SAND2023-03836C
LLNL-CONF-847852

Evaluating and Influencing Extreme-Scale Monitoring Implementations

Andrew Barry
Hewlett Packard Enterprises
St. Paul, MN, USA
andrew.barry@hpe.com

Jim Brandt
Sandia National Laboratories
Albuquerque, NM and Livermore, CA, USA
brandt@sandia.gov

Ann Gentile
Sandia National Laboratories
Albuquerque, NM and Livermore, CA, USA
gentile@sandia.gov

Christopher J. Morrone
Lawrence Livermore National Laboratory
Livermore, CA, USA
morrone2@llnl.gov

Eric Roman
Lawrence Berkeley National Laboratory
Berkeley, CA, USA
eroman@lbl.gov

Alec Scott
Lawrence Livermore National Laboratory
Livermore, CA, USA
scott112@llnl.gov

Kathleen Shoga
Lawrence Livermore National Laboratory
Livermore, CA, USA
shoga1@llnl.gov

Tom Tucker
Open Grid Computing
Austin, TX, USA
tom@ugc.us

***Abstract*—Over the past decade HPC practitioners have been able to gain new insights into application resource utilization and to detect and diagnose problems with decreased latency through fine-grained monitoring of our HPC systems while incurring no statistically significant performance penalty.**

The HPE Cray EX community is exploring a variety of tools for telemetry data acquisition with two major monitoring directions: a) Cray implementation of monitoring using CSM and b) customer designed/specified system software, including monitoring. Both include the Lightweight Distributed Metric Service (LDMS) for high-fidelity, high-volume node-level data collection as well as for other features such as dynamically modifiable data collection rates and integration of both synchronous and event-driven data. LDMS is Linux distribution agnostic and is utilized across a variety of OSs in both bare metal and containerized environments.

In this collaboration of HPE and user sites, we have begun to explore these two approaches on early-availability platforms at NERSC and LLNL. We seek to ensure that LDMS directions continue to support the intended diversity of approaches and that user contributions to directions and code continue to serve the greater community. Further, we seek to educate sites on configuration, deployment features and scalability requirements for extreme-scale systems and run time analytics.

***Index Terms*—LDMS; Monitoring; Slingshot; Configuration**

I. Introduction

High Performance Computing (HPC) systems have long been operated with minimal exposure of system and application telemetry data to system administration staff and the HPC users. This has been, in large part, due to the fear of negative system and application performance impacts. Over the past decade we have proven that we can expose reasonably large quantities of such data on a fine-grained cadence (order of 1 second) with no statistically significant adverse effects on system or application performance (e.g., [1]). What we get in return are new insights into how applications are utilizing system resources and the ability to detect and diagnose problems with much lower latency than previously possible (e.g., [2]). The Lightweight Distributed Metric Service (LDMS) [1], [3] is widely used for HPC monitoring on Linux-based systems because of its ability to collect data at large scale and at the fidelity necessary to resolve features of interest. Additionally, LDMS supports dynamic modification of data collection rates and integration of both synchronous and event-driven data. LDMS was designed for both stand-alone use and flexible integration with other monitoring technologies.

In the HPE Cray EX community, the software stacks, and hence the monitoring installation, configuration, and deployment are going in three major directions: a) the HPE-provided Cray System Management (CSM [4]) stack which is a prior Cray implementation with highly-tuned system software, including monitoring and compilers, b) the HPE-provided HPCM stack which is a version of the HPE Performance Cluster Manager [5] that includes provisioning, management, and monitoring capabilities, and c) customer designed/specified system software, which includes the monitoring system and its installation and configuration. With Perlmutter (see, for example, [6]), NERSC takes the first approach and with El Capitan (see, for example, [7]) LLNL takes the third approach. In this paper, we therefore focus on approaches a and c. In both cases, issues of scale in collection, transport, analysis, and management of monitoring data must be addressed. Both NERSC and LLNL utilize LDMS as their node-level monitoring system.

In this collaboration of HPE and user sites, we begin to explore commonalities and differences in configuration and deployment of LDMS as well as how the data is exchanged, stored and utilized. LDMS has pervasive considerations because it is used for high-fidelity, high-volume data collection which must be supported by back-end storage technologies which in turn must support simultaneous data ingest and queries for analyses. Identification of similarities and differ-

ences in deployment configurations and data use will ensure that LDMS continues to support the intended diversity of approaches and that user contributions to direction and implementation continue to serve the greater community. Our goal is to get early insight into requirements that will ensure continued coherent community development of LDMS components and their interfaces with site-specific components. Further, we seek to educate sites on configuration, deployment features, and scalability requirements for extreme-scale systems and run time analyses.

II. LDMS Background

LDMS is a daemon-based framework designed specifically for High Performance Computing as a low-overhead mechanism for collection, transport, and storage of monitoring data from arbitrary components on intervals small enough to resolve operational attributes of interest. This framework enables periodic collection of time-series system data, on regular intervals, that can provide insight into the state of the system at any collection point in time. LDMS daemons, called *ldmsd*, can provide capabilities for one or more functionalities of data collection, transport, and storage. A daemon's *plugins* and their configurations determine its functionality. Daemons on the compute nodes typically run sampler plugins and are referred to as *sampler* daemons. Transport is achieved via multi-hop transmission and/or *aggregation*. A typical system monitoring configuration has at least one level of aggregation collectively referred to as level 1 (L1) aggregators. L1 aggregators are typically sited on non-compute-nodes (NCNs) and *pull* data from one to many *sampler* daemons over an HPC system's High Speed Network (HSN). Multiple levels of aggregation (respectively referred to as L2, L3, ...) can be utilized to transit network boundaries and finally store data at a suitable destination for analysis. In order to minimize the impact on applications, the operations associated with data sampling at compute-nodes are kept to a minimum (e.g., minimal to no processing of raw data). *Pulling* synchronous data via RDMA reads by aggregators reduces the compute-node overhead associated with data transmission. A compact binary structure called a *Metric Set* is used to reduce transport of information to the minimum required (i.e., only data values and modified meta-data). Meta-data here is defined as information describing data values such as names. Meta-data is transmitted on initial connection from one *ldmsd* to another or on change. Figure 1 illustrates when Metric Set data is exchanged on the transport.

Synchronized data collection is achieved across LDMS sampler plugins, both within a single compute node and system-wide, through the use of a wake-up driven sampling process scheduled against each *ldmsd* host's local clock. Synchronization errors in data acquisition across a cluster are the result of clock skew, which is typically minimal on a well-managed HPC cluster, and of wake-up decisions by the OS kernel for each *ldmsd*. In practice, sample time variations of a few milliseconds are seen across compute nodes on large scale systems.

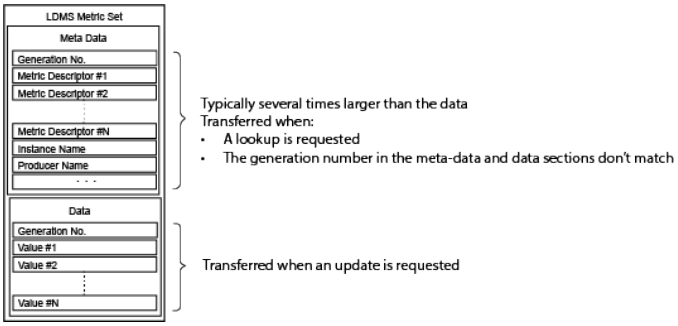

Fig. 1. Meta-data is only exchanged if modified

LDMS, being vendor and institution agnostic lends itself to incorporation, as a monitoring solution component, into any system. LDMS has been used for a number of years in production across a variety of large scale systems including Cori at NERSC and the commodity clusters at LLNL. As the HPE Cray EX systems come on line at these two institutions, each has taken a different approach to the continued use of LDMS as a technology for monitoring their compute-node infrastructure. NERSC, with Perlmutter, has installed the HPE CSM monitoring solution and LLNL is using the DOE Tri-lab Operating System Stack (TOSS) [8] provided LDMS components and their own configuration management approach. Note that the LLNL approach is advantageous to them as it enables a uniform approach to management and monitoring across all of their compute resources including El Capitan.

III. LDMS Implementation via HPE CSM on Perlmutter

In this section, we describe the LDMS implementation for metrics collection on Perlmutter, which is via CSM with some site-specific considerations. We further describe how the metric data is made available for exploration and how the LDMS implementation may be extended beyond the CSM-provided implementation to collect additional data of interest to system administrators and users.

A. The HPE CSM Architecture

The HPE Cray System Management (CSM) software stack is built around a Kubernetes technology in which the majority of the management software is run in isolated containers on a cluster of non-compute nodes (NCNs) communicating with one another through REST-formatted *http* connections. This architecture allows for high availability of services, simplified scalability as each service usage grows, and isolation from the failure of any particular hardware component. When a service needs to service a higher load, more copies of the application can be started within the Kubernetes cluster; each copy is called a *pod*. All System Monitoring Application (SMA) services are run as Kubernetes *pods* with the exception of LDMS samplers, which are run as native services within the node's operating system. SMA is a collection of technologies

which collect, aggregate, store, and display System Monitoring Data.

The Metrics pipeline has Kafka [9] at its center. The hardware metrics collectors (*hms-hmcollector*) listen to Redfish connections and publish the hardware data to Kafka. Node metrics are collected and transported off-node and into the Kafka bus using the LDMS infrastructure. System Monitoring Application (SMA) PostgreSQL persisters (*sma-pg-persisters*) are instances of a scalable, parallel tool for taking metrics (including LDMS metrics) from the Kafka bus and storing them in the HPE SMA PostgreSQL database. The number of running copies of *sma-pg-persister* can be scaled up or down based on system size. Test and Development systems will use only a single instance of the service, whereas a system with 5,000 nodes may require as many as 16 copies.

LDMS sampler daemons publish data to a scalable set of LDMS aggregator pods; one set for compute nodes and one set for NCNs. The LDMS aggregators then publish the data to the same Kafka bus that is used for transmitting log and hardware metric data. LDMS messages are indicated on Kafka by the topic `cray-node`. Each individual metric is a separate message, in JSON format, with additional self-describing information. An example is shown below:


```
{"metric":
{"name":"cray_storage.cray_iostat.w_await",
"dimensions":{"product":"shasta",
"system":"ncn",
"service":"ldms","component":"cray_iostat",
"hostname":"ncn-s003",
"cname":"x3000c0s33b0n0","job_id":"0",
"device":"sda3",
"persist":"wwn-0x5002538e001d844d-part3",
"device_type":"sd"},
"timestamp":1683150820123,"value":0.000000},
"meta":
{"tenantId":"f5d2c1289518469eb63f03fcf462f69b",
"region":"RegionOne"},
"creation_time":7450451465512054383}
```


Grafana [10] is used for viewing the metric data, and a number of HPE dashboards are included, though site custom dashboards may also be used.

LDMS configuration is accomplished by an *sma-ldms-config* pod taking the LDMS configuration from a Kubernetes ConfigMap and writing it to a local Simple Storage Service (S3) volume which is then read by nodes during the node personalization process.

The LDMS samplers provided as part of CSM are:

- `Mellanox` – the sampler for Slingshot10 which uses a Mellanox NIC
- `Slingshot` – a sampler for the Slingshot11 NIC
- `Ethtool` – an ethernet sampler that can be used with the above
- `procdiskstats` – basic I/O statistics
- `dvs`- a sampler for the HPE Disk Virtualization Service

Sampling intervals for all metric sets are 10 seconds by default but can be modified.

B. Perlmutter Monitoring

NERSC's monitoring of Perlmutter includes some site-specific variations to the CSM deployment. These are discussed below.

The LDMS RDMA `fabric` transport has been validated "in the small" on Perlmutter using HPE's `libfabric` over Slingshot implementation. However, since it has not yet been tested at scale or for resiliency in the presence of connection failure and transport errors, NERSC is currently using the LDMS socket transport on Perlmutter.

For the collection of additional metrics via LDMS, NERSC decided to run a separate LDMS instance alongside the SMA configured LDMS. The Ansible script provided by CSM doesn't provide much ability to do site-specific LDMS customization. Also, CSM doesn't provide many of the modules available in the open source version of LDMS. Because of this, site-specific customization of CSM configurations would need to be re-implemented with each CSM update.

NERSC exports LDMS data from SMA to a NERSC-wide shared data store called OMNI. OMNI utilizes VictoriaMetrics [11], an efficient and scalable time series storage that supports a rich query language (*promql*). A custom LDMS store plugin was written to export data to VictoriaMetrics via a gateway. The aggregator feeding this store runs alongside the SMA aggregators in a separate LDMS aggregator pod that is managed by the CSM Kubernetes cluster. This aggregator also makes direct connections to the compute node LDMS samplers started by SMA. Due to the small size of the LDMS data sets exported by compute nodes (a few kilobytes) it is not expected that the second socket connection will pose any burden to the network bandwidth available from the compute nodes, even at large scale.

NERSC has also made improvements to address problems with HPE's credential distribution to node LDMS configurations. In HPE's LDMS configuration, when nodes boot, SMA starts a pod on a fixed worker node (*ncn-w001*) with a Kubernetes Persistent Volume Claim (PVC) mount holding the LDMS secrets. SMA next copies the secrets to the node and then terminates the pod on *ncn-w001*. Using this approach on each node running a sampler is not only slow but unreliable. Nodes are frequently unable to retrieve the secrets, which causes the boot-time Ansible plays to fail which, in turn, prevents the node from booting. NERSC worked around this issue by moving the *sma-ldms-compute* Ansible role from *node personalization* to *image customization*. Also troubling in the credential distribution pipeline is the use of a fixed worker node (*ncn-w001*) for the PVC. If *ncn-w001* is unavailable or overloaded then a boot will either be extremely slow or fail altogether. Even though Kubernetes provides capabilities for load balancing and failover, SMA's approach for secrets distribution is unable to use these capabilities.

The NERSC run LDMS aggregator is managed by a python script with a boss/worker model, deployed in a Kubernetes pod. The boss finds all the nodes to monitor (compute, application, and management) from the Hardware State Manager

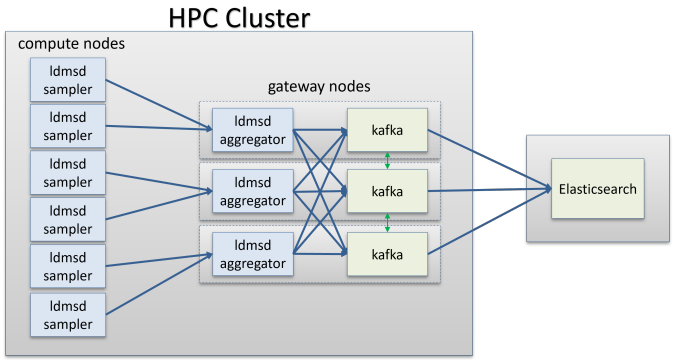

Fig. 2. LDMS data pipeline on El Capitan

(HSM) and generates a configuration file for each type. It then starts a sub-process worker for each configuration, with a message queue from the boss to each worker. When the boss detects node count changes, it generates a new configuration and restarts the worker. This second NERSC-supplied aggregator instance also enables us to run custom (or more recent) builds of upstream LDMS than provided by SMA without needing to replace the LDMS packages that SMA supplies.

HPE's *telemetry-api* acts as a Kafka consumer and http Security Service Edge (SSE) subscription service, feeding Kafka topic data to an http client. NERSC found the telemetry API to be unreliable and often inefficient. It stopped feeding data at random times, and frequently caused Kafka rebalancing events. From the http client side, one mitigation was to use *cron* to force a client restart. This is not ideal for production deployment. There were, however, no resource issues with HPE's *api-gateway*, and it has not seen a lot of active maintenance or improvement during the Perlmutter deployment. The *telemetry-api* could not handle the data rate (100K to 1M messages per second) of our full scale Perlmutter system.

NERSC tested a more efficient confluent-python based version of the *telemetry-api* and was able to show it could handle the full data rate.

NERSC continues to use the existing HPE LDMS aggregator, which uses the Cray-provided `Kafka storage` plugin as a Kafka producer to write to the `cray-node` topic. This data is read directly from Kafka and written to OMNI.

IV. LDMS Implementation via TOSS 4 for El Capitan

In this section, we describe the LDMS implementation for metrics collection intended for El Capitan via TOSS 4 with site-specific considerations. We further describe how the metric data is made available for exploration and how the LDMS implementation can be easily modified to address changing needs of system administrators and users.

A block diagram for the LDMS data pipeline on El Capitan is shown in Figure 2. The LDMS configuration for El Capitan needs to take into account the separate, but related, clusters running the same or similar hardware. In addition to El Capitan there are at least: a Test and Development System cluster, multiple Early Access clusters, and Lustre clusters. These systems will all run TOSS [8]. Since TOSS also runs on tens of other clusters at the lab, LLNL requires a monitoring configuration approach that can scale with the cluster count without requiring a corresponding scaling of staff.

At LLNL, all TOSS 4 based systems (including El Capitan and related clusters) are configured from one central Ansible repository. In the Ansible repository, there are two main *ldmsd* configuration templates: one template for the samplers' configuration(s) and another for the aggregators' configuration(s).

All decisions about which samplers are to run on a particular node are made through logic in Ansible. For instance, an LLNL custom "facts" script parses the output of `lspci`, and sets various facts about types of GPUs and network devices that are found. The template for the *ldmsd* sampler configuration file can then include the correct LDMS network configuration, and the correct respective samplers for any GPUs (e.g., `dcgm`, `rdc_sampler`) or networking devices (e.g., `slingshot_metrics`, `ibmad_records_sampler`). Other configuration decisions are made based on Ansible inventory settings. Changes to Ansible's configuration are automatically pushed to each cluster and Ansible automatically runs daily. System administrators can push and run Ansible changes manually when they do not wish to wait for the daily updates.

All LDMS operations are contained within their respective clusters. Each node of a cluster runs a sampler *ldmsd* while only a select subset of cluster nodes run a second *ldmsd*, on a different port, acting as an aggregator. The aggregators have a subset of *ldmsd* samplers statically assigned to them in their configurations via the Ansible template.

The Ansible template for the *ldmsd* sampler configuration file controls which samplers are running depending on a number of rules and tests in Ansible. While deployment is still in progress, LLNL's intent is to run the following samplers on El Capitan and related clusters:

- `meminfo` – dynamic node memory information
- `procnetdev2` – dynamic node network interfaces information
- `slingshot_info` – Slingshot interface information similar to that of the `cxi_stat` command line tool
- `slingshot_metrics` – a configurable set of Slingshot NIC counters
- `rdc_sampler` – metrics for AMD GPUs
- `lustre_client` – client-side lustre file system operations
- `lustre_ost` – server-side OST operations information
- `lustre_mdt` – server-side meta-data operations
- `zfs plugins` (under development) – metrics for ZFS zpools, vdevs, etc.
- `toss` – local compute node status

Two of the samplers, `slingshot_metrics` and `slingshot_info`, were custom developed by LLNL for El Capitan. Both samplers adhere to a philosophy of minimal

configuration. They automatically detect and report information for any number of slingshot interfaces. They both implement scaling through two relatively recent new features of LDMS: lists and records. The samplers use a single list of records in which each entry in the list contains a record of metrics for a single Slingshot interface. The samplers resize the list as interfaces appear and disappear.

The `slingshot_metrics` sampler reports on a configurable set of Slingshot NIC counters. Of the potentially hundreds of counters, the sampler defaults to using a set of roughly forty. These include performance and error related counters.

The `slingshot_info` sampler provides information similar to the information found in the default output of the `cxi_stat` command. The information includes FRU, part number, serial number, connection state, PCIe speed, link speed. This information is very useful for long term tracking, but generally these metrics are sampled at a much lower rate than the metrics found in the `slingshot_metrics` sampler. Implementing two samplers makes it easy to configure them with different sample rates.

LLNL intends to provide the job id of the currently running Flux top-level job on each node to fill in the `job_id` field in many LDMS metric sets. The prolog and epilog scripts could be used to publish the job ID to a `job_id` metric set in LDMS.

Most of the samplers are currently configured to sample on a five second interval, for instance `slingshot_metrics` and `rdc_sampler`. This is an initial compromise rate to help constrain the massive amount of monitoring data from El Capitan. The number will be adjusted as LLNL gains experience on El Capitan. Some samplers are set to a much slower rate, because they provide information that does not require fast sample rates, such as `slingshot_info` and `toss`.

On some TOSS 4 clusters LDMS is able to use its RDMA transport, but on El Capitan systems LLNL is currently using the socket transport. The Slingshot extensions for `libfabric` have not yet been upstreamed into `libfabric`. In TOSS 4 for El Capitan, the Slingshot-enabled version of `libfabric` would need to live in a side directory rather than being part of the standard TOSS 4 `libfabric` package, because TOSS 4 needs to run across many clusters and architectures, not only El Capitan. Thus far it has proven prohibitively difficult to build a standard TOSS 4 version of LDMS that can also use the Slingshot-enabled `libfabric`, because the LDMS build for TOSS 4 cannot be compiled against both the TOSS 4 `libfabric` and the custom Slingshot `libfabric` at the same time.

Each of the El Capitan related clusters, and eventually most LLNL TOSS 4 clusters, will have their own independent Kafka instance running internally. This provides all clusters with an independent and reliable location in which to publish all monitoring information.

In order to publish LDMS data to Kafka, LLNL and Open Grid Computing [12] collaborated to develop a new

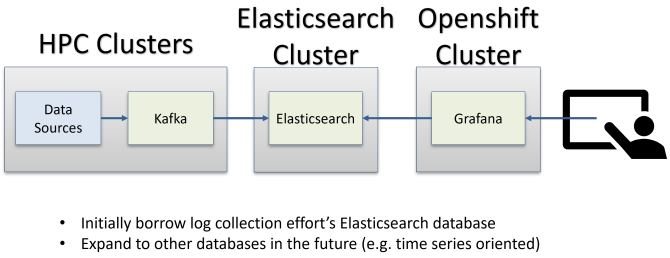

Fig. 3. LLNL Operational Data Flow Overview

LDMS store named `store_avro_kafka`. LDMS storage policies are configured to elect all or a subset of metrics from each metric set and pass these values as *rows* to `store_avro_kafka`. `store_avro_kafka` uses Apache Avro, which is a system for serializing data. In Avro, the data is described by an Avro schema. In order to either serialize or deserialize a piece of data, the schema needs to be known. To keep the binary-serialized data compact, the schema is not transmitted with the data. The serializer and deserializer must use some method to share or exchange the schema. `store_avro_kafka` automatically generates schemas for configured rows from LDMS metric sets and publishes those schemas to a Kafka Schema Registry. For each row of data, `store_avro_kafka` serializes it using Avro and publishes it to a topic in Kafka. For each schema, there is a separate topic in Kafka.

The Kafka data bus acts as a common point of data sharing between multiple producers and consumers. For instance, on the producer side, LLNL has at least LDMS and Redfish. On the consumer side, one of the main initial consumers will be a centralized Elasticsearch database.

LLNL has pre-existing Elasticsearch databases that were stood up by a different efforts to process logs. The original goal of only processing logs has been expanded to incorporate LDMS and other data from El Capitan. To support this additional load, the El Capitan project has contributed a sizeable set of hardware on which to run Elasticsearch. This data flow is shown in Figure 3. While Elasticsearch was a low effort place to send the data from the Kafka bus initially, LLNL may expand its database offerings in the future.

Data from Kafka topics is streamed to Elasticsearch using the Kafka Connect Elasticsearch Service Sink connector. This Sink is a piece of software that acts as a consumer of topics. As messages appear in one or more topics, the Sink consumes those messages, uses Avro to deserialize them (looking up the matching schemas in the Schema Registry), and writes the data into the correct indices in Elasticsearch. Using a simple regular expression, LLNL configures the Sink to watch many topics and write the data from each into their own respective index in elasticsearch. LLNL also employs a feature of the Connect infrastructure called a Simple Message Transform (SMT). The SMT allows them to rewrite all message fields named ”timestamp” to be named ”@timestamp”, the latter being the common default name in Elasticsearch.

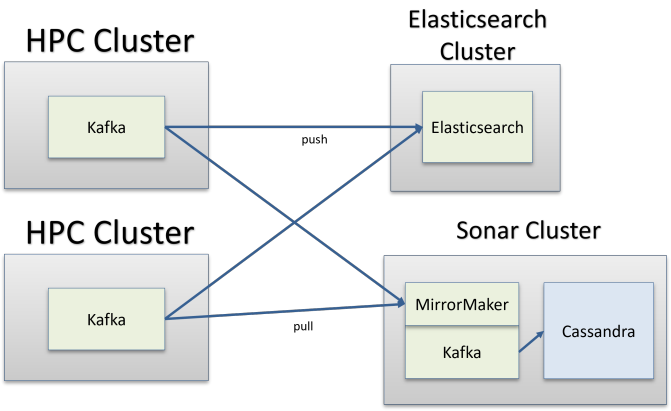

Fig. 4. LLNL's Sonar project provides an interface for users and administrators into El Capitan as well as other LLNL systems' data.

Unlike the Kafka instances, the Elasticsearch database lives on a separate, centralized cluster. All of the clusters stream their data to the central Elasticsearch via their own independent Sink connectors.

Once data is in Elasticsearch, it will be visualized though an instance of Grafana [10] that lives on a centralized OpenShift cluster. Much like the choice to use Elasticsearch was heavily influenced by it already being available within LLNL's center, the OpenShift clusters are pre-existing, and enable easy stand up of containerized services like Grafana.

Another consumer of the Kafka data from El Capitan will be LLNL's Sonar Project [13]. The Sonar project was created about a decade ago as a distributed monitoring system that could capture data from across all LLNL systems and provide easy-to-use analyses and visualizations to both users and administrators.

The architecture of the data flow supporting both Elastic-Search and Sonar is shown in Figure 4. The use of Kafka for the El Capitan system will enable the LDMS data collected on El Capitan to be easily ingested into the Sonar monitoring infrastructure through use of *Kafka MirrorMaker*. *MirrorMaker* replicates a subset of topics from the El Capitan Kafka bus (specifically timeseries LDMS data) onto the Sonar local Kafka bus where the messages can be further transformed before being inserted into the Sonar database. The advantage to using a message bus such as Kafka is that the data on the bus is database technology agnostic. Thus Sonar can continue to utilize Casandra to expose system metrics to users of the computing facility while El Capitan will transmit all LDMS topics to a central Elasticsearch database for use by the Operations and System Administration teams. This abstraction provides the flexibility to enable adaptation to potentially changing needs and opportunities (e.g., new database technologies and analysis targeted databases).

V. New LDMS Features

In this section we present some of the new features of LDMS that can significantly enhance a sites monitoring experience.

- RAILS - Encapsulation of multiple Zap connections inside a single LDMS transport.
- Streams - An API for applications to inject JSON objects or text strings into the LDMS monitoring infrastructure.
- Application Samplers - A set of samplers Includes application and system monitoring data and inter-operates with Kokkos (SNL), Caliper (LLNL), and Darshan (ANL)
- Slingshot-sited Sampler - A slingshot sampler than runs directly on a Slingshot switch controller
- Maestro - Distributed configuration of the monitoring infrastructure.
- Security - Credential based storage, discovery and delivery of monitoring data.
- Kafka Integration - Facilities for the delivery of Avro encoded LDMS metric data on the Kakfa bus.
- Logging - A common logging facility supporting subsystem specific message filtering.

A. RAILS

At the first level of aggregation, each LDMS daemon (*ldmsd*) has a connection to each sampler daemon, i.e., one connection and associated thread per sampler daemon. At the next and subsequent levels of aggregation, however, each *ldmsd* has one connection and associated thread for many sampler daemons by virtue of the aggregation done at the previous tier. Figure 5 shows the scaling issue absent RAILS.

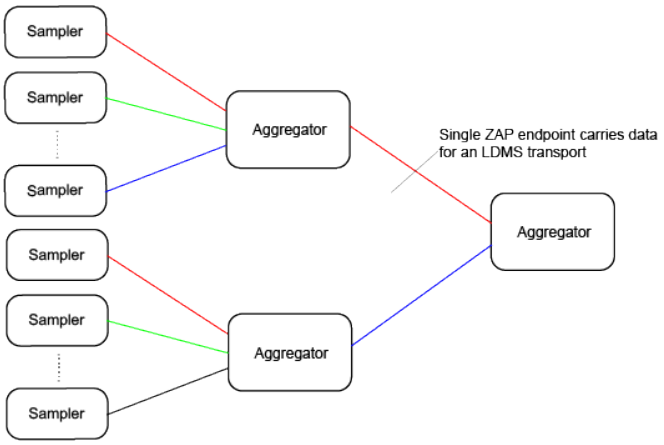

Fig. 5. Transports utilizes a single Zap endpoint

The RAILS feature (coming in LDMS version 4.4.1) addresses the scalability issues imposed by the multi-level aggregation infrastructure. By enabling the addition of multiple connections and associated threads to the communication channel (aka LDMS transport) between *ldmsd*s at different aggregation levels, RAILS provides higher aggregate network bandwidth and compute resource available for updating, storing and publishing metric set data. Figure 6 shows transports using RAILS.

B. Streams

It is desirable to collect message oriented data from applications and other system services. The *streams* API enables applications to inject structured (JSON) and unstructured

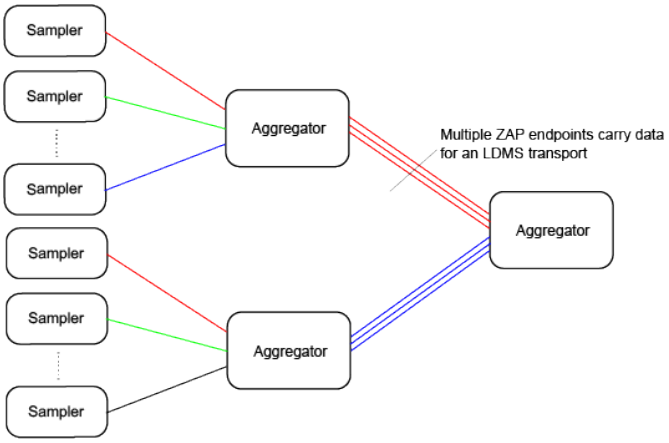

Fig. 6. Transport with multiple Zap endpoints

(STRING) data into the LDMS transport, thereby getting the data into monitoring pipeline. The producer API is message oriented and accepts a stream name, buffer, length and data type (JSON or STRING).

Clients of a message stream register for a stream by stream name. When data arrives on a stream, a callback is called with a pointer to the data buffer, and if the data is formatted as JSON, a parsed JSON object. As of LDMS version 4.4.1 and later, there are stream stores that can be used to store JSON object data to a storage back end (e.g. `store_csv`, `store_sos`)

C. Application Samplers

The *streams* capability has been leveraged to inject asynchronous data related to application events into the LDMS monitoring pipeline. The *AppSysFusion* [14] project has developed a capability to inject selected application kernel timing information from Kokkos via the Kokkos Profiling Interface [15] into the LDMS transport using the *stream* API via the `kokkos-connector`. Kokkos [16], [17] is a performance portability library which abstracts away architecture-specific execution and data management details of computational kernels. Application developers use Kokkos to easily port their applications to new architectures in a performant manner without having to rewrite and tune their codes. By obtaining kernel and timing information from Kokkos, application information is obtained in a scalable-fashion without requiring recompilation by the user. This enables lightweight "always-on" monitoring of applications in conjunction with systems.

The architectural design of the `kokkos-connector` is also suitable for use by other data sources. Sandia and LLNL developers have built connectors for injection of Caliper [18] data and Sandia and ANL developers have built connectors for injection of Darshan [19] data into the LDMS transport. While these external tools do have options for collection of time-series data at execution time, injection into the LDMS transport provides the benefit of bounding the on-node memory footprint for holding the data and removing the I/O costs associated with writing the data out to a file system.

D. Slingshot-sited Sampler

The Slingshot switch hosts an ARM processor which enables processing capabilities to be sited alongside switch functionalities. An *ldmsd* can be run directly on this processor to collect exposed slingshot metrics. It can create a metric set which can be transported over the management network via the `sock` transport. This is in contrast to the slingshot samplers mentioned elsewhere which are sited *on the node* and therefore can only collect Slingshot NIC metrics exposed to the node.

The LDMS slingshot-sited `slingshot_switch` sampler currently uses as its data source the HPE-provided `dump_counters` binary. The `dump_counters` program takes command line arguments to specify ports and counters, or sets of counters, of interest. There are over a thousand counters per port and over 64K total port counters per switch. The execution time to retrieve the full set of metrics is about half a second including nominal time for the *ldmsd* to parse the output and populate a metric set. The `dump_counters` arguments are provided to the `slingshot_switch` sampler in a configuration file. The output metric set is in the form of a list of records, one per port, analogous to the metric set output of `procnetdev2`. We would like to work with HPE to identify more efficient port counter access mechanisms.

E. Maestro

Many monitored systems consist of thousands of sampler daemons, and tens of aggregator daemons organized into multiple tiers (i.e. aggregation levels). There is a considerable amount of configuration data, in the aggregate, required to configure the monitoring infrastructure. The *Maestro* [20] service manages this data and distributes the required configuration to each *ldmsd* daemon in the system. Figure 7 illustrates the configured infrastructure.

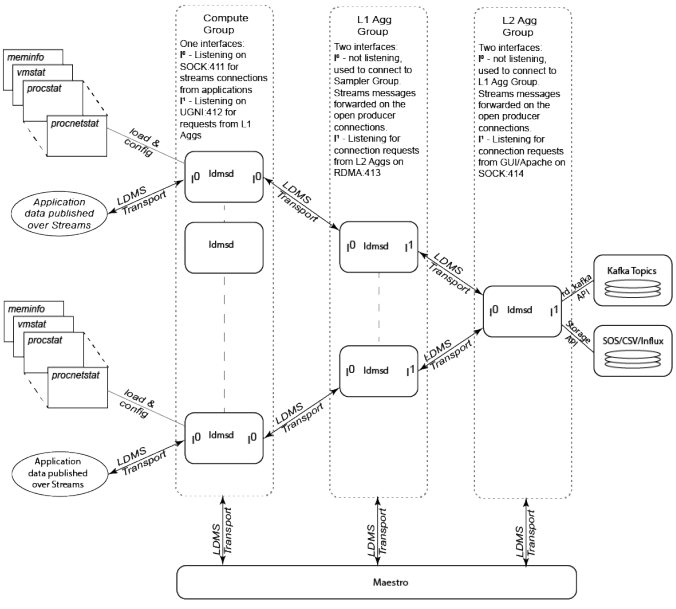

Fig. 7. Infrastructure Managed by Maestro

Maestro stores configuration data in a distributed and resilient database called *etcd* [21]. *Maestro* itself consists of three or more *maestro* daemons that collectively use the *RAFT* [22] protocol to implement the resilient distribution of configuration data. If any one *maestro* daemon fails, another daemon takes over the responsibilities of the failing daemon.

Maestro organizes *ldmsd* into groups. Typical groups include samplers and aggregation levels as shown in Figure 7. *Maestro* load balances and redistributes configuration to *ldmsd*s in each group. *Maestro* provides resiliency in the monitoring infrastructure by continuously monitoring *ldmsd* daemons. If an *ldmsd* fails, its configuration is re-distributed to the remaining *ldmsd*s in the group. The criteria for load-balancing is one of *producers* (i.e. an *ldmsd* instance) or *set_load*. The *set_load* is computed as follows:

$$set\_load = \sum_{k=1}^{set\_count} set\_data\_size_k * \frac{1000000}{update\_interval_k}$$

where *k* is the LDMS metric set id, *set_data_size* is the size of the metric set's data, and *update_interval* is the update interval in micro-seconds. If the *ldmsd* is a first level aggregator, then the *update_interval* is the sampler plugin's sample interval in microseconds.

When load balancing by *producer*, the number of producers started on each *ldmsd* is made as equal as possible. This works very well when the set count and set schema provided by each producer are roughly equal. When the number of sets are not equal or the schema of the sets are different, this can result in an unequal load on some *ldmsd*s. In this case, it may be preferred to load balance by *set_load* instead.

F. Security

Credentials, (UID, GID, and access mode) are assigned to transports, metric sets, and streams. Access to metric set and stream data is controlled by the *ldmsd* daemon. Credential information is carried in the transport such that clients attempting to access metric set and stream data are authenticated and their access controlled accordingly.

1) Transports: An LDMS transport is associated with an owner and group. How the owner and group are assigned depends on the authentication plugin chosen for the connection. If the authentication method is *none* or *ovis*, the UID and GID assigned to the transport is (0, 0).

If the authentication method is *munge*, the *munge* daemon is used by the active and passive side of the connection to validate the UID and GID of the connecting process. The UID and GID are obtained with `geteuid()`, and `getegid()` respectively. If munge authentication is successful, the UID and GID placed in the metric set header are (`getegid()`, `getegid()`).

2) Metric Sets: The owner and group of a metric set is contained in the metric set meta data and is therefore forwarded through the infrastructure from the sampler *ldmsd* to the top level aggregator *ldmsd*. When a client, for example *ldms_ls*, requests the metric set list from an *ldmsd*, the daemon consults the owner, group and permissions stored in the metric set header. If read access is not granted for the owner and group contained in the transport on which the request was made, the set is not included in the set list and is therefore invisible to the client.

The UID, GID, and access rights are assigned to the set when the set is created, or later by a configuration command. Only *ldmsd* running as root can assign UID, GID that are different than what is returned by `geteuid()`, and `getegid()`. This prevents non-root users from masquerading as other users.

3) Streams Data: The *ldmsd* is a broker for streams data. This data is only forwarded to subscribed clients that are authorized to receive this data.

G. Kafka Integration

Kafka [9] is an event distribution infrastructure that is popular for publishing data on Cray/HPE and other platforms. *Avro* [23] is a serialization protocol for encoding network data. *Avro* is often used because it enables the unambiguous exchange of event data values that are significantly smaller than the un-encoded data.

The LDMS *Avro-Kafka* store makes metric set data available on the Kafka bus. The Kafka message data is encoded either as JSON text or an Avro encoded object. When encoded in Avro, schema management is provided via an Avro Schema Registry [24]. The *Avro-Kafka* storage plugin adds new schema to the registry as required. Clients consult the registry to receive the schema definition necessary to decode the Avro objects contained in each Kafka message.

H. Logging

A generic message logging service and API has been added that:

- Normalizes message output
- Tags all messages with a subsystem and log mask

Internal to the message library, each subsystem and plugin can be assigned its own logging mask. This enables precise control over how much logging data is produced by each subsystem. For example, the *slurm2* plugin could be configured with a log mask that includes *DEBUG* without affecting the levels of message data logged by other subsystems and plugins.

The configuration of the logging mask can be performed dynamically with the *ldmsd_controller* application. This allows the administrator to change log masks at run time to provide greater log visibility for subsystems and plugins of interest.

VI. Versatility/Sustainability

In this section, we highlight key similarities and differences in two implementations (CSM and TOSS 4) of node monitoring on the HPE Cray EX system using LDMS and also present open issues and questions to be explored. Some implementation differences are site-specific choices (e.g., data store choices) and multiple options are intended to be supported. However, in some cases, implementation details may result in difficulties for sites' abilities to easily stay current with top of tree and take advantage of new LDMS features,

such as those described in Section V. Divergence in core LDMS functionality can also create an additional community burden because it creates an artificial need to support multiple concurrent implementation paths. Both NERSC and LLNL are regular contributors to the LDMS source and to directional discussions and can help drive common paths forward.

- Both NERSC and LLNL are using the LDMS socket transport (not RDMA), but for different reasons:
  - NERSC is not using the RDMA transport of LDMS because the LDMS `libfabric` transport has not yet been tested at scale for resiliency in the presence of connection failure and transport errors on HPE's `libfabric` over Slingshot implementation.
  - LLNL is not testing or using the RDMA transport of LDMS because HPE's `libfabric` implementation is different from the upstream Open Fabric Alliance (OFA) version used in RHEL and its use would require multiple concurrent versions of LDMS, one for OFA `libfabric` and one for HPE's Slingshot `libfabric`, to be maintained within TOSS 4.
- Version discrepancies:
  - HPE ships an internal version of LDMS with CSM which is significantly behind the latest LDMS release and does not contain many of the enhancements discussed in Section V. TOSS 4 is regularly updated and provides the latest LDMS release which keeps LLNL's LDMS deployments current. NERSC's self-installed version also stays current with the latest LDMS release.
  - HPE's CSM contains LDMS samplers that have not been up-streamed and may be proprietary.

  It is unclear what plans HPE has to upstream its enhancements to the open source release on github.
- Ansible scripts and other configurations: How different are these? Can they be shared/leveraged? How much does this matter?
- CSM Containerized deployments: How valuable is this feature/implementation? What are the tradeoffs vs RPM install on bare metal? How does the CSM implementation compare to the open source LDMS containerized dockerhub [25] release and what might be leveraged?
- Kafka message formats: LLNL and HPE have different formats. LLNL uses Avro encoding and sends configured combinations of metrics in a single Avro encoded message, resulting in significantly fewer bytes transmitted. HPE sends one metric per message in JSON format with additional self-describing information, simplifying schema management. Is there a plan for HPE to move to the Avro format?
- Kafka topics: Is there a need/advantage for a topic directory service for managing topics and descriptions to facilitate the processing of expanded LDMS data sources (e.g., per-application data) on the Kafka bus?
- Performance insights: Can the community meaningfully share any knowledge/experiences, given the implementation differences? For example, the variation in the Kafka format means there can't easily be a rule of thumb in one implementation that extends to the other. Perhaps fan-in and sampling-rates vs. number and size of metric sets vs. Kafka resources can be characterized and shared.

VII. Conclusions

In this paper we have begun to explore the implementation differences in deploying and configuring LDMS and in making LDMS-based data available for analysis on the NERSC Perlmutter system and intended for the LLNL El Capitan system. We have highlighted differences and open questions that may affect the versatility and sustainability of the LDMS instantiations as the open source LDMS codebase continues to evolve. The authors will continue to work together to explore possible convergence on some of the identified directions and open questions.

The designs in this paper may be subject to revision as they are exercised on the platforms. Perlmutter is currently undergoing work for acceptance and while there are El Capitan Early Access Systems, El Capitan has not yet been delivered. The authors will be working together to explore best practices to scalably support the potential data flow from these, and other, extreme-scale systems.

VIII. Acknowledgements

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

References

[1] A. Agelastos, B. Allan, J. Brandt, P. Cassella, J. Enos, J. Fullop, A. Gentile, S. Monk, N. Naksinehaboon, J. Ogden, M. Rajan, M. Showerman, J. Stevenson, N. Taerat, and T. Tucker, "The lightweight distributed metric service: A scalable infrastructure for continuous monitoring of large scale computing systems and applications," in *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2014, pp. 154–165.

[2] A. Agelastos, B. Allan, J. Brandt, A. Gentile, S. Lefantzi, S. Monk, J. Ogden, M. Rajan, and J. Stevenson, "Toward rapid understanding of production hpc applications and systems," in *2015 IEEE International Conference on Cluster Computing*, 2015, pp. 464–473.

[3] OVIS-HPC. OVIS/LDMS. [Online]. Available: https://github.com/ovis-hpc/ovis

[4] Hewlett Packard Enterprise Development LP. Cray System Management Documentation. [Online]. Available: https://cray-hpe.github.io/docs-csm/en-10/

[5] ——. HPE Performance Cluster Manager. [Online]. Available: https://www.hpe.com/psnow/doc/a00044858enw

[6] NERSC. Using Perlmutter. [Online]. Available: https://docs.nersc.gov/systems/perlmutter/

[7] LLNL. LLNL and HPE to partner with AMD on El Capitan, projected as world's fastest supercomputer. [Online]. Available: https://www.llnl.gov/news/llnl-hpe-partner-amd-el-capitan-projected-worlds-fastest-supercomputer

[8] E. León, T. D'Hooge, N. Hanford, I. Karlin, R. Pankajakshan, J. Foraker, C. Chambreau, and M. Leininger, "Toss-2020: A commodity software stack for hpc," in *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis, (SC'20)*, 2020.

[9] Apache Foundation. Apache Kafka. [Online]. Available: https://kafka.apache.org/

[10] Grafana Labs. Grafana: The Open Observability Platform. [Online]. Available: https://grafana.com/

[11] VictoriaMetrics. VictoriaMetrics: Simple & Reliable Monitoring For Everyone. [Online]. Available: https://victoriametrics.com/

[12] OGC. Open Grid Computing. [Online]. Available: https://ogc.us

[13] Gimenez, Alfredo A. and USDOE National Nuclear Security Administration, "Sonar," 2018. [Online]. Available: https://www.osti.gov//servlets/purl/1493001

[14] O. Aaziz, B. Allan, J. Brandt, J. Cook, J. E. Karen Devine, A. Gentile, S. Hammond, B. Kelley, L. Lopatina, S. Moore, S. Olivier, K. Pedretti, D. Poliakoff, R. Pawlowski, P. Regier, M. Schmitz, B. Schwaller, V. Surjadidjaja, M. S. Swan, N. Tucker, T. Tucker, C. Vaughan, and S. Walton, "Integrated System and Application Continuous Performance Monitoring and Analysis Capability," Sandia National Laboratories, Tech. Rep. SAND2021-11184, 2021.

[15] S. D. Hammond, C. R. Trott, D. Ibanez, and D. Sunderland, "Profiling and debugging support for the Kokkos programming model," in *High Performance Computing*, R. Yokota, M. Weiland, J. Shalf, and S. Alam, Eds. Cham: Springer International Publishing, 2018, pp. 743–754.

[16] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202 – 3216, 2014, domain-Specific Languages and High-Level Frameworks for High-Performance Computing. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731514001257

[17] C.R. Trott, et al., "Kokkos 3: Programming model extensions for the exascale era," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 805–817, 2022.

[18] D. Boehme, T. Gamblin, D. Beckingsale, P.-T. Bremer, A. Gimenez, M. LeGendre, O. Pearce, and M. Schulz, "Caliper: Performance introspection for hpc software stacks," in *Supercomputing 2016 (SC16)*, no. LLNL-CONF-699263, 2016.

[19] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, and R. Ross, "Understanding and improving computational science storage access through continuous characterization," in *Proc. of 27th IEEE Conference on Mass Storage Systems and Technologies (MSST 2011)*, 2011.

[20] OVIS-HPC. LDMS Monitoring Cluster Configuration Management and Load Balancing Service. [Online]. Available: git@github.com:ovis-hpc/maestro.git

[21] etcd.io. A distributed, reliable key-value store for the most critical data of a distributed system. [Online]. Available: https://etcd.io

[22] raft.github.io. The RAFT Consensus Algorithm. [Online]. Available: https://raft.github.io

[23] The Apache Foundation. Apache Avro™- a data serialization system. [Online]. Available: https://avro.apache.org/

[24] Confluent. Schema Registry Overview. [Online]. Available: https://docs.confluent.io/platform/current/schema-registry/index.html

[25] OGC. Dockerhub hosted containers for LDMS deployment. [Online]. Available: https://hub.docker.com/r/ovishpc/ldms-build