Significant Advances in Cray System Architecture for Diagnostics, Availability, Resiliency and Health

Christer Lundin
Research & Development
Cray
Seattle, USA
clundin@cray.com

Stephen Fisher
Research & Development
Cray
Seattle, USA
sfisher@cray.com

Abstract— **Keeping the system healthy and able to run compute jobs is one of the primary goals of the Shasta architecture. In the end, no matter how complex the infrastructure is, the main reason the system exists is to allow users to run customer workflows; everything else is really just to support this capability.**

Keywords — system health, resiliency, diagnosability, ava ilability, redundancy, serviceability

I. Introduction

System health can be viewed as a set of services that provides various capabilities for maintained health that span across the spectrum of components and subsystems that make up the Shasta system. System Health consists of diagnosability, which is the collecting and reporting of the system's health, resiliency, which is the determination of what automatic or manual action to take to fix the system given the specific diagnosis information, and serviceability, which covers how to fix and maintain system health.

The following diagram shows a high-level overview of system health in Shasta:

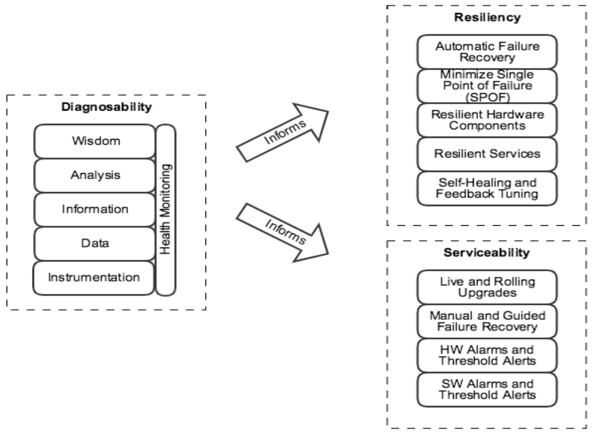
Diagnosability
Wisdom
Analysis
Information
Data
Instrumentation
Health Monitoring
Informs
Informs
Resiliency
Automatic Failure Recovery
Minimize Single Point of Failure (SPOF)
Resilient Hardware Components
Resilient Services
Self-Healing and Feedback Tuning
Serviceability
Live and Rolling Upgrades
Manual and Guided Failure Recovery
HW Alarms and Threshold Alerts
SW Alarms and Threshold Alerts

Fig.1. System Health Overview

II. System Health, what is it?

This section defines diagnosability, resiliency, and serviceability.

A. Diagnosability

The basic dictionary definition of “diagnosability” is “the condition of being diagnosable” (Wiktionary definition). Following that, we see that “being diagnosable” means “having a cause that can be determined”.

An illustrative analogy to diagnosing what is wrong with a large computer system is that of diagnosing what is wrong with a human being (which is also a large system, albeit a bit more complex than a supercomputer).

When a patient goes to the doctor with some symptom, say a joint that will not move correctly or a high fever, at that point in time there are several potential possibilities that might be the root cause. The main job of the doctor is to use various techniques to gather information, analyze that information, run tests, and narrow down those possibilities to provide a diagnosis and course of action. One of the first things they will have the patient do in this case is to fill out information on health history and recent health events (e.g. recent injuries, history of illness, trend information).

This can involve measuring streams of constant data coming from the patient, such as heart rate, O2 absorption, blood pressure, and other telemetry. It can also involve running diagnostic tests, as well as taking point in time snapshot measurements of the patient’s various components.

This troubleshooting process is an adaptive one that takes the results of the measurements and tests, analyzes them, and applies them to the next step in the decision process in narrowing down the root cause of a given problem. During measuring and testing, other problems may be found that can range from small items to major ones. As these are encountered during troubleshooting, a decision can be made on whether to fix that problem then or defer it. Also, at times a doctor's analysis will result in 'do no harm', take no action, or avoid taking destructive actions if the source of the problem is not well understood. We are looking for diagnostics not to just find problems and suggest solutions but to also provide context. If a solution has side effects (for instance taking a medicine with significant adverse reactions) and the problem is not known, the negative consequences or collateral impact/damage may increase.

Similarly, in the case of a large computer system, this same basic overall approach can be used. We can instrument the system with the capability to generate data, we can develop tools to measure this data, we can develop tests that measure the functionality and health of the system, we can use the data streams to derive information from them, we can develop tools that interpret this information, and we can provide the ability to display this information in a form that can be used to diagnose the system. For diagnosing systems, the equivalent to filling out health history and recent health events is log collection, log analysis, system event collection, and critical event/change notifications/recording. To add to the analogy with diagnosing a human, if a solution has side effects (e.g. rebuilding a compute node, giving more/unbounded compute capacity to an app, rerunning a broken app) and the problem is not well known, the negative consequences or collateral impact/damage may increase.

Some other analogous aspects between diagnosing a human and diagnosing a computer system are:

- Annual physicals – Proactive diagnostic checks on full system (e.g. running diagnostics when no failure/fault is expressed)
- Crisis/acute management – Body/system is sick, which includes reactive diagnostic tests to try and find the source of the problem
- Ongoing health monitoring – E.g. FitBits, health monitors that are proactively and passively monitoring the system

The intent here is to highlight that we always want to keep the system healthy, not just wait for faults or downtime to diagnose and address system health.

One other aspect of this analogy is the potential to leverage live or offline data from large groups. In the case of a human, this can take the form of population study and analysis (for example rates of recurrence among cancer patients in remission); in the case of large systems like Cray, this can take the form of the services/support data warehousing and higher-level support analysis (even from other customers in an anonymized way) to help identify issues. The problem space is not strictly local to one system, one body at that point, one can build up wisdom about specific problem occurrences into a knowledge base.

B. Resiliency

Resiliency is the ability of a customer application or system and its services to recover from failures and continue to function. It is not about avoiding failures; it is about responding to failures in a way that avoids downtime or data loss. The goal of resiliency is to return the application or system to a fully functioning state following a failure, while minimizing the impact on running workloads.

As an example, system resiliency is the ability of an application or a service to react to a problem in one of its components and still provide the best possible service. This is even more important as software is implemented more and more across multitier, multiple-technology infrastructure (architectural layering). Always-on architecture enables resiliency through several layers of architecture constructs, like infrastructure as a service, platform as a service, and software as a service. Similarly, in a team sport like football, a coach many times can see if a player is injured and can substitute the injured player with a healthy player. Sometimes it can be hard for the coach to tell if a player is healthy or not. To be a resilient football team, all players needs to help proactively watch for unhealthy players, by checking in on each other, asking if someone is hurt and how bad. Players themselves can inform if they are injured and even take themselves out of rotation to help their team.

System resiliency for a modular, service-based architecture like Shasta allows for decoupling the components and services from each other, which improves resiliency since it allows the system to load-balance and distribute services across multiple hosts.

There are some important aspects and types of resiliency required for this type of architecture:

- High Availability – An application or service can recover or continue to run without significant downtime. The application is responsive, and users can connect to the application and continue to use it.
- Reliability – The probability that an item will function (compile, compute, and access data) without failure under stated conditions (network, power, etc.) for a specified amount of time.
- Checkpoint and Restart or Backup and Restore – Protect against hardware errors (e.g. processor, memory, network), bad code, other service failures, and accidental deletion of any type of data, such as application or configuration data.
- Disaster Recovery – The ability to continue operation from rare but major incidents, like large-scale failures such as service interruptions that affect an entire region. Disaster Recovery starts when the impact of failures exceeds the ability of the High Availability design to handle it.
- Incidents – This may be a subcategory, but it is also important to have resiliency in control systems to allow the system to defend, protect, and/or respond to a malicious cybersecurity attack, or to an accidental manual operation that can put the entire system or parts of the system in a bad or harmful state, potentially impacting the confidentiality, integrity, and availability of information.

Resiliency provides higher availability and a lower mean time to recover from a failure. Resiliency does not just happen, however; it must be designed and built in from start.

What is the difference between reliability and resiliency and why does it matter? Reliability is a design engineering discipline that applies scientific knowledge to assure that a system will perform its intended function for the required duration within a given environment, including the ability to test and support the system through its total lifecycle. A reliable system is essentially a system that functions as it was designed and for its intended purposes, when it is expected to, and wherever it is being used. That is not to say that every component must operate flawlessly 100 percent of the time. Resiliency is recovering from failures, while reliability is the outcome. i.e. reliable operation is the result of a system that is designed to be resilient.

There is a distinct difference between reliability and availability: reliability measures the ability of a system to function correctly, while availability measures how often the system is available for use, even though it may not be functioning correctly. As an example, a service may run forever and have an ideal availability, but it may be unreliable, with frequent data corruption.

C. *Serviceability*

Serviceability provides continuous update strategy for the various Shasta system software, hardware and firmware components. Serviceability is after the fact, there is a problem and restoring the product into service. Serviceability provides features that facilitate more efficient product maintenance and reduce operational costs and maintains business continuity.

There are two types of updates:

- Live update – an update can be done without taking the component offline or losing access to the service provided by a component. There should be no loss of system function during a live update or any rollback that occurs.
- Rolling update – an update that affects a component requiring that horizontally scaled components to be taken offline one at a time. The components can then be "rolled" through, one failure domain at a time, until the entire system has been updated. NOTE: that during this process, the entire system should never have to go down at any one time. There is enough resiliency built in to the system to be able to continue some level of workload operations while upgrading or repairing hardware, software, or firmware modules.

Serviceability can be broken into the following general areas:

- Hardware and controller updates
    - Cray provided firmware updates – primarily embedded controller firmware (Mountain components – chassis controllers, node controllers, Rosetta blade switch controllers, FPGAs, microcontrollers, etc., River - Rosetta TOR switch controllers) and Cray blade BIOS updates
    - Vendor provided firmware updates – primarily PCIe cards such as NIC firmware, as well as the BIOS and BMC on commercial off-the-shelf (COTS) servers, and other River component firmware such as Ethernet switches, iPDU's, direct liquid cooled (DLC) doors and rack equipment, and other River hardware. Some of this is done as part of the host OS running on compute nodes, but there are also peripheral firmware updates that are not done as part of the host OS image, rather they are done via Redfish or other OOB mechanisms.
- Management infrastructure components
    - Non-compute node bare metal kernel and host OS updates/rollbacks
    - Kubernetes core orchestration component stack updates (API server, kubelet, etcd, etc.)
    - Cray provided system management services updates, including the update and rollback of data within these services
    - High Speed Fabric (HSF) / High Speed Network (HSN) component updates
- Managed services components
    - Shasta Linux software stack updates on compute nodes (host OS premium and standard). Includes peripheral firmware updates done as part of the host OS image on Mountain or River.
    - Parallel file system storage component updates (Lustre file system, ClusterStor storage)
    - Managed plane services (LNET router, DVS, etc.)

III. Why Do We Care?

A. *Changes in HPC Industry System Requirements*

One of the primary aspects of keeping a system healthy is making a system highly-available, just like scalability, by adding more resources of anything on demand. Other possible aspects for resiliency that also need to be accounted for, such as the ability to have an entire system in the same or even a remote geographic location in a disaster recovery scenario. Having a more modular architecture like Shasta allows Cray to offer solutions by replacing or adding modules (or components) that meet customers needs. Decomposing system health requirements for components in the system is important for many reasons. It allows more flexible options in providing diagnosability, resiliency and serviceability for what is important to customers. It is also important with a modular architecture to be able to be able to replace and upgrade different parts of the system without taking down the entire system which would impact serviceability.

There are several critical shifts in the HPC industry that help maintain system health, and Shasta products are designed to take advantage of these, as shown below:

B. *Extensible System*

Shasta products are designed for extensibility, following the principle of separating work elements into comprehensible modular components. These components can be customer-modified subsystems or experimental components (e.g., a custom compute kernel or image, a third-party management system, or different storage vendors) that are not directly controlled and tested by Cray. Cray can only control and know what is working for components provided or tested by Cray; there is a white-listed combination of versions of components dependencies that defines the boundaries. As an example, the Programming Environment (PE) container may have certain Operating System (OS) API. A custom compute kernel may cause a failure in the PE when an API it depends on is not available. This failure could be detected through monitoring and/or event logs, or through diagnostics, to identify if the problem was caused or not by a component provided by Cray. Since the PE runs on several different OS versions, Cray can only support those that it certifies, but the customer can still extend the system.

The Bill of Material (BOM) provides the complete inventory of all components provided by Cray and Cray-supported third-party components (with potentially several versions of backwards compatibility) that are tested and verified by Cray. All other components that are of the "unknown" category with an expectation that resiliency and failures can be unpredictable.

Shasta is an extensible system. E.g., it has:

- Different storage systems from multiple storage providers
- Replaceable system components (e.g., third-party compute/OS, system management, telemetry, etc.)
- RESTful interfaces that can be used to customize or modify the management system behavior.

C. *Decoupled and Layered System*

Decoupled code modules and layering provides the ability and flexibility to upgrade or update components independently to meet customer needs in a timely fashion. Customers can adopt new functionality and value from Cray on their own, without reinstalling or taking down the entire system. A good example is the Programming Environment (PE), which is independently released on a quarterly cycle. Customers can choose when to deploy it and what version to use.

Shasta is a decoupled, layered, and modular system:

- Separating management and managed system
- Variety of compute workload managers
- Monitoring and telemetry at different layers from kernel and hardware to components
- The ability to configure, upgrade, and update components independently

IV. Maintaining System Health

A. *System Health Awareness*

The goal of being able to monitor and diagnose a problem in any system is typically not just for awareness, although that is indeed a required first step. It is so that analysis can be done on the severity and impact of problems and figure out whether and how to effect automatic or manual repair actions on it. In some cases, these are automatic repair actions, which is where diagnosis ties into system resiliency and availability. In other cases, these are manual repair actions, which is where diagnosis ties into serviceability. In our ideal model, for any operation that has a well-known cause the system should be resilient to the repair/change (e.g. rebooting a node will not wedge the app or system), and automatic recovery is preferred. Manual should be the goal only when a human decision is involved (whether that be scheduled downtime, a disruptive change, or a risk analysis is required based on human/business need). In Shasta we aim for automation whenever possible and make this decision point of whether to effect automatic or repair a strategic one.

As described in the earlier example with a doctor diagnosing a human, he may find several problems in the course of taking measurements and running tests. Some of these may be directly related to the reason to go to the doctor in the first place, some may be ancillary, and some may be completely unrelated. Having knowledge of all of these is a useful thing, however only some of them may be the root cause of the initial symptom. And some of the data found during the diagnosis activities may well lead to treating problems that the patient was unaware of before.

An important thing to note with this analogy is that the expression of symptoms cannot always be traced back to a 'single root cause'. For example, if a patient have acute neck pain, a high fever, and is feeling lethargic then the patient could have viral meningitis (which can be very serious) OR the patient could just have a cold and have coincidentally strained his neck during exercise. When we diagnose based on the composite of the symptoms, we may inflate the impact of issues or miss root cause due to secondary manifestations. So, a caveat with this approach is that it is iterative and requires analysis at each iteration.

B. *Layered Levels of Health*

Likewise, when troubleshooting a large computer system, several problems may be found along the path of diagnosis. Sometimes these problems have little to no correlation to the initial problem which caused someone to be in diagnosing the system in the first place. But sometimes they do, and the act of troubleshooting leads to the discovery of other underlying problems that must be fixed first in order for any more meaningful diagnosis to be done. This highlights another aspect of diagnosability and monitoring, that of tiered or layered levels. For example, passive diagnostics may run at a high level (like taking a temperature), then more active diagnostics can be run to drill into specific concerning areas to gather more detail. In crisis situations, a disruptive/destructive diagnostic may be run - like a much more CPU intensive test that requires offline or non-compute friendly analysis and job interruption. The main goal is to start with passive tests, then move to more active test, and finally move to disruptive/destructive tests as needed, with diagnostic tools and services at each layer to help diagnose issues.

C. *Comprehensive System Dashboard Readout Support*

Another one of the primary goals of system health within Shasta is to provide support for a complete readout of all failed components, along with an estimate of the actual and potential impacts. A common example of why a problem is that Cray has seen many times in their past large supercomputing systems around the failure of network hardware (cables, NICs, switches, etc.). Even though the individual piece of failed network equipment may not seem on the surface directly related to some bigger question of, for instance, "why is my job running slow", the systemic effect of this network component failure across running jobs can ripple. Based on the knowledge built up over time from service and support, it can be shown that if running in a system with known failed components, there are interrelationships in a complex system that can cause unpredictable results. So, the first order of business and why this is one of the primary initial goals in Shasta is to monitor the system for known errors, and provide the data needed to sort and repair known system failures before trying to debug larger more systemic issues. System health monitoring and its associated role in system diagnosability are crucial components for increasing system resiliency. Just like the team sport example described earlier, monitoring is needed to detect and prevent problems before they occur or as they are happening, while diagnosis gives feedback on current problems or ones that have already happened.

Another software specific example in the Shasta architecture might be the failure of a service that is common to several other services, like the API gateway. Having a real time readout of system services and the overall system management is crucial to system health.

D. *Improved System Troubleshooting and Service Dependency Knowledge*

Note that if the cable/NIC/switch (or any faulted component) is failed, we do not want to just know all the current failure components, however. We also want to know the relationships between the various system services (user context, tenant, services, host platforms, connections between them). If we have that full relationship graph, or even an approximation, then a human or even an analysis engine can walk the graph and source the common cause. It reduces the need for sleuthing or having a human connect all the dots. We know which components are connected to which, and since the Shasta system is much more finite state machine than a human body, simply showing health states on various services and relationships would likely surface the issue more quickly. This allows us to provide high-level diagnostics, and then drill down deeper to specific targeted areas.

In addition to this, a major goal of the Shasta system diagnosability attributes is to negate the requirement to move components around in order to diagnose a problematic or broken part. This is a common troubleshooting mechanism in Cray systems today, but is very time consuming and disruptive, and the goal is to be able to identify and replace components without using this approach except as a last resort.

E. *Cray Service and Support Requirements*

The new Shasta architecture and product line will support many types of processors, storage systems, diverse Compute/OS platforms, new system management, and new monitoring capabilities. Shasta is designed with high-reliability components and targeted redundancy to minimize job and system interruption. Job recovery options let administrators and/or users restart failed jobs with minimal interruption. The Shasta infrastructure will take advantage of technologies that keep jobs from failing when they lose nodes. It will also support the use of monitoring data to support predictive failure analysis, which will allow the system to proactively take failing resources offline while keeping the system running. Hot swapping and an easy-to-service modular hardware design allows either Cray or customer technicians to fix failed components without a complete system interruption. In most cases there should be no interference, only a very minor degradation of the total system service, usually completely not affecting most running applications (only affecting one or two).

Capabilities in Shasta include the following:

- The ability for system administrators to upgrade or repair without blocking system management for components like the compute and managed applications
- The ability for an Application User to provision different compute images that already exist
- Compute images can be upgraded without impacting other running workloads on the nodes
- Containerized applications that can easily be recovered, developed on a laptop, and run as part of distributed workflows
- Security patches, hot fixes, and firmware updates that can be applied without shutting down the entire system (live upgrades)
- Shasta system software that automatically recovers from non-responsive nodes
- A Cray Slingshot interconnect that automatically detects defective links, and reroutes without affecting operations or requiring a reboot

F. *Requirements due to Industry Standards and Competitive Offerings*

As anyone that has seen a modern NOC or SOC (network operations center or system operations center) can verify, the capabilities of industry standard OSS and commercial system health monitoring and management tools in the enterprise data center have grown tremendously in the last decade. There are literally hundreds of tools out there that can easily be set up to monitor and manage various aspects of systems and their components, if the systems and components provide some form of programmatic access to the monitoring data they generate. Most data center operations personnel have come to expect the ability to integrate with these tools, so this drives

Cray to improve the ease of integration with these industry tools.

Another strong driver for Cray to integrate a significantly better system health monitoring solution into Shasta is competitive disadvantage. While Cray is a leading competitor in HPC, is not the only competitor in the market. Several of our key competitors are also pursuing system health enhancements, and actively market integrated solutions for easier remote and self-support. Competitor tools broadly advertise client benefits such as: "decreased call duration, optimized part usage, reduced time to resolution and system downtime, fixing it right the first time, and reduced onsite service calls", and "improved problem diagnosis, accelerated parts dispatch, and saving customers time and expense" by enhanced diagnosis, automated support cases, and automatic parts dispatch.

G. *Customer Driven and Defined Requirements*

But the #1 reason for Cray to make significant system health advances in Shasta is that customers are both expecting and demanding it. They are requesting this directly in their RFP's, with a common theme being the requirement that Cray provide an automated mechanism to diagnose the system down to the field replaceable unit (FRU) and diagnose the system in a timely manner without having to rely on hardware swapping as a primary isolation mechanism.

It is also a large part of the drive to exascale systems. Cray systems are large, complex, and costly. Most customers cannot fail over to a backup supercomputer when the primary one experiences an outage, and the impact of hardware failures grows as we increase scale. According to a 2017 journal article on resilience techniques for exascale computing platforms from IEEE ("An Analysis of Resilience Techniques for Exascale Computing Platforms"[viii]):

"As the computing power of large-scale computing systems increases exponentially, the failure rates of these systems increase exponentially as well. While current large-scale computing systems experience failures of some type every few days, projection models indicate that the next generation of these machines will experience failures up to several times an hour."

As failure rates increase, system resiliency must keep up, and while diagnosability cannot prevent hardware failure, it may be able to detect a problem before it completely fails and provide the data needed for the system resiliency features to kick in and keep the system operational for running workloads.

As part of overall system health monitoring, Cray has worked with several of our customers for some years running as part of the HPCMASPA (HPC Monitoring and System Performance Analysis) workgroups at IEEE Cluster Conference to help improve diagnosability of existing systems.

Some example conclusions from this research and analysis effort among numerous customer participants, there are several concrete requirements for next generation Cray Shasta system health, as shown in Table I in the document above[vii], and summarized below:

1) System must provide proactive notifications to users of system condition assessments.
2) System must support scheduling and job allocations based on application and resource state, including more fine-grained and dynamic resource allocations and task mappings.
3) Useful responses to system conditions requires increased analysis capabilities and more complex interfaces to schedulers and component/subsystem controls.
4) Need to access and integrate a variety of data sources and types: numeric and text, raw data, derived data, test results, analysis results, off-platform (e.g., facilities). Data can come from the system, applications, and external sources.
5) Vendors need to provide in depth documentation of capabilities and user accessible APIs for reading data that is already being produced so that sites can collaboratively develop and share tools. Vendors need to enable abstractions that facilitate sharing across multi-vendor platforms. Vendors should provide well-documented interfaces for accessing raw data at maximum fidelity with the lowest possible overhead. Vendors should expose all possible data sources for all possible subsystems.
6) All monitoring system capabilities should be production capabilities and documented, exposed, and supported as such. Extensibility and modularity are fundamental to support evolutionary development of these capabilities.
7) Tools to transport and store the data in native format are highly desirable. System should support all potential data sources, to include traditional text (e.g., logs), numeric (e.g., counters), diagnostic test results, and application performance information.
8) Easy access to historical data and the ability to access historical data in conjunction with current data is required. As with the storage of application data, all storage does not have to be equally performant; hierarchical storage models with the ability to locate and reload data as needed are desirable.
9) High dimensional and long-term data need to be handled in analyses and visualizations. Visualization interfaces and tools should facilitate easy development of live data dashboards.
10) Reporting and alerting capabilities should be easily configurable. These should be able to be triggered based on arbitrary locations in the data and analysis pathways.

V. Shasta System Health Strategy

Shasta is a complex but finite system. Because it is a complex system, diagnosability, resiliency and serviceability have to operate within the architectural complexity of the system. Diagnosability happens as a result of understanding the architecture of the system, measuring the function of that architecture, comparing that to the observable state of the system, and drawing inferences and conclusions based on that information. This follows the data, information, knowledge, wisdom hierarchy (DIKW hierarchy), as shown in the diagram below. To diagnose a complex system like Shasta, we collect generated data, warehouse it, and build analytical utilities that can help diagnose both current actual and future potential problems.

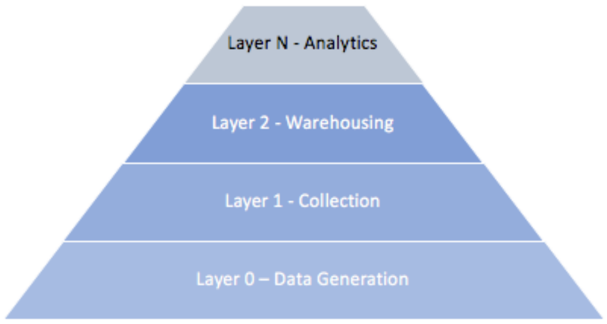

Fig. 2. Wisdom Hierarchy

The goal of the Shasta system is to provide system health capabilities in resiliency and serviceability by using metrics for diagnosability from various system components like:

- Management nodes, containers, services, and orchestration components
- Other infrastructure hardware and software components
- Compute nodes and associated hardware and software components
- Networks and associated hardware and software components
- Storage and associated hardware and software components

However, since Shasta is a finite system, it means that with enough sensor data, logging, event generation and collection, along with tools for running diagnostics and analyzing the data coming back from all these sources, we can provide new capabilities within Shasta. E.g.:

- Comprehensive system health readout APIs
- List of all known bad hardware components
- List of all known bad or misbehaving software components
- Areas where existing problems are indicated based on health checks and sensor data
- Diagnosability enabled infrastructure
- Local system health monitoring control and enablement (each component is instrumented, and data access provided)
- Common system monitoring infrastructure
- System configuration change tracking and history
- Continuous system health monitoring and reporting
- Services reporting state asynchronously to common location via common mechanisms where feasible
- Hardware being monitored asynchronously and reporting state to common locations via common mechanisms where feasible
- Events reported to a common location via common mechanisms where feasible
- Logs collected into a common location via common mechanisms where feasible
- Redundancy for resiliency
- Retrying transient failures. This can be a momentary loss of network connectivity, a timeout because a dependency (such as a database or filesystem) is too busy, or a dropped connection.
- Load balancing to improve resiliency because it allows a component that is in a bad state to be taken out of rotation.
- Data replication to provide fallback option to be resilient from data loss and/or can be recovered
- Throttling to protect the overall health of the system by controlling the number of requests. Applications and/or services may intentionally flood a system with huge number of requests that can reduce the overall availability of those applications or services (from a DoS attack, for example). When a single client makes an excessive number of requests, the application or service might throttle the client for a certain period

VI. Shasta Architectural Features and Capabilities to Support System Health

Resiliency failure handling in Shasta is designed to prevent component failures from causing more widespread failures, both to the system and to applications where possible.

The following capabilities are provided in Shasta in order to support system health:

A. Shasta Diagnosability Components

There are many data sources where events, sensor data, and logs are generated. These are all configurable via REST APIs all the way down to the point of generation to allow for control of what is being generated and how often it is generated.

There is also a common Shasta Monitoring Framework (SMF) for all these data sources that provides the collection points into common data stores for the entire Shasta system. This framework allows for visibility into the collection across all of Shasta and is configurable and accessible via REST APIs as well.

1) *System Health Monitoring Infrastructure*

As mentioned before, monitoring and telemetry are instrumental to provide system health monitoring and diagnosability by using metrics to identify, repair, and recover from failures. The Shasta Monitoring Framework (SMF) provides a common monitoring facility to collect information. The goal is to detect failures as early as possible, preferably before users even know there is a problem. The health model and health services provide the ability for each service to report its health through standard interfaces and a health rollup service that can collect and even diagnose health issues from the services.

The following diagram shows a high-level overview of most components of the general infrastructure architecture as it relates to Shasta system health monitoring. Each channel described in the previous diagrams provides information for continuous monitoring, continuous testing, and on-demand troubleshooting, along with REST API access at every level.

The framework for the data collection and first level warehousing is provided by a Shasta facility called the Shasta Monitoring Framework, or SMF. The following diagram shows a high-level general overview of the SMF.

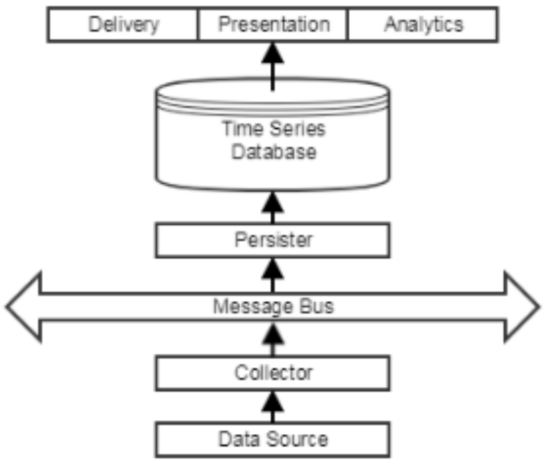

Fig. 3. System Health Monitoring Overview

2) *Diagnosability Components*

The following list of capabilities shown, along with the diagram below that shows functionality that is existing or in-progress in Shasta as a first pass at system health and diagnosability features. Note that on-demand is basically the ability to take a snapshot of system health data via a GET command. Periodic polling of this snapshot (for instance calling the /health GET API in a script) can yield a basic health history over time. Streaming indicates that the monitoring stream is coming into the service asynchronously via a pub/sub mechanism and being persisted into a longer-term system health history data store.

a) *System Health Service (SHS) and REST interface*

The SHS is a standard SMS-node-based system management stateless microservice, with a northbound API that is accessed through the API gateway using the standard AuthN/AuthZ access controls to Cray Shasta APIs, and multiple instances running for services scaling and availability.

b) *Hardware State Manager (HSM) hardware data (via on-demand GET to HSM REST endpoints)*

The HSM provides some basic hardware state relative to various system components, primarily provided by the Redfish BMC interfaces on River and Mountain systems. This includes things like whether a nodes power state is on, off, or unknown (i.e. Redfish not responding), and whether there are any hardware errors being reported by Redfish when HSM does the hardware inventory. It will also aggregate the logical state (such as Admin Down), along with other node reported health data from the HMI service.

c) *Orchestration and Services Metrics Collection (via on-demand GET to Kubernetes and services /health REST endpoints)*

Since Kubernetes was designed from the start to support detailed metrics, it has an integrated API, called the Metrics API (available at: /apis/metrics.k8s.io/ as part of the k8s.io/api/core/v1 package) that allows the user or admin to access a wealth of data as a snapshot of Kubernetes information. These are the same APIs that are used by the various "kubectl " commands, such as "top" or "get component status" that are accessible from the command line in Kubernetes.

The Resource Metrics Summary API is an effort to provide a first-class Kubernetes API (stable, versioned, discoverable, available through the apiserver, and with client support) that serves resource usage metrics for pods and nodes within Kubernetes. It provides metrics about the following Kubernetes objects:

- Node metrics
- Pod metrics
- Container metrics

d) *System Health Service APIs*

For all hardware and software components in the system, the goal is to initially provide a list of:

- Overall health status (good, degraded, error, unknown) for each area. Areas are initially hardware objects, orchestration objects, and services objects.
- A list of objects that are in degraded, error, or unknown state, along with an indicator of what the degraded or error condition is for each object if the system is not healthy (i.e. status is not "good")
- Filter and sort arguments can be applied to the GET calls that allow for the JSON data returned to be filtered and/or sorted.

The following APIs are examples of those targeted to be provided by the initial system health service APIs for diagnosability.

- GET /health - returns all health data known for this Shasta system
- GET /health/hardware - returns all hardware health data known for this Shasta system
- GET /health/services - returns all services health data known for this Shasta system

Some examples of how these APIs are initially planned to work are shown below. Although the actual API calls may change, the same basic functionality of being able to get the overall and detailed status from these system components will be provided.

Ex. a GET to /health might return the following if there are no errors in the system:

```
{
  "SystemStatus":"Good",
  "HardwareStatus":"Good",
  "ServicesStatus":"Good"
}
```

Ex. Whereas a GET to /health might return the following if there are known pieces of failed hardware in the system:

```
{
  "SystemStatus":"Degraded",
  "HardwareStatus":"Error",
  "ServicesStatus":"Good",
  "DegradedHardware"
  {
    <list of all failed hardware in the system, with details on each one showing what is known to be in error>
  }
```

3) Diagnosability Attributes

The Shasta system specifically provides the following attributes to allow for basic diagnosability:

- Enhanced deterministic hardware failure detection and isolation
    - Modular HW architecture provides a sufficient number of sensors in each module to detect any hardware failure down to the Field Replaceable Unit/Customer Replaceable Unit (FRU/CRU) level. FRUs are typically something that is replaced in the field via Cray service personnel, while CRUs are something that can be replaced in the field by a customer. For the purposes of diagnosability, however, these are treated the same, as it represents an atomic level of replaceability, no matter who does the actual replacement. As mentioned earlier, the goal for hardware failure detection and isolation in Shasta is 100%, reaching as high as possible with HW.
    - Data from the sensors in all system components is collected in a granular enough manner to allow for FRU/CRU level failure isolation.
    - Common component naming approach (cname) from the XC platform has been modified and expanded to include all of the Shasta components and associated geolocation data. This new attribute in Shasta is called the xname and allows for isolation down to the exact failed components.
    - Hardware failure states are deterministic, meaning they are latched as failed until the failure clears, either by itself, via a repair action, or via explicit command. Critical to this from a service standpoint is the ability for the complete failure lifecycle to be captured, so that failure and recovery of a component is logged.
- Common log collection and formatting
    - Uses common telemetry/monitoring bus and common logging database
    - The SMF system described above provides a common logging collector. All logs generated by any firmware or software in the system end up in this common system log. This allows for the use of industry standard tools like LogStash and ElasticSearch to be used for working with system logs across all components.
    - All log messages use a human readable, but easily machine parseable JSON format.
    - All these log messages are also formatted into a common user readable format, which makes correlation and analysis of the log data easier.
- Common event collection
    - Uses common telemetry/monitoring bus and event database
    - Event data from every component in the system is gathered into the event database via the same common telemetry bus that the rest of system health monitoring uses.
    - Audit log information such as configuration changes made to the system also logged to the event stream, allowing for correlation of configuration changes with system health monitoring.
- Continuous system monitoring
    - Uses common telemetry/monitoring bus and common telemetry database
    - The firmware in all Shasta Mountain embedded controllers provides an asynchronous push telemetry stream for the dense compute platforms. This is provided using the Redfish EventSubscription mechanism and REST APIs to POST the telemetry data. This same thing is true of the Shasta River Slingshot network TOR switches.
    - The Shasta River COTS servers, because they are provided by third party vendors, do

not typically provide this same asynchronous push telemetry stream, so a synchronous polled mechanism is used to provide the telemetry monitoring collection.

- All Shasta hardware telemetry, whether push or polled, is collected by the common Shasta telemetry service in the management plane, published to the common telemetry and monitoring bus, and aggregated into a common telemetry database.

B. *System Management Services*

Shasta system management services consists of system (or infrastructure) management, part of the management system that is responsible for the configuration, operation, and monitoring of the entire system, component management, also part of the management system in Shasta, but responsible for the configuration, operation, and monitoring of the individual components and embedded controllers, and the managed system, that is the user facing portion of Shasta like the Cray Managed Ecosystem (CME).

From a resiliency perspective one of the main benefits with separate management system and managed system is that they can be upgraded or updated independently.

Several resiliency features are provided by the Shasta System Management Services (SMS) architecture, as can be seen in this diagram. These features, and how they enhance the function of the management system, are as follows:

- *API load balancing* - Load balancing capability is built into the API gateway and container orchestration. Higher-level load-balancer microservices, or physical load balancers sitting above the gateways, can also be added.
- *Stateless services* - Service configuration and state are stored and retrieved by the micro-service instances in an internal data store that is only directly accessible to that service; the only external accessibility is provided by REST APIs to that micro-service. By design all the services are stateless, allowing them to be replicated so they can be scaled up or down by creating more or fewer instances. Persistent data is stored in a shared data store.
- *Desired state reconciliation and service replication provided by container orchestration* - Service high-availability and resiliency are provided by the container orchestration service. If a service container becomes unavailable (e.g., becomes unresponsive), it is detected and recovered via the container orchestration service. Service replication, managed across the nodes by the orchestration tool, also gives the system resiliency in availability, as well as reliability. If one of these replicated containers goes down, the service remains available to service APIs and perform its primary functions. Container orchestration services will also reschedule the stopped container on the same or a different host, allowing the services to be migrated in the event of host node failure or maintenance actions.
- *Services instances spread across all SMS nodes* - The containerized micro-service instances (3-n) can run on any SMS node. This allows the service to continue to function if an SMS node is lost. This multi-instance, micro-service architecture enables expanding service instances out across as many physical machines as needed for redundancy, availability and scalability.

C. *Application Resiliency*

Applications based on Message Passing Interface (MPI) have some resilience to transient network failures provided by a resend capability for short MPI control messages. Link failures are also automatically routed using adaptive routing in many cases.

D. *Key Infrastructure Components - Compute, Storage and Networking*

Shasta systems are designed with far more flexibility in mind than past Cray high-end systems, both in hardware and software. Key infrastructure such as compute, network, storage is outlined in the section below.

1) Compute

When a compute node goes down, any application running on the failed node is usually also lost since the application's state cannot be guaranteed. The part of the application that was running on the failed node is completely lost, leaving the other parts of the application that were running on other nodes to be cleaned up. The Shasta job launch architecture will detect node failures and report these back to the workload manager (WLM), so that the scheduler can take appropriate action to either recover them or remove them from the list of available nodes to run a workload on.

Software errors from user code should never result in a compute node failure, although they could cause an application failure. The User Access Services (UAS) allow the user to create a User Access Instance (UAI) that allows the user to use the WLM and job launch service to run jobs on compute nodes. These various services provide both command line utilities and REST APIs to allow the user to launch applications onto a set of compute nodes. They also provide supporting functionality such as helper tools, binary transfer mechanisms, job state querying, and job signaling, among others.

Since the UAS instance-creation functions, and the WLM service and scheduler functions are all containerized, running within the System Management Services (SMS) Kubernetes cluster, and since they use that infrastructure to store their operational data in a clustered decentralized database that is accessible across nodes, it ensures that these services can move to different nodes for high availability. These services can also be scaled up across nodes to handle additional load by starting more instances within the Kubernetes orchestration system.

The primary objective of compute node-failure handling is one of the recovering resources and making them available to run jobs as quickly as possible. The system monitoring framework provides a declarative state of compute nodes that is the first part that will manage this failure, by providing the diagnosability. Once a node is known to be bad, part of resiliency around automatic recovery is to remove that node from the WLM and job launcher, so it no longer causes any application failure. Another part is to signal to the system administrator that the node is bad, along with enough information to diagnose exactly why. The last part is to affect a repair action on the node, part of which may be done automatically (like ordering new memory), and part of which may be done manually (such as actually replacing the memory).

2) *Storage and I/O Failure Handling*

The Lustre parallel filesystem provides high-speed, large-volume storage for applications. Lustre is designed to enable high availability of data by using no-SPOF (single point of failure) hardware designs. Lustre can utilize multiple paths to access data in the event a that any single component in the data path fails. Lustre assumes servers are grouped and connected as "failover pairs": two servers, each normally serving separate RAID arrays, can take over for each other in the event that one of the servers fails. This model requires that the endpoint of the data path, the underlying backend storage array, is durable, with drives that will always be available. This in turn requires that the backend storage devices must be dual-ported, offer RAID protected-data, and use redundant power and fan modules.

The ClusterStor hardware platform meets Lustre's availability design requirements by providing fully redundant paths to the stored data. ClusterStor's Scalable Storage Units (SSU), SSU-based model uses dual controllers (servers, a/k/a OSSs) with two sets of RAID arrays (a/k/a OSTs) in each physical enclosure. The enclosure and rack hardware are specially designed to provide dual-path access to data via these dual controllers, dual side cards, dual NICs and cables, dual top-of-rack switches, dual-ported disks (SAS drives), as well as multiply-redundant fans and power supplies to ensure system resiliency when a failure occurs. However, this guarantee of continuous data access comes at a significant cost, as more complicated hardware designs require expensive dual paths and the manufacturing volumes of such systems are low, compared to commodity hardware used in some vendors' products.

Two types of service nodes support the Lustre filesystem, the Object Storage Server (OSS) nodes and the Meta-Data Server (MDS) nodes. The OSS nodes host the Object Storage Targets (OSTs), while the Lustre MDS nodes are responsible for handling and storing the filesystem metadata. At a basic high level, the Lustre software consists of the OSS nodes, the MDS nodes, and a Lustre client, that runs on each of the compute nodes. There may also be a Lustre network (LNET) router that runs to provide connectivity between different areas of the filesystem and the compute nodes.

If a compute node client fails to make a timely response, the Lustre system evicts the failed client from the system. This action frees resources (such as file handles and export data) associated with the client and allows other clients to acquire them. Cray Management Services (CMS) logs the problem but no special action is requested of the system administrator in this case.

Failures are possible at several places along the Lustre I/O data travels:

- The compute node Lustre components
- The compute node itself
- The Lustre service nodes
- The network components connecting the service nodes to the disk consolers
- The paths from the disk consolers to the disks
- The disks themselves

3) *Networking*

The network and fabric management services architecture are composed of:

Managers that:

- act as the customer-facing front end for network management
- provide orchestration for customer-provided policies
- manage network and fabric telemetry gathering

Services that:

- provide fabric-level agencies, such as MAC-to-port learning
- support basic network protocols, such as IGMP
- offer scalable solutions for network protocols, such as ARP and DHCP

Controllers that:

- provide an abstracted interface for fabric management
- persist fabric configuration data
- Agents that:
- offer Controllers remote access to the switch device driver

Failures can happen at any of these levels, with varying degrees of impact. E.g.,

- Manager failures should allow the system to continue functioning with only the loss of active customer control, i.e., the existing network and fabric configuration should continue to run, only further configuration changes and telemetry gathering should be affected. Some examples:
- The failure of the etcd Key/Value store, used to store configuration data, would result in the Network Management Services (NMS) managers being unable to save or restore configurations
- The failure of the fabric manager would result in the customer being unable to monitor or change the existing fabric configuration

- The failure of the monitoring manager would result in the loss of scheduled data collections
- Service failures could result in basic network protocol failures, such as a node being unable to get a DHCP address, or an ARP resolution
- Controller failures would result in loss of active customer control, and an inability to respond to error conditions
- Agent failures could result in a switch, and its attached nodes becoming lost to the rest of the fabric

a) Management Networks

A Shasta system includes a robust management fabric, composed of commercial off-the-shelf switches. The network architecture is a leaf-and-spine design, with redundant high-bandwidth spine switches, and optional redundant leaf switches at the cabinet level. Switches below the cabinet level, at the chassis and blade levels, are not redundant.

Failures within the management fabric hardware could impact accessibility and functionality depending on the location of the failure within the switch hierarchy.

The failure of a management fabric NIC would result in management traffic, such as LDMS event gathering or service traffic, becoming lost. The following might result:

- If the NIC is connected to a service node (SSN), then the loss could mean an interruption of services based on that SSN, depending upon whether a given service has its data cached locally, or is dependent on a connection to remote storage.
- If the NIC is connected to a compute node, then the loss would result in the loss of event data and customer management access to the node.

The failure of a management fabric switch could result, depending upon where it is in the management fabric, in widespread loss of connectivity on the management networks.

While the spine and cabinet switches are redundant, the loss of a chassis or blade switch would result in all management NICs below that point in the hierarchy becoming unreachable.

b) Connections to the Data Center

Shasta systems can be connected to customer data center networks through redundant connections.

c) Management Fabric

The customer's Data Center Admin Network can be connected to multiple Management leaf switches in River racks, providing redundant paths to the management elements in the system, like in Figure 4.

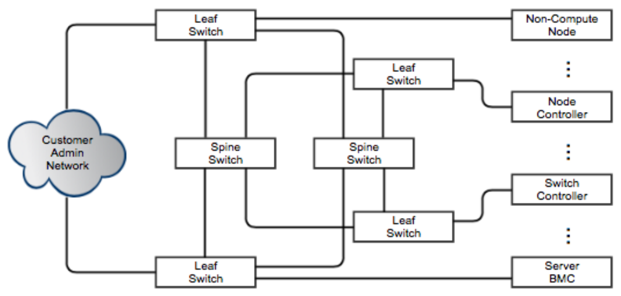

Fig. 4. Redundant Management Network

d) High Speed Fabric

The Slingshot High Speed Fabric can be connected to the customer's data center through either a switch or a router. Not only can multiple paths to the fabric be connected, but these connections could be composed into a link aggregation group (LAG) that provides automatic load balancing and failover, like in Figure 5.

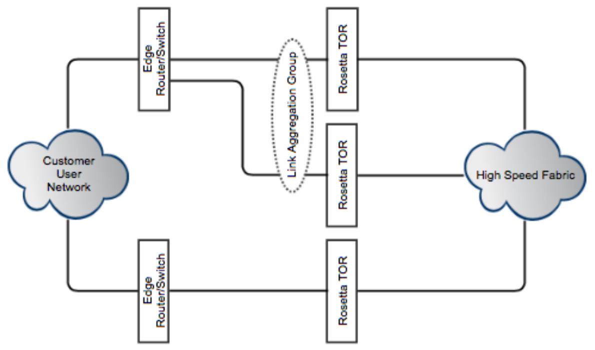

Fig. 5. Slingshot Highspeed Fabric

VII. Conclusion

The architectural and design goals within Shasta are to allow for complete knowledge of the system at every available layer, by providing all the attributes, such as monitoring, measurability, and diagnostic tools, all in a common format and location, to allow for visibility into the health of the system. The new approach in Shasta using modular architecture provides multi-instance orchestration resiliency. With its redundancy of major components, the Cray Shasta system is designed to be highly available and highly reliable. Compute nodes can be taken out of service and returned independently of one another. Should any component of a compute node fail, only the application running on that node is affected. Both software and "soft" hardware failures, such as many uncorrectable memory errors, allow the node to be rebooted. Only "hard" hardware failures require service.

Because the Rosetta router switches are located on a separate switch module, compute blades can simply be removed once all applications using the healthy nodes on the blade are terminated and the nodes have been configured as unavailable. This blade can then be swapped with a new one and brought back into operations.

The ability to monitor and diagnose the system is a critical piece that informs the ability to provide recovery actions (whether manual or automatic), as well as to be able to take service actions like perform live or rolling updates to system components or replace hardware. These facilities working together provide an end to end system health capability within the Shasta system which has been engineered to be a significant improvement over the system health maintenance of previous Cray products.

Key System Health features of the Cray Shasta products include the following:

- Container orchestration by Kubernetes for both management and user applications
- A consistent and standard platform for service node layers that allows services to grow or expand dynamically based on need
- Redundant power supplies, voltage regulator modules and converters
- Redundant cabinet cooling infrastructure
- Compute nodes with no local spinning disks
- System software that automatically marks non-responsive nodes as unavailable in the resource manager
- Automatic detection and rerouting of defective links in the Cray Slingshot interconnect without affecting operation or requiring a reboot
- Adaptive routing that provides alternate routes to the target node
- A Cyclic Redundancy Check (CRC) at the packet level between source and destination
- In addition to the CRC, there is forward error correction and link-level retry
- Packets protected by Error Correction Code
    - The Cray NodeKARE™ software tool detects certain correctable errors and marks nodes down when they have high error rates
- Resiliency communication agents that monitor all nodes and initiate failover of services, where applicable
- A high level of system failure detection and isolation.
- Comprehensive system health readout APIs
- Diagnosability enabled infrastructure
- Continues system health monitoring and reporting

VIII. Refences

[1] Jeffrey J. Schutkoske (2017). “Cray® XC40TM System Diagnosability”, https://cug.org/proceedings/cug2017_proceedings/includes/files/pap140s2-file2.pdf

[2] Jeffrey J. Schutkoske (2014). “Cray XC System Level Diagnosability: Commands, Utilities and Diagnostic Tools for the Next Generation of HPC Systems”, https://cug.org/proceedings/cug2014_proceedings/includes/files/pap120.pdf

[3] Cray System Snapshot Analyzer (SSA). https://www.cray.com/support/cray-system-snapshot-analyzer

[4] Cray SSA White Paper. https://www.cray.com/sites/default/files/resources/Cray-SSA-White-Paper.pdf

[5] Numerous workshop authors, "Large-Scale System Monitoring Experiences and Recommendations", Workshop paper: HPCMASPA 2018, https://ovis.ca.sandia.gov/images/7/7d/HPCMASPA_Large-Scale_Monitoring_Experiences_Recommendations.pdf

[6] Daniel Dauwe ; Sudeep Pasricha ; Anthony A. Maciejewski ; Howard Jay Siegel, "An Analysis of Resilience Techniques for Exascale Computing Platforms", https://ieeexplore.ieee.org/document/7965137