indicate when the change in time is either negative or greater than some user specified value. The latter is to account for lost data values and resulting delta values that may be greater than anticipated for the expected collection frequency. This flag also helps us to discover missed collections.

Figure 3 is a heatmap based on the collected data from early in our deployment. Missing data points here are explictly colored as blue. Here missing data is primarily due to data loss while attempting to send the entire data set through syslog-ng to the database. We resolved this issue by forwarding only the derived data to the database via syslog (Section V-A).

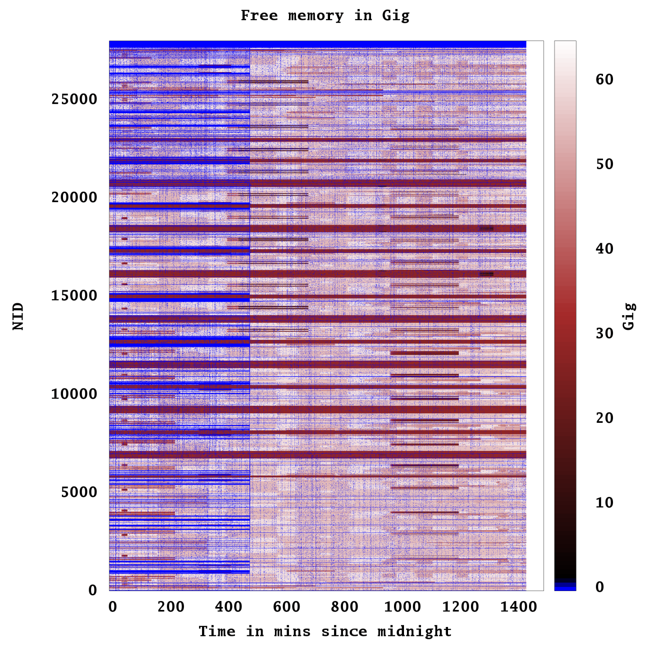

Figure 3. Heatmap based on the collected data early in our deployment process. Values for component id (y axis) through time are shown. Missing data here is colored in blue and is principally due to data loss while attempting to send the entire set through syslog-ng to the database.

We have had periods where storage of data for a subset of the system had failed after a system reboot. As previously mentioned, data loss has occurred prior to when the daemon was excluded from OOM killer actions (Section III-B) and when the result of network quiesce events on the processor prevented data from being collected at a given wall time (Section VII). Finally, we have had missing data when a system configuration changed, for example, after an upgrade to the `nvidia-ml` library which resulted in the failure to collect GPU data properly.

Missing data may go undiscovered if there is not an immediate need for the data. Even while the data is undergoing analysis, missing data might be overlooked when analysis results are an aggregation of values across many nodes, as occurs for metrics across large jobs or the full machine. We handle this in part by trying to characterize the data quality within plots. A simple quality indicator we use is including a count of datapoints used in each aggregated value.

We have found that it is important to document as early as possible anomalies within our long term data. Simple examples include dates and times that the file formats change as the collection evolves. This can include changes in data indicies and addition or removal of metrics. Such changes make it challenging to navigate through seemingly simple requests to describe long term trends when the exact meaning of the data transforms.

X. Recommendations

In the above we have presented some particular challenges in the development and deployment of an effective persistent system-wide monitoring service at scale. While some of those are particular to the platform, they each impact functional components of a monitoring system in general and serve as insights into the large-scale HPC experience.

In this section, based on our experiences, we present recommendations for effective development of a monitoring system (beyond the obvious implementation details).

- Numeric Data Sampling
  - Widely varying sources of data with different underlying acquisition mechanisms can imply different CPU overhead properties – always measure the impact.
  - One cannot infer impact on actual large scale applications based on CPU overhead as measured by benchmark codes such as PSNAP.
  - It is necessary to understand caching policies and mechanisms.
  - Look for possibilities of collisions between monitoring service and users utilizing the same mechanisms.
  - Just because it isn't documented doesn't mean there isn't an interface. Connect with the engineers.
  - There are persistency issues beyond the obvious. The optimization of keeping a pseudo file system handle open and re-reading didn't work when it was Lustre stats for a file system that got unmounted and re-mounted.
  - There is always the need for more flexibility as the system changes and heterogeneity increases. Where possible, utilize configuration files with reasonable defaults. Flexible configuration options are invaluable, even if they require more up-front understanding and time to code.
- Data Aggregation
  - Even low probability events do occur over a large enough collection of components.
  - Pay attention to use of timeout mechanisms. At large enough scale all resources can be tied up in timeout and incur resource starvation with respect to operational resources.
  - Configuration via shell script, at large scale, can be a non-starter. On Blue Waters it took about a minute to add 14,000 hosts to an aggregator via shell script.
  - Utilizing resources in a non-standard way can create unintended consequences. Cray did not envision a continuous system service that maintained long lived connections to all components. Their assumption that only user applications would utilize the HSN fabric required painful workarounds for

a monitoring system.
  - You will always want more data with respect to both number of metrics and complexity. Estimate your architecture requirements with this in mind.
- Data Storage
  - A single data set across a heterogeneous system has some advantages for processing but doesn't scale with data disparity (lots of storage space wasted on null data).
  - Transformation of data before storing in a database is good for rapid processing but should be done at aggregation points to minimize CPU footprint on compute nodes.
  - Processing data differences per time interval in a database for large data is a mistake.
  - Use multiple storage methodologies that support the way the data is intended to be used: Utilize tiered storage with well-known data manipulations for fast access, short term data vs. full, raw long term data; prioritize storage space over human-readability (e.g, no blank space after a comma) for data that will be mostly processed by code.
- Numeric Data Analysis Support
  - As the variety and volume of data increases, so does the need for streaming processing to provide functional combinations of data that make it immediately useful for insight.
  - Single time attribution for all data on a node is a good idea for searchability and fast "snapshot" processing.
- Platform Specific Issues
  - There may be obscure or wrong documentation as to how mechanisms work – in this case the security mechanisms provided by the protection domains.
  - There may be proprietary mechanisms which induce issues that cannot be overcome – e.g., quiescing network traffic in a high congestion situation and the resultant impact on the processor.
- Operations
  - Design configuration and installation methodologies that can work within the HPC operational environment, which may not support agile changes.
  - Beware port scan/network garbage issues and how they might affect a monitoring system listening for a connection with no enforcement of appropriate handshake protocols.
- Data Stewardship
  - Data of interest, including name to source bindings, will change over time causing changes in metric names and sets. Relatedly, there will be changes to an implementation or a data source or a platform that will need to be known about in subsequent short and long-term analysis attempts. A methodology and implementation are needed for data to metadata binding and propagating that binding through all phases of the collection, storage, analysis, and visualization process.
  - Monitoring data should be treated as long term resource in the same way that application code results are, with intended long-term and post-processing analysis of interest. There is increasing interest both for more complex analysis by system administrators and by researchers for this data. Long term format and storage support is necessary.

References

[1] B. Semeraro, R. Sisneros, J. Fullop, and G. Bauer, "It Takes a Village: Monitoring the Blue Waters Supercomputer," in *1st Wrk. on Monitoring and Analysis for High Performance Computing Systems Plus Applications (HPCMASPA) Proc. IEEE Int'l Conf. on Cluster Computing (CLUSTER)*, 2014.

[2] "Blue Waters." [Online]. Available: https://bluewaters.ncsa.illinois.edu

[3] A. Agelastos *et al.*, "Lightweight Distributed Metric Service: A Scalable Infrastructure for Continuous Monitoring of Large Scale Computing Systems and Applications," in *Proc. Int'l Conf. for High Performance Storage, Networking, and Analysis (SC)*, 2014.

[4] R. Alverson, D. Roweth, and L. Kaplan, "The Gemini System Interconnect," in *Proc. 2010 IEEE 18th Annual Symposium on High Performance Interconnects (HOTI)*, 2010.

[5] Cray Inc., "Using and Configuring System Environment Data Collections (SEDC)," Cray Doc S-2491-7001, 2012.

[6] NVIDIA, "NVIDIA Management Library (NVML)," https://developer.nvidia.com/nvidia-management-library-nvml.

[7] Cray Inc., "Managing System Software for the Cray Linux Environment," Cray Doc S-2393-5202axx, 2014.

[8] Advanced Micro Devices (AMD), "BIOS and Kernel Developer's Guide (BKDG) for AMD Family 15h Models 00h-0Fh Processors," Technical report 42301 Rev 3.14, January 2013.

[9] M. Showerman *et al.*, "Large Scale System Monitoring and Analysis on Blue Waters using OVIS," in *Proc. Cray User's Group*, 2014.

[10] J. Brandt *et al.*, "OVIS-2: A Robust Distributed Architecture for Scalable RAS," in *4th Wrk. on System Management Techniques, Processes, and Services (SMTPS) Proc. IEEE Int'l. Parallel and Distributed Processing Symposium (IPDPS)*, 2008.

[11] "The Foundation of Log Management (syslog-ng)." [Online]. Available: https://www.balabit.com/network-security/syslog-ng

[12] Cray Inc., "Using the GNI and DMAPP APIs," Cray Doc S-2446-4003, 2012.