SAND2019-5617C

Exploring New Monitoring and Analysis Capabilities on Cray's Software Preview System

Jim Brandt[§], Connor Brown[‡], Scott Donoho[*], Ann Gentile[§], Joe Greenseid[*], William Kramer[†], Patti Langer[*], Aamir Rashid[*], Kevan Rehm[*], and Michael Showerman[†]
[*]*Cray Inc.*
Email: (sdonoho|joeg|planger|arashid|krehm)@cray.com
[†]*National Center for Supercomputing Applications (NCSA), Urbana, IL USA 61801*
Email: (wtkramer|mshow)@ncsa.illinois.edu
[‡]*SAIC, Albuquerque, NM USA*
Email: conbrow@sandia.gov
[§]*Sandia National Laboratories (SNL), Albuquerque, NM USA 87123*
Email: (brandt|gentile)@sandia.gov

***Abstract*—Cray, NCSA, and Sandia staff and engineers are collaborating to jointly investigate and provide new insights on the monitoring aspects of Cray's recently released "Software Preview System." In the preview system, Cray has implemented the LDMS framework within the monitoring infrastructure. In this work, we extend that implementation and leverage the Cray infrastructure to include new monitoring capabilities suitable for addtional node-level and application monitoring. We use the Cray-provided telemetry bus for transport and consumption of the new metric data. We explore scale and performance considerations. We provide details on the issues impacting or facilitating implementation of these functionalities within Cray's new, container-based services system. In our implementation, we adhere to Cray's design philosophy which is intended to ensure the reliability and availability of the Cray-collected metrics and system services.**

I. Introduction

Performance of systems and applications is highly dependent on effective use of system resources. Contention for shared resources, such as the interconnect; running on faulty components; or running in imbalanced decompositions can significantly degrade performance. However, continuous detailed insight into resource state and utilization, where such understanding often relies on higher-level analysis and integration of raw data, is not a delivered capability on large-scale systems. Often system operators must develop one-off, site-specific capabilities to deliver individual aspects of what is needed for a complete system solution.

To support this need for deriving actionable information at run time at scale from raw data, in the next evolution of Cray's system software (as deployed in Cray's Software Preview System), Cray has advanced the architecture and capabilities of its monitoring software and infrastructure.

At the same time, the platform's architecture, system services, and potential user environment have evolved as well. In previous generations of Cray's supercomputing systems, system services run on a bare-metal-installed OS on specific physical servers which may be configured as HA failover pairs. In a substantial departure from previous platform architectures, it is expected that Cray's next generation platform, Shasta, will have system services running in containers that may or may not be able to migrate between servers as part of load balancing efforts. Other services, such as compute nodes, may have bare-metal OS installations and support user-launched jobs that run in the host OS or user-launched containers.

These changes will require a re-evaluation of how sites adapt local customization and supplementation of Cray-provided services. New capabilities may have to be developed to enable desired complex scenarios such as monitoring system services or resource demands of applications running inside containers, as well as feedback of analysis results to both user and system software. Such development requires understanding of how data from within containers can be extracted and integrated with other monitoring data, how custom data can be injected into the telemetry bus, how available networks (e.g., management, High Speed Networks (HSN)) can be used for low-latency data transport, and how performance optimizing responses (e.g., workload-resource mapping via container migration) can be invoked. Further, sites must understand how collectors and aggregators provided by Cray as part of its delivered monitoring services function, and what adaptations are necessary to create site/user/job- specific collectors and aggregators in this mixed container/bare-metal-OS environment.

Cray developed its Software Preview System to enable interested sites to begin exploring the implications of new features and mechanisms currently under consideration for inclusion in the Shasta platform. Included in this is a monitoring service which contains Cray-specific components, Lightweight Distributed Metric Service (LDMS [1]) for in-band data collection, and a telemetry bus for distribution of telemetry data to system users and services. Use of the

Sandia National Laboratories is a multimission 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.

Preview System is expected to generate feedback about what works, what doesn't, and suggestions for desirable changes based on experience with evaluation/pre-release software. Identified issues may addressed in the production Shasta release.

Cray, NCSA, and Sandia staff and engineers are collaborating to investigate, and provide new insights on, the monitoring-related aspects and potentials for this system. This includes support in the design for site-specific customizations and data access, which have been high priority requests by sites in the past [2]. In this paper, we present our current progress in this investigation, using Sandia's Cray Software Preview System, Perkins, running Software Version 0.4.0. We first present a high level architecture of the Cray Preview System's Monitoring system components and how we are extending and leveraging them to support additional user metrics in Section II. We present details of the implementation enabling additional monitoring and making the data available to the telemetry bus in Section III. We present options for getting metrics off of the telemetry bus and making them available for viewing and analysis in Section IV. Within those sections we provide informative detail of our experiences using the current Software Preview System. We present implementation and results of a scaling study in Section V. We present our next steps and future work in our collaboration in Section VI. We conclude in Section VII.

II. System Overview

In this section we address the Cray-provided monitoring capabilities in the Software Preview System and how we are extending them to support addition of monitoring capabilities to the system, while leveraging the current architecture.

A. Software Preview System Architecture

The preview system components are described in [3] as "The system is a cluster consisting of a single 19 inch rack with eight nodes. Nodes are configured as management nodes, service nodes, or compute nodes (all server hardware is identical). The system includes a high-speed interconnect network (HSN) based on Mellanox ConnectX-5 NIC, an integrated storage system with a pre-configured Lustre file system, two management networks, and two power distribution units (PDUs). The HSN is shared by the management nodes, service nodes, compute nodes, and the storage system." A graphic of the system is shown in Figure 1 (from [4]).

B. Software Preview System Monitoring Architecture

A diagram of Cray-provided monitoring for the system is shown as Figure 2. The architecture supports data collection and transport from a variety of numeric and text based sources, both out-of-band and in-band. Cray has not yet

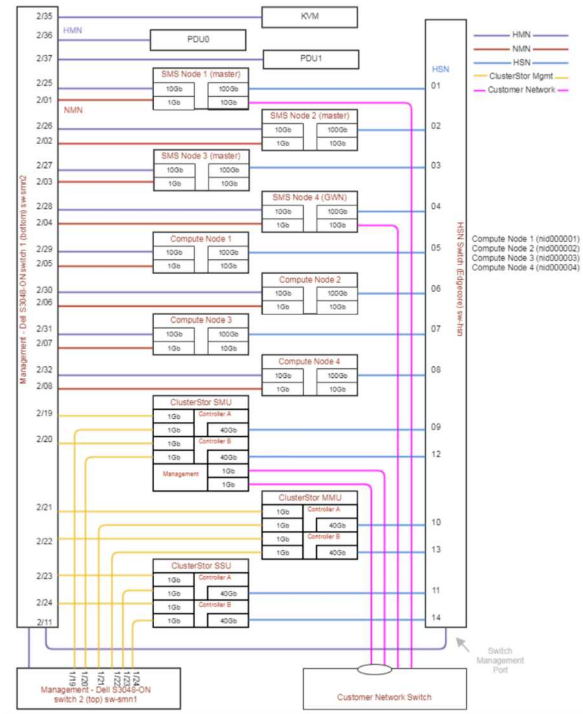

Figure 1. Diagram of the Cray Preview System (from [4]).

announced the full set of data which they intend to collect and make available.

In this work, we focus on the LDMS-initiated monitoring and subsequent data flow. (Relevant high-level information on LDMS can be found in Section II-C). LDMS collectors run directly on the bare metal of the system nodes. The LDMS aggregator is run in a container that for our system has been configured to reside on SMS04 in the *sma* namespace. In the general case the *SMS* nodes collectively host all of the containers related to monitoring and metrics collection for the system. The aggregator runs a Cray-implemented LDMS store plugin which writes its metrics to a kafka [5] based telemetry bus. The metrics that Cray is collecting via LDMS are read from the telemetry bus, and stored in the Postgres database, where they drive a Grafana [6] based display. Metrics or Groups of metrics are identified by the topic to which they are published on the telemetry bus. All Cray LDMS-collected metrics are currently published to the topic "metrics". The sampler data in our current system is transported via high speed 100Gb/s ethernet connections that connect the SMS04 node running the aggregator pod to the compute nodes running the samplers.

C. LDMS

Details of the Lightweight Distributed Metric Service (LDMS) [1] are beyond the scope of this work. Information relevant to this paper is described in this subsection.

LDMS collects and transports data from distributed

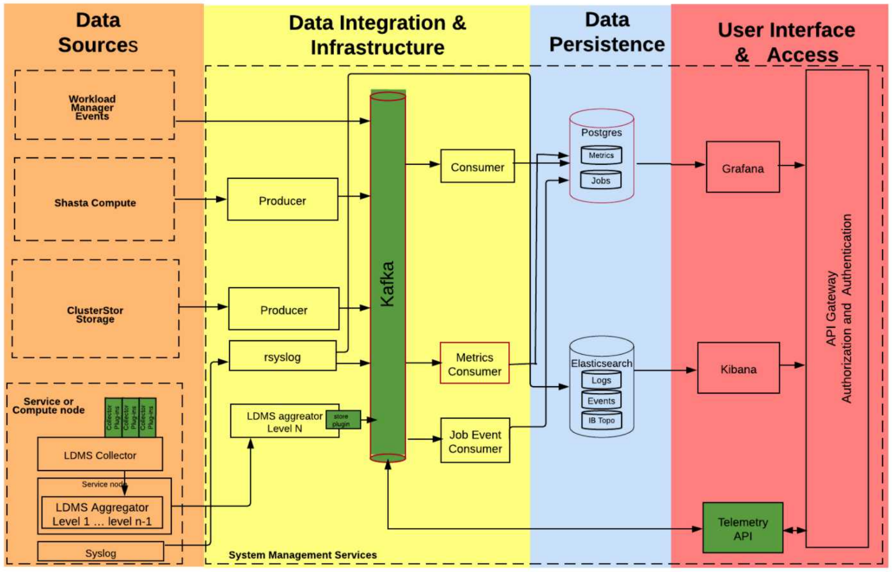

Figure 2. Diagram of monitoring in the Cray Preview System. Components elaborated on in this work are the LDMS sampler, aggregator, and store to kafka; the kafka bus; the telemetry API; and the encompassing container-based environment.

sources at resolutions necessary for detecting features and events of interest and to respond on meaningful timescales. Data is collected, typically at intervals of 1 second of less, at the same time at distributed data sources in the system (e.g., individual nodes), within the errors of the system clocks and offsets due to competing system processes (in practice this is on order of a few ms).

LDMS daemons run as *samplers*, *aggregators*, or *stores*, with each daemon consisting of the same base, but with functionality discriminated by the daemon configuration and *plugins*. LDMS supports high fan-in of data-sets to aggregators (i.e., tens of thousands to one), using RDMA or a socket based transport over a system's High Speed Network (HSN). Multiple instances of LDMS can be run concurrently.

Data is collected into *metricsets* typically consisting of multiple metrics from the same data source (e.g., `/proc/meminfo`) that may be efficient to collect at the same time or of multiple metrics from different data sources for the same component (e.g., selections from `/proc/meminfo` and `/proc/loadavg`) that it is convenient to keep together with the same timestamp. The memory layout is prioritized for metricsets and metricsets are pulled together in a single RDMA fetch.

In current site installations on XC and XE systems, LDMS is used for in-band collection of node-exposed data such as memory and CPU utilization counters, PAPI and/or MSR [7] hardware counters, and Cray-exposed Aries [8], [9] and Gemini [10], [1] performance counters.

The version of LDMS that is deployed in Preview software is LDMS v3 with Cray-developed versions of some samplers, supporting changes in the core, and a `store_kafka` plugin. To support extending the monitoring capabilities, we ported Cray's `store_kafka` to LDMS v4 [11] in order to run the most recent samplers and to exercise the collection rate change feature. (This plugin is not yet part of the LDMS v4 release.) Version differences and details are orthogonal to this work; nevertheless, we were pleased that it was straightforward to port the store and support multiple versions. The LDMS developers and Cray will determine a consolidated development path, going forward.

D. Extending the Architecture for User-Driven Metrics and Storage

The goal of this work is to explore and extend the vendor-provided monitoring infrastructure to enable additional, user driven, monitoring of the Software Preview system components and storage of both vendor and user configured metrics. In this work, we then refer to this additional monitoring and storage as *user-driven*, referring to the operations staff as opposed to the users running applications on the system.

Cray's intended architecture for the additional metrics is for the user to run separate samplers, whose data is aggregated by one or more LDMS aggregators running in separate user-launched, containers. The Cray-provided aggregator would be untouched. Metrics from these aggregators will be transported over the same telemetry bus as the Cray-collected metrics, but can be differentiated by topic (currently one per aggregator). In this work, we publish each separate set type to a separate topic. This is shown in Figure 3.

Metrics can be read off the telemetry bus via the telemetry API, as is Cray's intent, or by reading from the bus directly. Ideally, we would read the metrics off the metric bus and redirect them off the system to a large-data store, for analysis, or to other downstream consumers. Since the Perkins system does not include off-system storage, we instead investigated two different models that each capture only part of the ideal scenario. In one option, on nid00004, we read off the telemetry bus via the telemetry API and wrote the output to CSV files. In another, we investigated insertion into a database; this case required more storage space, which we had on SMS04, and therefore, we chose to read directly off the bus and test insertion there. Neither case is ideal, but both enabled us to study options and performance for consuming off the bus.

We had constraints on adding external systems to the preview system, so we were forced to use some of the compute nodes for more administrative purposes. In our current setup, we have added additional packages and RPM's to nid000004 in order to build and test LDMS. We further use nid00004 for using the cray-provided pre-release `client.py` script to pull metric data off of the telemetry bus via with the telemetry API and to locally store the metrics produced by our additional monitoring. This represents a user-driven model where a user would like to sample additional data and store it in their own database for processing without interfering with the existing cray monitoring infrastructure.

III. Implementation of User-driven Metric Collection

Additional data collection in the Software Preview architecture is not as simple as merely adding additional collectors to the currently running LDMS aggregator, due to design considerations to ensure reliability and availability of the Cray-provided data and services. We describe the requirements and resulting implementation in this section.

A. Design Philosophy

Figure 3 depicts the stock Cray data paths (blue) and our user data paths (green). We provide a parallel data path for user data from data collection through aggregation, publishing to the system telemetry bus, and consuming from the telemetry bus. Shared monitoring infrastructure is limited to telemetry bus to avoid possible configuration corruption or data gathering perturbation of critical system services or data that feeds them. The reason for sharing the telemetry bus is that Cray is providing this infrastructure in a scalable way to provide a single data sync with a common output format for both Cray and system administrator supported system services as well as system user data consumption. Use of the system telemetry bus requires the bus to host topics other than what Cray is using, however, the configuration is straightforward and lives in a single file.

B. Building LDMS

While the packaged LDMS (LDMS v3) is available for use by anyone on the system, it is not the latest version available. We decided to build and use the latest version (LDMS v4) because it has a number of extended features with two in particular that we wanted to exercise as part of this deployment: the main one is the ability to dynamically modify a LDMS sampler's sampling frequency and have that change trickle down through the aggregators, the other is the ability to create test sets with arbitrary set configurations with respect to types and ordering of data. The latter we utilize for scalability testing of the telemetry bus (detail in Section V). Some issues in the build we discuss in Section III-F.

In order to write to the telemetry bus from our LDMS v4 aggregators we did need to port the Cray-provided `store_kafka` to LDMS v4, as mentioned in Section II-C. The port was straightforward though we did omit some Cray specific pieces that support some Cray specific LDMS sampler plugins. We expect to add these back in so that both Cray and user data collection can utilize the LDMS v4 infrastructure.

We were able to copy the full build directly into the our container image which we had copied from the original Cray-provided container and kubernetes [12] configuration to assist in creating our own kubernetes pod. We titled this new pod "sma-ldms-sandia-v4". Changes needed to be made to the `start.sh` script located in the root directory of the new pod in order to stop it from exiting when running multiple LDMS daemons. Cray's initial intended use of their pod was to have a single LDMS aggregator daemon running as they are only writing to a single telemetry bus topic. They wrote a cron script to enforce this which kills the pod if it detects more than one *ldmsd* process running.

In alignment with the design philosophy, we publish the additional metrics to different topics than the Cray collected metrics, which are read off by topic and inserted into the Postgres database. Further, to facilitate processing when reading off the telemetry bus, we publish each metricset type to a different topic. While a single LDMS aggregator daemon supports simultaneous aggregation of an arbitrary number of metric set types, it currently only supports a single instance of any particular store plugin. Also a single instance of the `store_kafka` plugin only supports publishing to a single topic. Thus to satisfy multi-topic data publishing we must run multiple LDMS aggregator daemons (one per topic). We also had to modify Cray's multi-*ldmsd* detector cron script to enable this mode of operation.

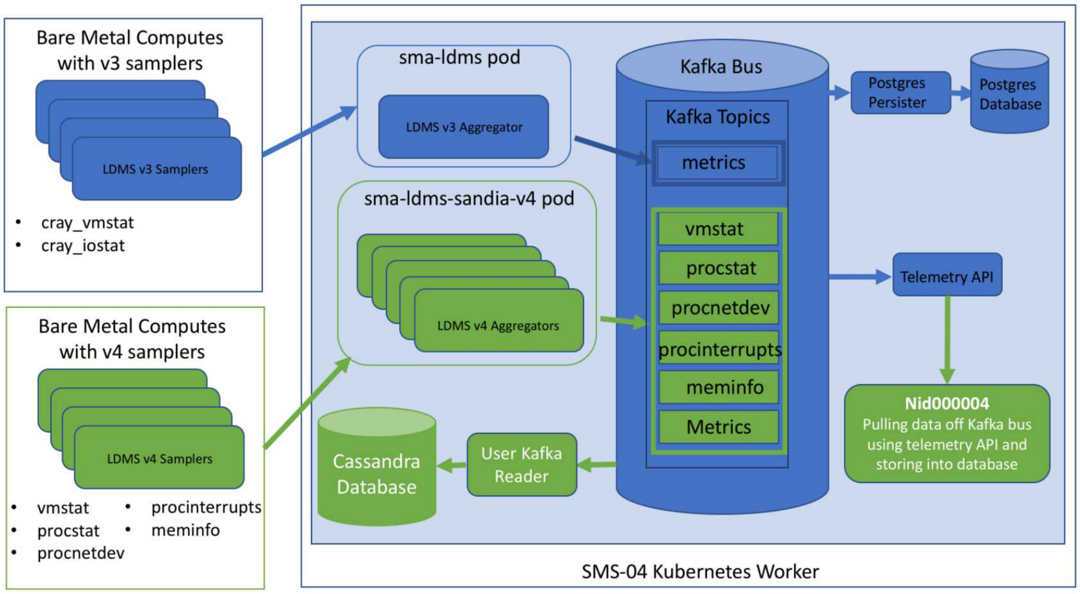

Figure 3. Extended configuration to support user monitoring within the Cray-provided monitoring architecture. Cray-provided data flow is in blue; extensions are in green. Multiple aggregators is a Cray-driven design choice to ensure reliability and availability of the Cray-provided metrics. Metric extraction from the telemetry bus was done both by direct reading from the bus and by using the telemetry API for comparison purposes. Locations of reading and storing from the bus are investigatory within the Preview System are are not intended to be how they would be used in a production configuration.

In our custom aggregator pod we currently run one LDMS aggregator daemon per sampler plugin type deployed on the compute node LDMS sampler daemons (5 total). We have created a separate configuration file per LDMS aggregator daemon. Each daemon writes to a kafka topic whose name matches the sampler plugin associated with the metricset being collected (e.g., the `meminfo` plugin's metricset is published to a kafka topic called "meminfo").

C. Compute node-side Deployment

On the compute nodes, as with the Cray LDMS sampler daemons, our user LDMS sampler daemons are run on the host OS. Cray utilizes a *genders* [13] based infrastructure to define the configuration parameters for their daemons. This can potentially simplify, for users, the process of separately configuring and starting multiple samplers and aggregators, particularly when only using the Cray-provided build (as opposed to the additional v4 build we used in this work).

We instead took the approach of writing a configuration file, a launch script, and a stop script. The same launch script, configuration file, and stop script are utilized for each node with the differentiation being population of some environment variables at run time to configure the host and component identities of the metricsets. A script to call the compute node start and stop scripts using the `pdsh` utility resides on the SMS04 node. In addition we wrote two scripts and associated configuration files to dynamically modify the sampling frequency of one of the sampler plugins. These scripts are also run via scripts residing on the SMS04 node using `pdsh` and simultaneously modify the same plugin in the same way on all compute nodes. In production deployments, these are configured and started more standardly as system services, of course.

The sampler plugins that we utilized for this work were `vmstat`, `meminfo`, `procstatutil`, `procinterrupts`, and `procnetdev`. These plugins sample all data from `/proc/vmstat` `/proc/meminfo`, `/proc/stat`, `/proc/interrupts`, and select interface information from `/proc/net/dev` respectively.

D. Application and Container Monitoring

Monitoring an application running on bare metal is no different that standardly doing so with LDMS. Typical metrics include node level data and application performance counters. For demonstration purposes we ran a toy application we call *memeater* which continuously allocates memory until it is killed by the OOM killer. This enabled us to verify the collection and the data storage.

Due to time constraints, we were limited in our ability to comprehensively investigate monitoring applications launched in containers. Our eventual plan is to investigate what information can be obtained outside of the container and what information needs to be obtained inside the container, and how to pass the latter to an aggregator. This includes assessing the ability to attribute resource utilization to processes within the container.

As an initial foray, however, we included monitoring of the container running the LDMS aggregator. We further limited the monitoring to collection of resource utilization data from files in `/proc` by pid. Explicitly, we modified the LDMS `dstat` sampler to take an optional pid (as opposed to dstat reading the ldms pid's data) in order to collect from `/proc/pid/iostat`, `/proc/pid/stat`, `/proc/pid/statm`. In order to determine the pid of the container, we needed to manually perform the commands shown in Figure 4.

We then started a sampler on the node hosting the aggregator container (SMS04) with the modified `dstat` sampler configured with the extracted pid.

As the locations of the container metrics are dynamic, there will be difficulties in creating a sampler that can follow kubernetes pods upon creation and destruction. Further complicating the matter is the idea that docker [14] images will be able to reside on any of the kubernetes workers in the cluster. This makes finding resource utilization for individual containers a challenge with our current sampler setup.

E. SMS node-side Deployment

By creating a new kubernetes pod for our own custom configured aggregators, we were able to leave the default sma-ldms kubernetes pod as it was configured from Cray. In addition, we were able to create our own kafka topics. While we are sharing the same telemetry bus, the creation of additional topics allows for us to keep separation between the user metrics and the native Cray metrics.

F. Difficulties Encountered

LDMS v4 was built successfully on nid000004 once all dependencies had been resolved by adding several packages to the build environment. The only dependency that presented a problem was openssl-devel as we were unable to locate a package compatible with the pre-packaged version. Our solution was to downgrade the packaged openssl in order to install a compatible openssl-devel RPM. This was a fairly easy fix, but it would be beneficial in the future to be provided with the full .iso that was used to install the image to avoid potential versioning issues.

The build and implementation of LDMS v4 within a container provided some difficulties. Most of the difficulties that we ran into involved the customization of our LDMS v4 docker image. We were able to take the existing default Cray provided sma-ldms container image and adapt it for our own application which saved us a significant amount of time that would have spend configuring the image otherwise. The overall workflow for LDMS implementation and configuration was noticeably more complex than we have experienced on previous systems. Cloning docker images, editing them, saving them, then loading them into the docker repository was a process that needed to happen frequently during our deployment. We felt that it was necessary to restart our pod frequently while making changes in order to verify that our changes did not affect the overall integrity of the default Cray monitoring platform. While restarting the pods is a relatively easy process, it does change the name of the kubernetes pods every restart. While it is simple to find the new name of the kubernetes pod, it does add to the complexity of use. We found ourselves typing the same few kubernetes commands frequently.

The currently provided `start.sh` script which limits the number of running ldms-like-named processes will have to be revised not only for supporting multiple concurrent instantiations, but also because it conflicts with supporting concurrent running of LDMS query tools, such as `ldms_ls` which is used for command line querying of LDMS daemons to obtain current metricset information.

While the current pid-based container monitoring worked sufficiently for the current investigation, supporting monitoring for dynamically changing containers will require a different process.

IV. Telemetry Information and Remote Access

Metrics are made available via the telemetry bus. Cray provides a Telemetry API to facilitate access to data on the bus, however, metrics can be read directly from the kafka bus as well. We explore both options in this section.

A. Telemetry API Definition and Operation

The Telemetry API is a read-only interface that allows messages from the Shasta message bus to be consumed outside of the Shasta Monitoring Framework. The telemetry API resides in a kubernetes pod inside of the SMA namespace and acts as bridge between the kafka message bus and clients outside of its namespace. This is illustrated in Figure 5.

User clients outside of the framework subscribe to a particular kafka topic from the API using an HTML request. Each HTML request results in a new kafka client that consumes messages from the user specified kafka topic. The API bundles, compresses and pushes messages received from the kafka client to the user client. Messages are pushed from the API to user clients using HTML5's Server Side Events (SSE) feature. The use of SSE enables the client to use its original HTML request connection to receive messages. No polling is required, the client simply blocks reading from its original HTML connection.

The user clients can scale through the use of kafka consumer groups. A number of user clients can be run under the same consumer group name enabling each of the clients to load balance the message stream from the API. Consumer groups also have the additional benefit of offset message management. The API keeps track which messages each consumer group has consumed. Therefore, a client can be restarted and resume where it left off automatically if it uses the same consumer group name and is restarted within the kafka retention policy time.

```
sms04-nmn:~ # docker container list | grep ldms

07ed9ebdd998        d8022eb6ca4e                                    /start.sh              19 hours
     ago        Up 19 hours                                   k8s_sma-ldms-aggr_sma-ldms-aggr-7
    cd68f6fb-mh5n7_sma_2420ae5f-60a2-11e9-8bac-0060dd470896_0

3fcfb162e9fc        k8s.gcr.io/pause-amd64:3.1                      /pause                 19 hours
     ago        Up 19 hours                                   k8s_POD_sma-ldms-aggr-7cd68f6fb-
    mh5n7_sma_2420ae5f-60a2-11e9-8bac-0060dd470896_0

sms04-nmn:~ # docker inspect 07ed9ebdd998 | grep Pid
             Pid: 186729,
```

Figure 4. Example commands to obtain the pid of the ldms aggregator container

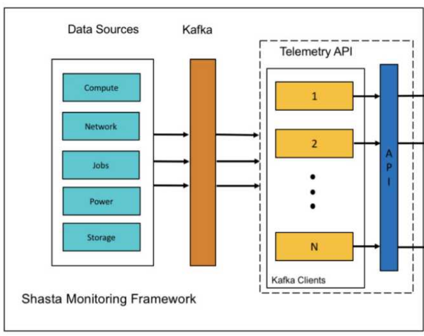

Figure 5. Telemetry API Message Flow. Use of the Telemetry API establishes the kafka clients which get data from the telemetry bus and make it available to the user clients.

Cray provided a pre-release version of the telemetry API and client to use on our Preview version 0.4.0 system. The telemetry API was recently released in Preview version 0.5.0 and some info may be found in [15].

B. Data formats on the telemetry bus

Determination of the format of the metrics on the telemetry bus can be generally arbitrary. Cray's `store_kafka` plugin pre-defines a few possible json-based formats which can be used. Configuration of the store determines which form is used.

One particular format has been specified by Cray for metrics which they will read off the bus and store into the Postgres database. This is called *cray_fmt* and it is shown in Figure 6. This consists of a json string per individual metric and contains the base information of timestamp, cname of the relevant component, metric identifier (e.g., MemTotal), and value, with other information.

Another format was included for diagnostic purposes, which writes all the metrics in a metricset together into a pretty-print, nearly-json structure, minus some required quotation marks, and with some integrated newlines. We revised these features for transport and parsing and will call it *metricset_fmt* (the original version is actually called *json* format, and is the default option in `store_kafka`) and it is shown in Figure 7. Note that nested lists have been reduced in size for clarity.

The advantage of the *metricset_fmt* format is there is *much* less overhead per metric. The *cray_fmt* has ∼500 bytes of overhead per metric:value. Note that in the native LDMS transport, even less data is transferred – for example, metric names are metadata in the native structure and only transported when changed.

C. Reading directly off the telemetry bus

One can also make calls directly to the kafka bus for reading off the data, for example: `kafka-console-consumer.sh --bootstrap-server kafka:9092 --topic metrics`. However this methodology does not provide the eventual load-balancing capabilities that use of the telemetry API and associated clients would.

We intend to use this method to compare the performance for reading directly from the bus vs using the telemetry API. We implemented a proof of concept analysis and visualization pipeline from an existing production environment but adapted it to use Cray's kafka bus and container management system. The final product is a web-based interface to view the compute node monitored metrics. The display provides a timeseries charts with Google Charts, where the data is either a metric for a single node or a reduction operation `(sum,min,max,etc)` across a group of nodes. The current implementation groups by all nodes or by a single node due to the lack of availability of job data. The data pipeline and resulting visualization are shown in Figure 8.

We built two Kubernetes pods and a standalone docker image.The general approach taken was to attempt to utilize unmodified docker images as much as possible and to mount any directories with customized files or configurations. This enables us to preform upgrades on services piecewise without the need to maintain our own customized images. The first Kubernetes pod is a simple python image with kafka support. This runs a python application that reads the kafka topics for the user metrics in *metricset_fmt* and performs an insert per data set into a Cassandra data

{metrics: {messages: [{metric: {timestamp: 1556290290104, name: cray_storage.cray_meminfo.
MemTotal:, value: 196442928, dimensions: {product: shasta, job_id: 0, service: ldms,
hostname: nid000003, component: cray_meminfo, system: compute, cname: c0-0c0s0n3 }}, meta
: {region: RegionOne, tenantId: f1aa39eb19d74f5c96c64e99838de3f7}, creation_time:
3386706919782612992}]}}

Figure 6. Format for Cray-specified metrics on the message bus (a.k.a *cray_fmt*)

{ metrics: { messages: [{ instance_name : nid000001/procstat, schema_name : procstat,
timestamp : 1556314203.012071, metricset : { component_id : 1, job_id : 0, app_id : 0,
cores_up : 36, cpu_enabled : 1, user : 33627332, nice : 34593, sys : 2377911, idle :
23065406125, iowait : 3637430, irq : 0, softirq : 59986, steal : 0, guest : 0, guest_nice
: 0, hwintr_count : 1324653743, context_switches : 1192575344, processes : 2918644,
procs_running : 1, procs_blocked : 0, softirq_count : 1108229508, per_core_cpu_enabled :
[1,1,1,...,1,1], per_core_user : [964457,989215,960860,...,884212,883872], per_core_nice
: [153,2590,2195,...,0,0], per_core_sys : [135318,175062,98275,...,9593,9650],
per_core_idle : [640452892,640464741,640657868,...,640836229,640849001], per_core_iowait
: [112678,164833,92849,...,117294,104172], per_core_irq : [0,0,0,...,0,0],
per_core_softirqd : [2409,2242,2895,...,690,687], per_core_steal : [0,0,0,...,0,0],
per_core_guest : [0,0,0,...,0,0], per_core_guest_nice : [0,0,0,...,0,0] }}]}}

Figure 7. Format for user-specified metricsets on the message bus (a.k.a *metricset_fmt*)

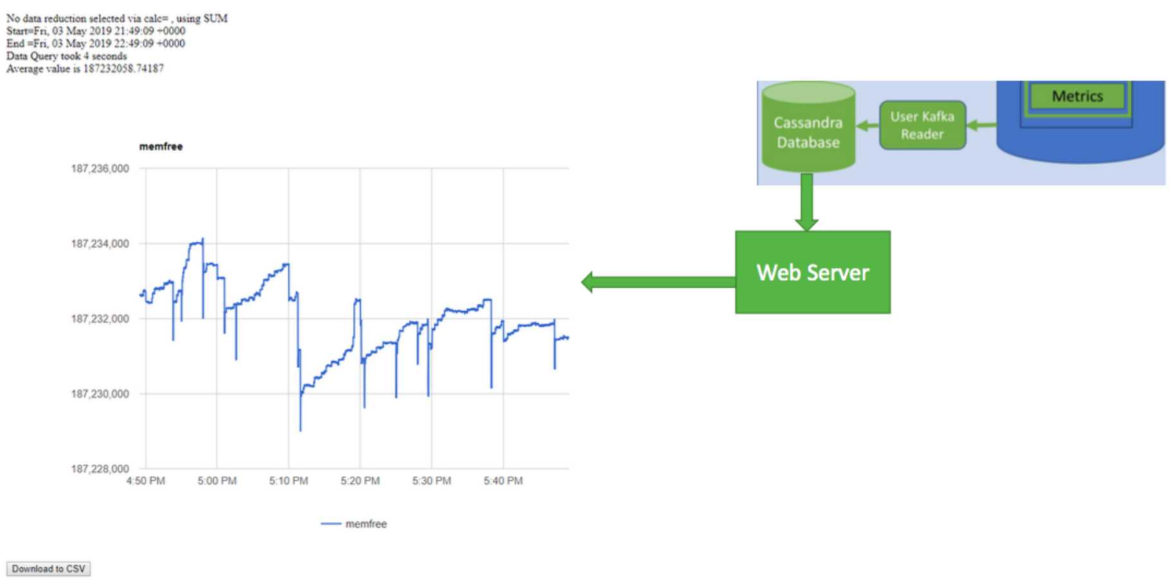

Figure 8. Graphical display using Google Charts (left). Webserver driven by the monitoring data inserted into a Cassandra [16] database. Components involved in this datapath (right); these are extracted from the full diagram of Figure 2

store. The second pod is a standard Cassandra docker image with the data directory mounted as a local device on SMS04. This enables persistence of the database across restarts. Finally, a docker image was generated to provide an Apache2 web server with PHP support and the DataStax PHP Driver for Apache Cassandra. That docker image was run on SMS04 to enable easy exporting of the web server port to external systems. The Web server contains php pages that process the URL and form Cassandra queries to acquire the metric data. Those results are added to a `google.visualization.DataTable` and displayed with `google.visualization.LineChart`.

D. Reading off the telemetry bus using the telemetry API

1) Telemetry API: The telemetry API resides on a kubernetes pod in the sma namespace on SMS04. We were given the docker image and kubernetes configuration from Cray ahead of release in order to further test the possible user functionality of the telemetry API. This telemetry API pod opens up a port on SMS04 that allows for external hosts with access to SMS04 to be able to access the API, and therefore pull data directly from the topic of your choosing on the kafka bus. In this case, the LDMS metrics data can then be stored in whatever format works best for the user.

2) Reading off the bus using the telemetry API: Cray provided us an example python script (`client.py`) that uses the telemetry API and assumes the *cray_fmt*. We enhanced it to parse the *metricset_fmt* to write out metricsets as `key,value` pairs in CSV format. Generally speaking, this reads metrics off the bus, performs a `json.loads()` of the data to parse it, and enable whatever subsequent use of the data is desired. Highlights of this are shown in Figure 9. Since each metricset is a different topic, it is straightforward to write each metricset to a different file. One could just as well have inserted the data directly into a database (as in the case where we read off the bus directly), or forward the data on elsewhere (e.g., to a named pipe).

E. Difficulties Encountered

In general, the challenges in the implementation came from a lack of familiarity of the containerized environment and new APIs used. Reading off the message bus via the Cray-provided `client.py` script that used telemetry API and adapting it to store to CSV was fairly straightforward. Similarly reading directly from the kafka bus and inserting into a database was also straightforward. While the installation of the database in the infrastructure presented some challenges, this was not intended to be an expected implementation.

V. Scale Testing of LDMS and Telemetry Bus

In this section we present details of our scale testing of the data collection, transport, and storage infrastructure provided with the Software Preview system. The LDMS component has been running on large scale production Cray systems for years and has been shown to scale to tens of thousands of nodes with multiple data sets, each containing hundreds of metrics, per node. The Software Preview system is the first to take the approach of interposing a data broker (kafka in this case) between the aggregation points and the data store. Thus we want to evaluate scalability of the system as a whole with the addition of this new infrastructure. This section describes our progress in scale testing the configuration with respect to LDMS sampler daemons running on compute nodes, the LDMS aggregator daemon responsible for aggregating data from all scale testing sampler daemons, the telemetry bus, and the data store for data being pulled off of the telemetry bus. Note that the LDMS aggregator daemon, telemetry bus, and data store are all hosted on our *SMS04* node.

Since the system Preview system only ships with four compute nodes, use of it for scale testing required emulating a much larger scale system by running a large number of LDMS sampler daemons per node, aggregating from all of those daemons, and pushing the high volume aggregate to the telemetry bus. This section describes how we were able to use this methodology to emulate 4,000 nodes each producing nine metricsets, with each set containing 650 metrics.

1) Compute Node Configuration: When running many LDMS sampler daemons on a compute node one of the problems encountered is the inability to find enough data sources with a large enough number of metrics to stress the monitoring system while not over taxing the compute node resources in collecting them. Note that one can only use a single instance of any particular sampler plugin on a single LDMS daemon and many of the plugins produce a relatively small number of metrics. The ones with a large number of metrics can incur significant node resource overhead in reading the data sources over many daemons. Our solution to this is the LDMS "test" sampler. The test sampler enables the user to define sets containing any number of scalar and vector metrics. An example configuration along with a subset of resulting metrics is shown in Figure 10.

On the compute nodes we created 1,000 distinct configurations for running 1,000 LDMS sampler daemons per node differentiated by listening port. The port range used was duplicated on each node and ran from 20001 to 21000. All configurations utilized the socket based transport. The metricsets for each daemon were the same: "set1" through "set9". Each set consisted of 100 integers, with labels of "metric1" through "metric100", and 10 integer vectors of size 10 to 100 in increments of 10 with labels of "array1" through "array10". All values are incremented by 1 on each successive sample interval. A script to start and stop all daemons is launched simultaneously on all nodes using `pdsh` from *sms04*. We ran these daemons for 10 minutes to evaluate the effects on the telemetry bus.

2) Aggregator and Telemetry Bus Configuration and Results: With respect to aggregating the test sets we utilize a single LDMS aggregator located in our aggregator pod on *sms04*. The aggregator is configured to collect from all 4,000 LDMS samplers every second and is configured to write to the kafka store using the topic "test_sets". We validated that the sets can be aggregated and stored directly to CSV files on the *overlayfs* file system. We first launched the aggregator and then the samplers running on the compute nodes. Running our revised telemetry `client.py` script we are able to pull the test_sets data off the kafka bus and write it in CSV format to another directory in the *overlayfs* file system. As can be seen in Figure 7, user metricsets are written to both the telemetry bus and storage in CSV format where each datum includes a colon separated label and value.

The data volume across the test sets is $\sim$7 GB/min to the telemetry bus and to CSV store. The rough calculation for this, over the 10 minute interval uses $\sim$ 11 bytes per scalar and $\sim$4 bytes per array value. This translates to 1100 scalar bytes + 2200 array bytes or 3300 bytes per set being written

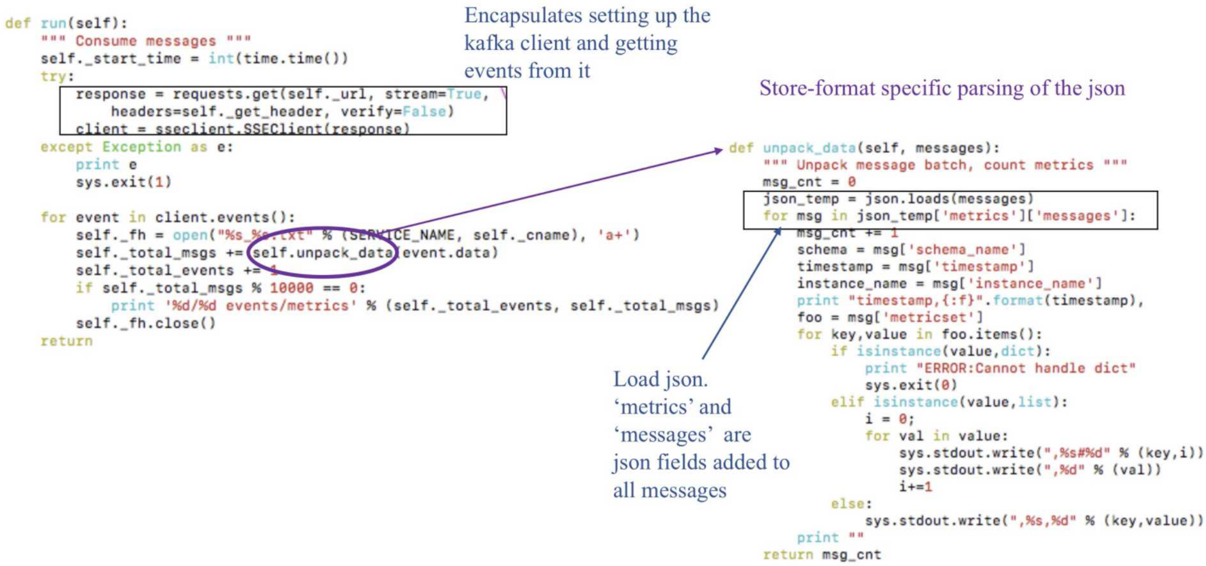

Figure 9. Components of the script which uses the Telemetry API to read metrics off the bus and write them out. Marked sections (counter-clockwise from top left) include: a) high level code which uses the API to set up a kafka client to read events matching a particular topic off the bus b) loop over events and hand them off to the local unpack data for parsing, and c) format-specific parsing of the json data. The user would principally have to change only (c).

```
config name=test_sampler action=add_schema schema=set4 metrics=meta1:meta:U64:0,component_id:meta:U64:320332,job_id:data
:U64:1,scalar1:data:U64:1,scalar2:data:U64:1,scalar3:data:U64:1,scalar4:data:U64:1,scalar5:data:U64:1,scalar6:data:U64:1
,scalar7:data:U64:1,scalar8:data:U64:1,scalar9:data:U64:1,scalar10:data:U64:1,scalar11:data:U64:1,scalar12:data:U64:1,sc

alar97:data:U64:1,scalar98:data:U64:1,scalar99:data:U64:1,scalar100:data:U64:1,array1:data:U64_ARRAY:1:10,array2:data:U6
4_ARRAY:1:20,array3:data:U64_ARRAY:1:30,array4:data:U64_ARRAY:1:40,array5:data:U64_ARRAY:1:50,array6:data:U64_ARRAY:1:60
,array7:data:U64_ARRAY:1:70,array8:data:U64_ARRAY:1:80,array9:data:U64_ARRAY:1:90,array10:data:U64_ARRAY:1:100
```

Figure 10. Selection of configuration of test sampler which supports arbitrary set specifications and sizes to support scale testing. Specification of scalar and array components highlighted.

every second (200KB/min). Note that the data values start out small (all values are initially 0 and increment by one each iteration) and are written as strings to the telemetry bus so the metric value byte count actually averages about 3 bytes over the 10 minute run interval. Given that the telemetry bus is configured to only store 10GB total, there is an obvious mismatch between the rate of data ingest and storage and age-out policy. This would be significantly worse had we been using the *cray_fmt* which would have incurred an additional ∼11 GB/sec of overhead for this data.

Over the 10 minute test interval we lose some of the data. As of this writing we have not concurrently written to both the native LDMS csv store and the kafka store to identify if the same data would be missing in both. We have only run this experiment twice, once writing the output to CSV and the other just to standard out. In both cases data seemed to be flowing for the entire test but the kafka pod subsequently died. It is not clear yet if this is what caused it to die or if it was coincidence. We plan on conducting more of these experiments while doing some low level monitoring of the kafka bus to identify what is causing the problems. Note that the current retention policy has the data retained for four hours which is probably not realistic for a large scale system. Also, in the production system Cray intends to make the kafka bus distributed and implement load balancing which should enable it to scale out much better.

VI. Future Work

As we continue to investigate monitoring possibilities on the Software Preview system we will be looking more into the following issues:

A. Full port of LDMS to v4

We will be working to coalesce the Cray-enhancements to LDMS v3 into LDMS v4. The most significant Cray-developed change is a change to the format of the metricset *schema* which is the structural definition of the metricset. Cray integrated into the schema a nested array of schema which could be associated with the same global variables represented in the top schema. The need for this construct does not apply to any of the samplers presented in this work, but could apply to representing, for instance, data from multiple filesystems to be included one per element of the array. In the `store_kafka` initial implementation, the existence of this structure is assumed for Cray-provided metrics. In LDMS v4, there is a related construct, *metricset groups*, which provides some of the same flexibility. We will be looking at how to best integrate the two designs.

B. Genders-based configuration

The Cray genders-based configuration is intended to simplify the configuration of the various samplers and transport topology. In future work we will investigate the ability of genders-based configuration to support the expected flexibility and variability in configuration needed on complex systems.

C. Application Monitoring

We will include monitoring within containers in our next steps. This will include more comprehensive monitoring of the data that we can get from outside of the container, as well as getting data from the container out and into the data stream. We will be assessing what other data and metadata needs to be included in container-based LDMS metricsets. For example, currently component id and instance information is carried for node-based metricsets. Application identification is provided in LDMS via a SLURM spank plugin that enables LDMS to include the current job id as a metric within the set.

D. Scaling out the Telemetry Bus

Cray intends to support load balancing of transport and access capabilities of the bus to support increased data load. We will expand the scaling and performance study when these additional capabilities are made available. It is important to assess to ability of the telemetry bus, and the required supporting hardware, to support the numbers and rates of metrics that we are collecting even off of current systems (e.g., NERSC is currently reporting collecting 16TB/day of LDMS data) particularly in the face of the metric size expansion in the current json formats.

E. Completing Web Based Monitoring Environment

The way these interfaces are typically used involve identifying features in broader datasets and drilling down on subsets of that data to define the source. This process typically utilizes node metadata such as job id, application id, username running workload, and group the user is associated with. When this type of data is available, we will stream the data via the Telemetry API and insert it into the datastore to be applied in data reduction functions.

In addition, we failed to export some of the service ports from the Kubernetes pods to the external networks. This was the intention for accessing the datastore. It would enable us to operate the web server on a remote resource and reduce the number of forwarded connections necessary to access the web graphs. In this prototype, since the web service was on SMS04 via docker, we needed an additional persistent port forwarding service on the gateway system.

For very large data volumes, we will likely need to spool the incoming datasets and bulkload it into the datastore to reduce transaction overhead. This can be added to the python/kafka pod in a simple fashion.

VII. Conclusions

In this work we extended the Cray Software Preview System to include user-driven metrics into the telemetry stream. These were obtained via additional LDMS instances on the preview system. We extracted the metrics using the telemetry API, demonstrating how metrics from the preview system could be pulled off and fed into site-specific storage. Further we assessed the performance, overhead, and usability of the monitoring system design through the implementation and scale testing of the user metrics set up. Finally, we provided additional assessments of the general usability of the system, through the mechanics of building, configuring, and installing the necessary components and services in the Cray Preview Environment.

Interaction over the Software Preview System is the sites' opportunity to feedback experiences and recommendations to Cray. The monitoring system architecture provides improvements over past implementations for extensibility and data access. This is a timely opportunity for the sites to interact with Cray to ensure that the monitoring system has the potential to appropriately scale and to incorporate the necessary resilience features for enhanced monitoring on the next generation large-scale production systems.

Acknowledgements

Sandia National Laboratories is a multimission 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. The views expressed in the article 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, "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.

[2] V. Ahlgren *et al.*, "Cray System Monitoring: Successes, Requirements, Priorities," in *Proc. Cray Users Group*, 2018.

[3] Cray Inc., "Cray Software Preview System Administration and User Guide (0.4.0) Rev A," 2019. [Online]. Available: https://c-compass.atlassian.net/wiki/download/attachments/101449729/Cray_Software_Preview_System_Administration_and_User_Guide_040_Rev_A.pdf?api=v2

[4] ——, “Cray Software Preview System Installation Guide (0.4.0) Rev A,” 2019. [Online]. Available: https://c-compass.atlassian.net/wiki/download/attachments/101449729/Cray_Software_Preview_System_Installation_Guide_040_Rev_A.pdf?api=v2

[5] The Apache Software Foundation, “Apache kafka: a distributed streaming platform,” (Accessed 2019). [Online]. Available: https://kafka.apache.org

[6] Grafana Labs, “Grafana,” (Accessed 2018). [Online]. Available: http://grafana.com

[7] G. Bauer, J. Brandt, A. Gentile, A. Kot, and M. Showerman, “Dynamic Model Specific Register (MSR) Data Collection as a System Service,” in *Proc. Cray Users Group*, 2016.

[8] Cray Inc., “Aries Hardware Counters,” Cray Doc S-0045-20, 2015.

[9] J. Brandt, E. Froese, A. Gentile, L. Kaplan, B. Allan, and E. Walsh, “Network Performance Counter Monitoring and Analysis on the Cray XC Platform,” in *Proc. Cray Users Group*, 2016.

[10] Cray Inc., “Using the Cray Gemini Hardware Counters,” Cray Doc S-0025-10, 2010.

[11] ovis hpc, “LDMS github (ovis-hpc/ovis),” (Accessed 2019). [Online]. Available: https://github.com/ovis-hpc/ovis

[12] The Kubernetes Authors, “Kubernetes: Production-Grade Container Orchestration,” (Accessed 2019). [Online]. Available: https://kubernetes.io

[13] CHAOS Development Team, “chaos/genders,” (Accessed 2019). [Online]. Available: https://www.github.com/chaos/genders

[14] Docker, Inc., “Docker: Enterprise Container Platform for High-Velocity Innovation,” (Accessed 2019). [Online]. Available: https://www.docker.com

[15] Cray Inc., “Cray Software Preview System Administration and User Guide (0.5.0),” 2019. [Online]. Available: https://c-compass.atlassian.net/wiki/download/attachments/138870789/Cray_Software_Preview_System_Administration_Guide_050.pdf?api=v2

[16] The Apache Software Foundation, “Apache Cassandra,” (Accessed 2019). [Online]. Available: https://www.cassandra.apache.org