ibm-crassd Documentation

Release 1.1-5

Justin Thaler

Feb 24, 2020

Contents:

CHAPTER 1

Introduction

1.1 Executive Summary

Cluster monitoring has different requirements based on data center size, purpose, the amount of available bandwidth for monitoring functions and the need for 3rd party support. We show a RAS aggregation approach which leverages open source tools to aggregate sensor and alert data from OpenBMC's management interface. Through a highly configurable broadcast subscription service and an extensible plug-in framework, telemetry data can be collected, associated with cluster jobs, and nodes actively monitored without impacting job performance. We demonstrate how AC922 management is possible with Netdata, Kibana, and Grafana.

1.2 Introduction

IBM's AC922 system design motivated by the SUMMIT and SIERRA supercomputer requirements. As a large scale-out system, each contribution to management network load deserves careful consideration to ensure quick communication of reliability or failure events and the flexibility of a prescriptive configuration a requirement.

The AC922 was IBM's first system to use OpenBMC for its onboard system control. OpenBMC implements a REST API with features motivated by the Redfish API. These interfaces allow for the configuration of event and monitoring along with enabling tuning per the desired load on the cluster management network.

There are several options for cluster management and many administrator teams have written extensible solutions tailored to their sites. In supporting a test and development cluster, our goal was to use up to 50% of the network for system sensor monitoring without adding jitter to workloads. We explored NetData because of its active community, the availability of plugins for major cluster functions, and because it was running for us within a half day after the initial download.

In as much as real time sensor data was important, this is eclipsed by the importance of the system identifying a functional fault or has otherwise detected a non-optimal condition. Depending on the cluster policy, these events may indicate a need for service due to loss of redundancy or even immediate action. These events are used by cluster managers, like IBM's Cluster Service Manager (CSM), to take nodes out of service for maintenance. We show how these events can be logged to Logstash and displayed by Kibana or by Grafana.

This white paper documents the architecture of our cluster monitoring system with the intended audience being administrators who are looking for an Open Source solution for monitoring an AC922 cluster or administrators integrating AC922s into existing solutions. In either situation, the key components are defined in the Key Software Components table and of those, the IBM Cluster RAS Service Daemon (ibmcrassd), is the highly configurable cluster aggregator that allows for scalable processing of node telemetry and system alert propagation.

1.3 Key Software Components

Open Source Software	Function	Host
OpenBMC	Monitors system check-stop, fan failure, vital component deconfiguration, power supply failure, DIMM failure, CPU failure, GPU failure, voltage regulator failure, thermal shutdown, unable to communicate with OCC, hostboot procedure callouts	Compute Node BMC
xCAT (optional)	Enables autoconfiguration for ibmcrassd by providing the association BMC IPs and service group in clusters with tiered service node architectures	Service Node
Ibmcrassd	aggregation, event creation, pairing RAS policy to events, plug-in support for downstream processing	Service Node
Netdata plugin for ibmcrassd	This plugin for NetData allows it to communicate with ibmcrassd service for sensor monitoring	Monitoring node

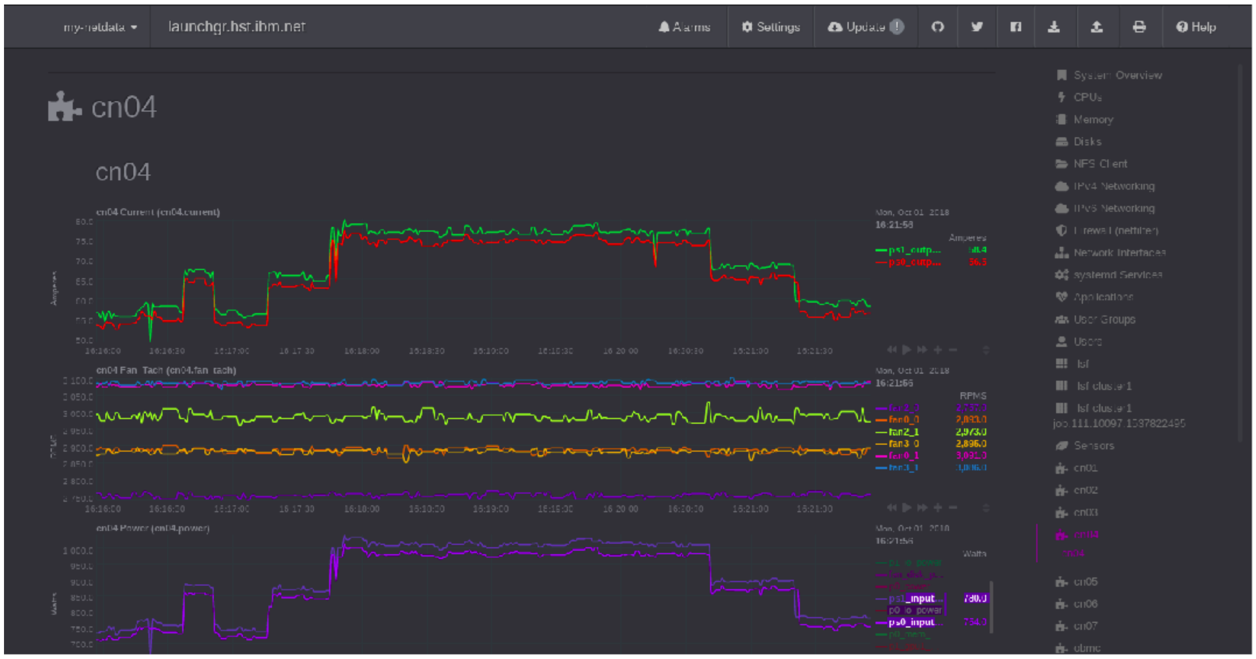
1.4 Sensor Monitoring

OpenBMC presents sensor data to the management BMC interface and receives some from the Host OS. There are 113 sensors that are available from the BMC's management interface.

The default for Netdata was to poll data every 1 second. This level of monitoring requires at worst 21 Kb/s per compute node from the BMC to the Service node running ibmcrassd and 6.8 Kb/s from the Service node to the Netdata server per compute node. The connection to the OpenBMC is more efficient than outlined above, since the OpenBMC only sends a push notification when the sensor value changes. The 21 Kb/s figure represents all sensors being updated every second.

In addition to the network load outlined above, the sensor data is cached by ibmcrassd, using ~21 Kb of memory per monitored node. Furthermore, the processing of push web-socket notifications from the BMCs uses 0.3% to 1% of a CPU core per monitored node. Scaled up to 300 monitored nodes, ibmcrassd scales the service with 50 monitored nodes per CPU core. For example 6 monitoring cores are used to monitor 300 nodes, with each core utilizing 15% to 50% of the core. In addition to this another core is dedicated to sending data to clients, which uses on average 0.3% of the CPU core per subscribed client.

To start monitoring sensor readings with Netdata, we installed the Netdata plugin for ibmcrassd, started ibmcrassd service on the service node with telemetry enabled in the configuration file, then started the Netdata service on the monitoring node. The plugin subscribes to the telemetry stream on the ibmcrassd service, by opening the configured socket defined by a port number and the IP/hostname. Sensor data transmissions then begin immediately with sending everything. The plugin does not take advantage of the filtering and update frequency rates as it gets all sensors at a rate of once per second. It then updates Netdata with the sensor values at a rate of once per second.



The screen shot above was used to review the power supply balance during the execution of a CPU GEMM test.

1.5 Alerts

The alerts provided by the OpenBMC are used to indicate a change in the state of the compute node. While the ibmcrassd service does NOT track the state of each compute node, it aggregates these and enables other programs to subscribe and use the information for active cluster management. This is implemented for CSM and designed for easy integration to existing management frameworks.

The ibmcrassd service converts the base alert provided by the BMCs into a common format so that all openbmc based machines have a common look and feel. This is done using a policy table that is specific to a machine type. These policy tables can be converted and tailored by an end-user to add or change alert properties in accordance to the service strategy of the site. In addition to this, the CSM plugin also has its own policy table, in addition to the base one used for translating the BMC alerts, so it can be instructed how to react, without needing any base knowledge of the underlying hardware.

Each endpoint for alerts to the ibmcrassd service is treated as a plugin that must be enabled for alerts to be forwarded to it. The service tracks the status and last communication to each endpoint individually, such that, if communication is lost to an endpoint, it will be resumed, once the endpoint is back online. This status is tracked in a file that must be located within persistent storage. To support stateless service nodes, a persistent storage location can be specified in the ibmcrassd configuration file.

The following figure shows an example of sending alerts into multiple systems managers including an IBM proprietary manager for Elastic Storage Server (ESS), CSM, and Logstash as an entry to an ELK stack.

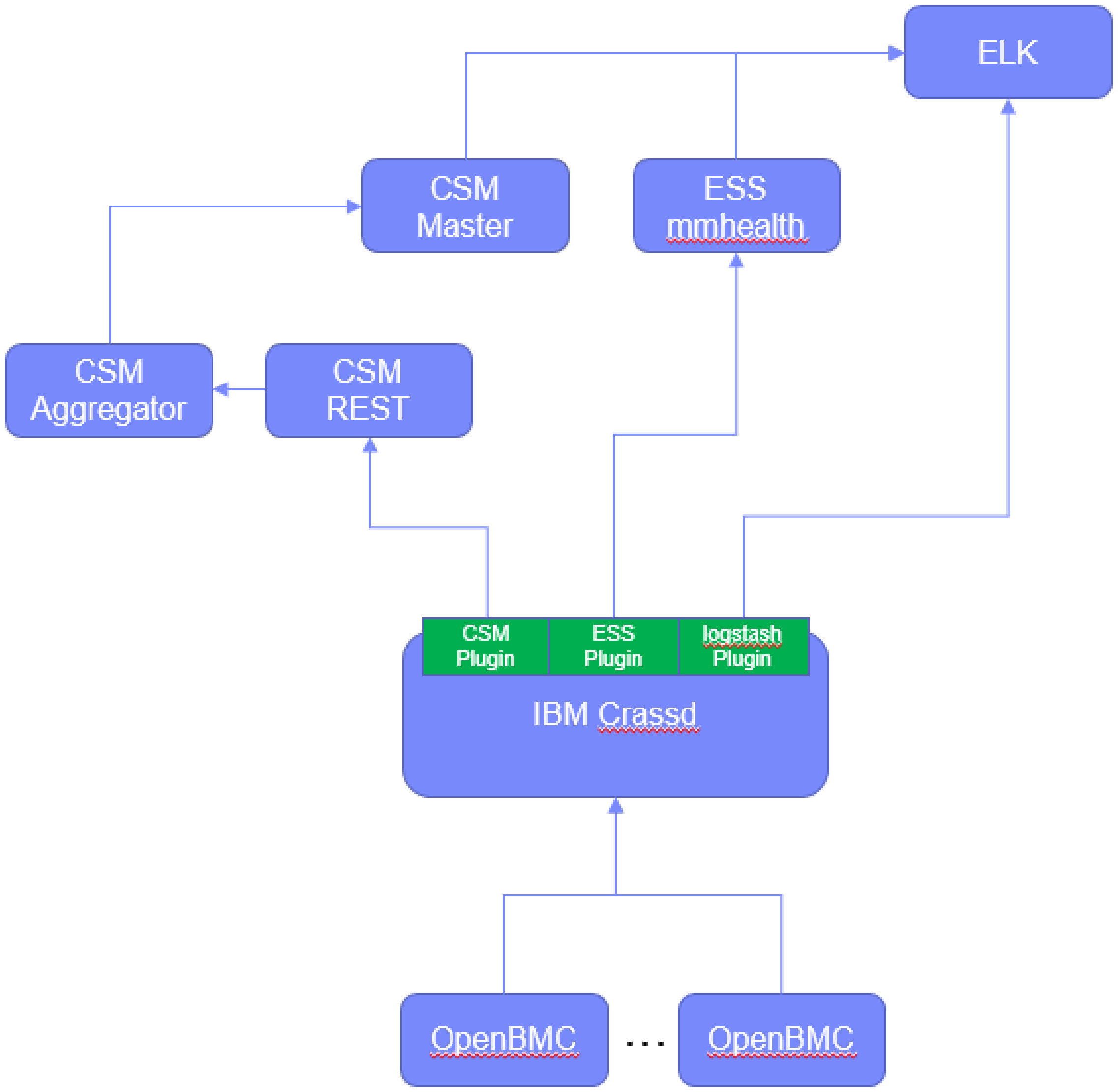


Figure : Example alert flow with ibmcrassd to multiple endpoints

It is important to note that all of the alert management is contained to a single physical CPU core at any given point in time. It leverages multiple I/O based threads to process the information as needed. The service has proven very stable, even in an event storm. When original put on-line in a cluster with 6 months of old alters, it processed about over 35,000 alerts from over 4,000 nodes in less than five minutes.

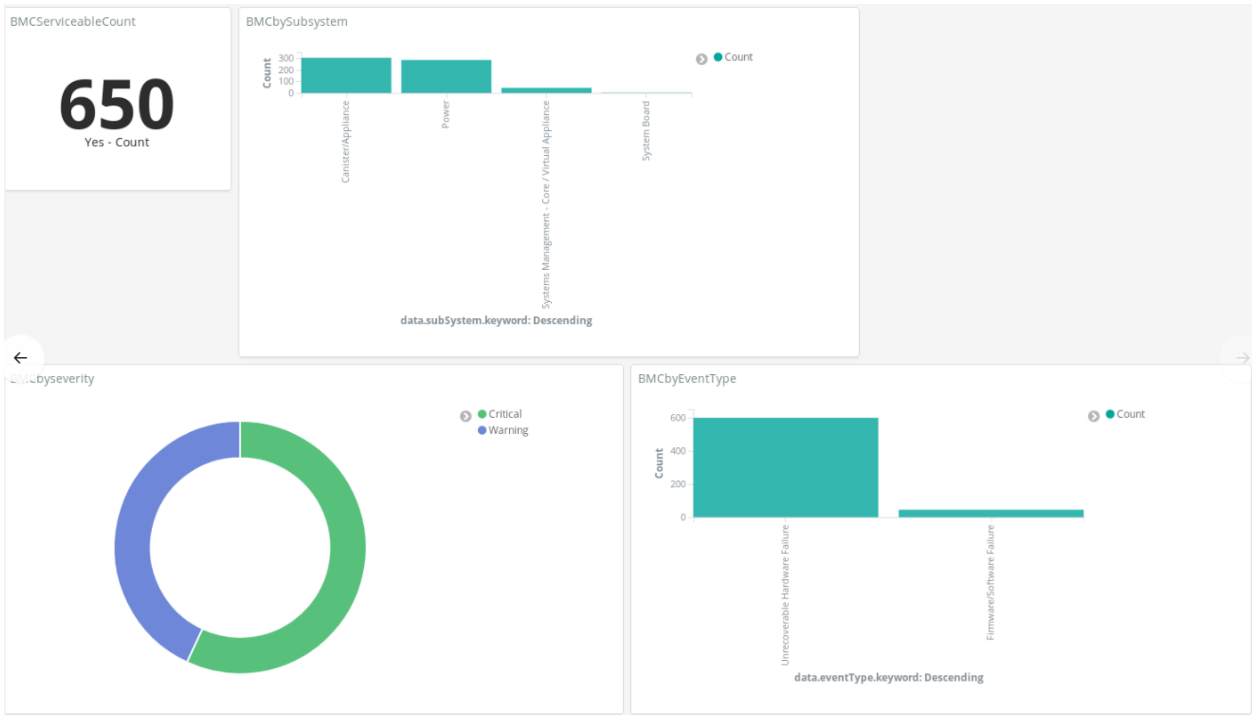

BMCServiceableCount
650
Yes - Count
BMCbySubsystem
Count
Canister/Appliance
Power
Systems Management - Core / Virtual Appliance
System Board
data.subSystem.keyword: Descending
BMCbyseverity
Critical
Warning
BMCbyEventType
Unrecoverable Hardware Failure
Firmware/Software Failure
data.eventType.keyword: Descending

Figure : An Example Kibana Dashboard

1.6 Scalability

The ibmcrassd can effectively monitor up to 288 compute node BMCs with 8 POWER9 grade CPU cores. Its scalable architecture registers a web socket with each compute node BMC and configures events with OpenBMC.

Using the HTTPS REST API management interface, the ibmcrassd service first logs into the node to establish a session. It then uses the session information to subscribe to BMC alerts. An I/O based thread is used for each monitored node. This will result in a maximum of 288 threads that listen for push notifications.

When an alert notification is received, one of the configured number of worker threads are woken up to process the node alerts. It then ensures no alerts were missed by querying the specified BMC for all alerts. Each event is reviewed against the programmable policy table to extend the event with node attributes. Each new event is then forwarded to the configured plugins. The plugin architecture allows downstream archiving and analysis, active cluster monitoring with event propagation (less than 20 sec), and visualization. Plugins are currently available for IBM's Cluster Service Manager (CSM), Netdata, and Logstash.

For the sensor data stream, more processing power is needed, and the service had to be made scalable to use multiple physical cores of the service node's CPUs. In addition to all of the I/O threads above for alerts, we again have 288 I/O threads for listening to sensor data with a web-socket push mechanism. Each of the I/O threads is paired with another I/O thread that processes the sensor data notification and stores it into a cache. The updates are then forwarded to the client subscription sub-process.

A separate CPU core is dedicated to handling client subscriptions to ensure that the clients receive a responsive interface, regardless of what the remainder of the ibmcrassd service is handling. Given the dedication of multiple physical cores, these will appear as multiple instances of the ibmcrassd service, if for instance, the top command is run on the service node.

1.7 Conclusion

With providing both a live feed of sensor data and alerts to varying systems managers, hardware centric monitoring no longer requires a separate management system. Folding nicely into existing open-source management tools, with great flexibility, a single pane of glass for systems management becomes possible. This holds true through the two types of systems management models of distributed vs centralized.

The ibmcrassd service is light enough on systems resources it can be run alongside systems managers that are distributed, or it can be run on a centralized service node, to allow for maximum performance of managed nodes. By fitting both models of management, it will drive consistency for IBM hardware, regardless of what manager is used, and ensure a quality production environment is delivered.

1.8 Key References

Netdata: https://github.com/netdata/netdata/wiki

Grafana: https://grafana.com/

Logstash (ELK): https://www.elastic.co/elk-stack

OpenBMC: https://www.openbmc.org/

IBM Power System AC922: https://www.ibm.com/us-en/marketplace/power-systems-ac922

CHAPTER 2

Installing IBM CRASSD

2.1 Prerequisites version < 0.8-10

- Python 2.7 or greater
- Java 1.7.0 or greater
- python-configparser (found in EPEL)
- python-websocket-client
- python-requests if monitoring 8335-GTG, 8335-GTC, 8335-GTW
- openbmctool if monitoring 8335-GTG, 8335-GTC, 8335-GTW
- ipmitool

2.2 Prerequisites for ibm-crassd 0.8-10 and above

This version switches to python 3 compatibility

- Python 3.4 or greater
- Java 1.7.0 or greater
- python-configparser (found in EPEL)
- python34-websocket-client
- python34-requests if monitoring 8335-GTG, 8335-GTC, 8335-GTW
- openbmctool if monitoring 8335-GTG, 8335-GTC, 8335-GTW
- ipmitool
- pexpect

2.3 Installation

2.3.1 ESS

1. From the ESS management node type `yum install /install/gss/otherpkgs/rhels7/ppc64le/gss/ibm-crassd-0.8-1.ppc64le.rpm`

2.3.2 RHEL Based Systems

1. Transfer the rpm to the management system or service node you wish to install this service on.
2. Type `yum install /path/to/rpm`, where /path/to/rpm is the full path to the rpm you just copied to the system.

CHAPTER 3

Upgrading

3.1 Before you begin

If you are updating from 0.8-9 to version 0.8-10 or newer, the package requirements changed. Specifically python 3 based packages are used. During this upgrade, the ibm-crassd service will stop temporarily and monitoring will cease during this time.

3.2 Upgrading ibm-crassd

1. Backup the ibm-crassd.config file found at `/opt/ibm/ras/etc/ibm-crassd.config`. This location may vary and the default is shown.
2. Backup the bmclastreports.ini file found at `/opt/ibm/ras/etc/bmclastreports.ini` This location may vary and the default is shown.
3. Stop the service using `systemctl stop ibm-crassd`.
4. Install ibm-crassd using the instructions above.
5. Restore the two backed up files to their original locations.
6. Start the service using `systemctl start ibm-crassd`.

CHAPTER 4

Configuring ibm-crassd

4.1 Manually Configuring Nodes

1. Open the configuration file located at `/opt/ibm/ras/etc/ibm-crassd.config`. This location may change.
2. The `[nodes]` section is located near the top.
3. Nodes must be added or removed sequentially. For example, node1, node2, node3. a. See the section on nodes to get an explanation of each of the fields comprising nodes.
4. Save and close the file.
5. Node changes will be picked up when the service is restarted.

4.2 Node Entry information from the ibm-crassd configuration file (/opt/ibm/ras/etc/ibm-crassd.config)

Node entries in the ibm-crassd configuration file contain several elements. The entire property set must also be encompassed with {}. Below is an example entry. `node1 = {"bmcHostname": "my-9006-bmc.aus.stglabs.ibm.com", "xcatNodeName": "xcat2", "accessType":"ipmi"}`

Each property comes as a pair on the right hand side of the equals sign. The properties don't have to be in a specific order. All property IDs and values must be encased in quotes. • bmcHostname: This is the string that is either the BMC hostname or the BMC IP address.• xcatNodeName: This is the hostname of the Host OS. This may also be the IP of the Host OS. • accessType: This tells what the connection method is to the BMC. Currently accepted values are ipmi (for 9006, 5104 Machine Types) and openbmcRest for 8335-GTC and 8335-GTW systems.

4.3 Setting up the base configuration section

This section allows the user to specify some basic controls for the ibm-crassd service.

- The maxThreads variable is used to define the number of processing threads that are used to collect, parse and forward BMC alerts to the various plugins, based on what is enabled. The current recommended setting for this variable is 40.
- The enableTelemetry option can be set to **True** to turn on telemetry streaming, or **False** to disable telemetry streaming.
- The nodesPerGathererProcess option is used to set the number of BMCs assigned to a sub-process. This can allow performance to be finely tuned for the telemetry streaming service. The default setting is 10.
- The enableDebugMsgs option can be set to True when trying to debug a difficult problem or to help find problems with initial setup. The default setting is False.

4.4 Configuration of the Tracker File Storage Location

Configuring where to put tracker files can be found under the section of [lastReports]. This directory is also used to specify where the configuration file for ibm-crassd is located. The default location is **/opt/ibm/ras/etc** To support multiple tracker and configuration files in the same storage location (directory), it is imperative to name the tracker and configuration files with the format **<nodewithcrassd_hostname>.ibm-crassd.config**. The ibm-crassd service places highest priority when the hostname.ibm-crassd.config file name format is used.

4.5 Configuration of Analysis Scripts for ibm-crassd

Deep analysis of alerts is sometimes wanted or desired. The ibm-crassd service supports this behavior by allowing the creation of analysis files, and then adding configuration options for them. To create and run analysis scripts, a few things must be done.

1. Create a python module, with the following name format analyze**alertID**.py.
2. Add a configuration option to ibm-crassd.config file under the analysis section using the following format **alertID=option**.
3. Configure the setting to one of the following three options.
   - **clear** - This option, after a positive analysis return, will have ibm-crassd delete the alert from the BMC which surfaced it.
   - **filter** - This option, will only cause ibm-crassd to filter the alert so it's not sent to the varying reporting plugins. It will leave the alert on the BMC.
   - **disable** - This option will prevent the analysis script from being run, and the alert will be forwarded as normal.

CHAPTER 5

Plugin Configuration

5.1 Configuration for integrating into ESS

1. Open the configuration file in `/var/mmfs/mmsysmon/mmsysmonitor.conf`.
2. Locate the `[general]` section of the file.
3. Add entry: `powerhw_enabled = true`
4. Save and close the file.
5. Open the configuration file located at `/opt/ibm/ras/etc/ibm-crassd.config`
6. Locate the `[notify]` section
7. Set `ESS=True`
8. Set `CSM=False`
9. Enable any other plugins by setting their name to True. Disable any plugins you do not wish to use by setting their value to False. a. For example `ESS=True`, `CSM=False`, `logstash=False`.
10. Save and close the file.
11. Type `mmsysmoncontrol restart` to reload the mmhealth service and pick up the changes.
12. Start the service with `systemctl start ibm-crassd`. It is also recommended to enable the service so it starts automatically when the Host OS starts. This is done using the command `systemctl enable ibm-crassd`

5.2 Configuration for integrating into CSM

1. Ensure CSM services are running according to CSM documentation • csmrestd must be running on the same node as the ibm-crassd service.
2. Open the configuration file located at `/opt/ibm/ras/etc/ibm-crassd.config`

3. Locate the `[notify]` section
4. Set `ESS=False`
5. Set `CSM=True`
6. Enable any other plugins by setting their name to True. Disable any plugins you do not wish to use by setting their value to False. • For example `ESS=False`, `CSM=True`, `logstash=True`.
7. Locate the `[csm]` section of the configuration file. • Specify the IP address and Port for the csmrestd service. The defaults are specified and will work with most configurations.
8. Save and close the file.
9. Start the service with `systemctl start ibm-crassd`. It is also recommended to enable the service so it starts automatically when the Host OS starts. This is done using the command `systemctl enable ibm-crassd`

5.3 Configuration for integrating with logstash

1. Ensure logstash services are running according to logstash documentation
2. Open the configuration file located at `/opt/ibm/ras/etc/ibm-crassd.config`
3. Locate the `[notify]` section
4. Set `logstash=True`
5. Enable any other plugins by setting their name to True. Disable any plugins you do not wish to use by setting their value to False. • For example `ESS=False`, `CSM=True`, `logstash=True`.
6. Locate the [logstash] section of the configuration file. • Specify the IP address and Port for the logstash service. The defaults are specified.
7. Save and close the file.
8. Start the service with `systemctl start ibm-crassd`. It is also recommended to enable the service so it starts automatically when the Host OS starts. This is done using the command `systemctl enable ibm-crassd`

CHAPTER 6

Checking health of the ibm-crassd service

6.1 Controlling the ibm-crassd service with system controls

The following commands can all be run from the command line with the system that has the IBM crassd service installed.

6.1.1 Verify the service is running

```
systemctl status ibm-crassd
```

6.1.2 Starting the service

```
systemctl start ibm-crassd
```

6.1.3 Stopping the service

```
systemctl stop ibm-crassd
```

6.1.4 Enabling the service so it starts automatically on subsequent boots

```
systemctl enable ibm-crassd
```

6.1.5 Checking the system logs for problems

The journalctl command is the best way to check for alerts within the Linux Journal. The ibm-crassd service will create entries for problems it encounters and can be useful for debugging setup problems. The following command can be used to find all journal entries. `journalctl -u ibm-crassd`

6.1.6 Checking for forwarded BMC alerts in CSM

To check the power status of the nodes in csm the following command can be used: `/opt/ibm/csm/bin/csm_ras_event_query -m "bmc.%" -b "2018-07-25 14:00:00.000000" -l "sn01"` The above command specifies several things, the first option being the message. Here, we use "bmc.%" to get all alerts sourced from the bmc. These are the only types of alerts forwarded by ibm-crassd. The next option -b is used to specify all alerts from a time forward and can be omitted if needed. The example provided displays all alerts since 2:00 PM on July 25, 2018. The last option is used to specify a location. This is optional as well and can be used to specify a specific node. This is listed as xcatNodeName in the configuration file.

6.1.7 Checking for forwarded BMC alerts in ESS

To check the power status of the nodes in ESS the following command can be used: `mmhealth node show POWERHW` To see more specifics of that output `mmhealth node show POWERHW -v`

6.2 Viewing system log information about the crassd service

The ibm-crassd service will report alerts into the Linux system logs. These can be viewed with the following command: `journalctl -u ibm-crassd`

CHAPTER 7

Guidelines for creating a new plugin

7.1 Mandatory functions

1. initialize() function to test basic connection to the location for pushing alerts to
2. notify<Endpoint> function. This is called by ibm-crassd to push the alert to the endpoint

7.2 Data Format for the ibm-crassd structures

7.2.1 Event

The following are the common properties that appear on every event. Many of these properties can be customized in the openbmctool policy table.

```
{
    "CerID": "string. EventID which, can be found in IBM Knowledge Center per MTM",
    "message": "string. Description of the problem",
    "logNum": "string. logPostion as a number",
    "severity": "string. Severity of the event. One of Critical, Warning, Info",
    "subSystem": "string. Description of the part of the system impacted",
    "eventType": "string. Description of the type of event",
    "serviceable": "string. Indicates if a human needs to take action on the log entry
↪",
    "callHome": "string. Indicates if the alert will be automatically sent to support␣
↪in an environment with the IBM CALL HOME service enabled",
    "compInstance": "integer. Indicates a slot or socket number for the specified␣
↪component",
    "lengthyDescription": "string. A verbose description of the problem",
    "LogSource": "string. The source of the log entry",
    "RelatedEventIDs": "string. A dictionary containing IDs related to this one. Not␣
↪currently used",
    "timestamp": "string. The timestamp for the log entry as a UNIX timestamp␣
↪(Seconds since epoch)",
```

(continues on next page)

(continued from previous page)

```
    "vmMigration": "string. Indicates if workload should be migrated from the system.␣
↪Not currently used."
}
```

7.2.2 Impacted Node

The following information provides details about the node structure. The following detials provide information about the python dictionary used, along with the type.

```
{
  "bmcHostname": "string. Hostname or IP of the BMC",
  "xcatNodeName": "string. Reference description for the node, commonly host side␣
↪hostname",
  "accessType": "string. One of ipmi, openbmcRest, redfish. Redfish not currently␣
↪implemented.",
  "username": "string. Username for the BMC",
  "password": "string. Password for the specified BMC username",
  "dupTimeIDList": "list of strings. List of IDs that occurred most recently with the␣
↪same timestamp",
  "lastLogTime": "integer. Timestamp of the last log(s) entry. UNIX timestamp",
  "pollFailedCount": "integer. Count of the number of times a bmc poll has failed.␣
↪Resets after a successful poll"
}
```

7.2.3 entityAttr

This contains a set of information for the endpoint that is being reported to. ibm-crassd tracks the status of endpoints where alerts are pushed to, and the plugin is required to update these attributes appropriately.

```
{
  "function": "pointer. A pointer to the notify function",
  "receiveEntityDown": "Boolean indicator if the plugin is able to contact the␣
↪receiving entity. For example, not network reachable.",
  "failedFirstTry": "Boolean. if the first attempt to send the alert has failed. The␣
↪plugin should attempt at least twice.",
  "successfullyReported": "Boolean. True, if the alert was successfully reported to␣
↪the endpoint. If successful, receiveEntityDown should be set to False, and also␣
↪failedFirstTry should be set to False"
}
```

7.3 Creating Configurable options

1. Add options to the ibm-crassd.config file, creating a section for your plugin name. ex: `[pluginName]`
2. import the config module to your plugin.
3. Plugin variables will come in three pieces found in the following sections with detailed descriptions and informaiton about accessing them.
4. Configuration options should be added to the ibm-crassd.config file, or create your own config file in your plugin's directory and load the settings during plugin initialization.

7.3.1 Plugin Configuration Values from config file

These settings are stored into a dictionary in the publicly available config variable. This is done to support the multi-threading approach with ibm-crassd's design. Accessing these is simple by using the following syntax.

```
mySetting = config.pluginConfigs['pluginName']['mySetting']
```

These settings should be validated in the plugin initialization section. Invalid settings for your plugin should halt ibm-crassd on startup.

7.3.2 Plugin Policies for alerts

This setting is stored into a dictionary for use by multiple threads. The policy is loaded from the file system one time and cached for future use. Policy updates require a restart of the ibm-crassd service. The policy table should be loaded during the initialization of the plugin. Failures should halt the ibm-crassd service.

```
myPolicy = config.pluginConfigs['pluginName']['myPolicy']
```

7.3.3 Storing plugin Variables for global use

If variables for your plugin need to be stored so they are centralized for access across multiple threads, the following mechanism can be used.

```
myVar = config.pluginVars['pluginName']['myVariable']
```

It is the responsibility of the plugin to use proper thread locks when setting this variable outside of the initialization.

```
import config
#some code processing an alert
#=========================================================
with config.lock:
  config.pluginVars['pluginName'][myVariable] = myValue

#=========================================================
#Continue with operations
```

7.4 Reporting Errors to the system journal

If your plugin is experiencing issues or needs to write debug messages to the system journal a mechanism is provided. Keep in mind debug messages should use the syslog `LOG_DEBUG` severity, and will only appear in the journal if the ibm-crassd configuration has debug messages available.

To create a journal entry, a simple method is called with two variables. The first is the syslog severity, which uses the syslog library values such as `LOG_INFO` and a string describing the problem or statement being made.

The following is an example of creating a log entry from a plugin:

```
import config
# Missing configuration for a remote server
config.errorLogger(syslog.LOG_ERR, "Unable to find configuration for the remote␣
↪logstash server. Defaulting to 127.0.0.1:10522")
```

CHAPTER 8

Creating a listener for streamed sensor readings

This section discusses creating a listener for metric (telemetry) streaming. This includes filtering the data received and setting the frequency of updates.

8.1 Overview

The ibm-crassd service will host collected sensor readings on a socket. This socket is setup to listen on all IP addresses for where the Host OS is installed, and uses a predefined port number. This port number is configurable in the ibm-crassd.config file. When ibm-crassd has started the telemetry service and is ready to receive a connection from a client, a system journal entry is posted indicating the telemetry service has started. It's at this point a client may connect to the socket server and start listening to the stream of readings. Additionally, once connected filters can be applied. The data is sent from ibm-crassd to listeners in a serialized JSON format with a 4-byte header. The ibm-crassd service will also handle re-connection attempts to dropped BMC connections, should a connection to the BMC be lost. Until the connection is re-established, **ibm-crassd will forward the last known reading of the sensors**. A full example of a python metric listener can be found in the examples directory on the github repository.

8.2 Data Structure Review

The sensor data is formatted in a dictionary at the top level. Most of this data is accessed directly using a combination of the reference name for the node `xcatNodeName` from the configuration file, and the name for the sensor. The AC922 systems have a maximum of 111 sensors. Below is a generic example showing dictionary representation.

```
{
    "Time_Sent": "String. Time in seconds since UNIX epoch when the sample was sent",
    "xcatNodeName": {
        "sensorName1": {
            "scale": "Float. Indicates what to multiple the value by.",
            "value":  "Integer. The reading of the sensor.",
            "type": ["string. Description of the sensor.", "string. Units for the␣
↪sensor."]
```

(continues on next page)

(continued from previous page)

```
        },
        "LastUpdateRecieved": "String. Time in seconds since epoch, when the last␣
↪update from the BMC was received.",
        "Node State": "String. The state of the node. Can be Powered On or Powered Off
↪",
        "Connected": "Boolean. true if the connection with the BMC is active,␣
↪otherwise false."
    }
}
```

The following is an example of a single node and two sensors.

```
{
    "Time_Sent": "1563222466",
    "compute1": {
        "cpu0_temp": {
            "scale": 0.001,
            "value": null,
            "type": ["temperature", "Celsius"]
        },
        "total_power": {
            "scale": 1,
            "value": 161,
            "type": ["power", "Watts"]
        },
        "LastUpdateReceived": "1563222466",
        "Node State": "Powered Off",
        "Connected": true
    }
}
```

8.3 Connecting to the Telemetry Server

Connection to the telemetry server has been simplified. All an application needs is to open a socket to the configured port number and the IP address of the system ibm-crassd is running on. Once the socket is opened, ibm-crassd will begin sending sensor readings once every second to the client. A client can then provide filters or frequency adjustments which will be discussed in the next section. Below is a simple connection created to ibm-crassd from a listener on the same server.

```
import socket
import struct
import json

HOST = '127.0.0.1'   # Standard loop-back interface address (localhost)
PORT = 53322         # Port to listen on (non-privileged ports are > 1023)
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
print("Connecting to ", HOST)
s.connect((HOST, PORT))
print('connected')
```

8.4 Processing the Data Received from the Telemetry Server

Once the connection is setup as shown above we can begin processing the packets received from the ibm-crassd service. There's 2 pieces to deal with from the data packets. The first is the header. The header contains the size of the JSON structure. The struct library is used to unpack the header and get the message length. Below is an example of reading this:

```python
def recvall(sock, n):
    """
        Helper function to receive n bytes or return None if EOF is hit
        @param sock: The socket to read data from
        @param n: The number of bytes to read from the socket
        @return: The data that was received and is ready to process
    """
    data = b''
    while len(data) < n:
        packet = sock.recv(n - len(data))
        if not packet:
            return None
        data += packet
    return data

def crassd_client(servSocket, sn):
    """
        Function to manage the opened socket with ibm-crassd
        @param servSocket: The socket to read data from and write filters to
        @param sn: The hostname/IP of the service node running ibm-crassd
    """
    raw_msglen = recvall(servSocket, 4)
    if not raw_msglen:
        break
    msglen = struct.unpack('>I', raw_msglen)[0]
```

Next we need to continue in the crassd_client function, and get the actual sensor readings. The code snippet below uses the length collected from the header to retrieve all of the data that was sent, from the socket buffer. The data is then loaded into a dictionary to prepare for usage, using the JSON.loads function.

```python
def crassd_client(servSocket, sn):
    # continuation from above
    data = recvall(servSocket, msglen)
    if not data:
        break

    sensData = json.loads(data.decode())
```

8.5 Filtering the Data ibm-crassd Sends

The telemetry server offers a few different options for filtering the data it sends to the subscribed clients. The following are a list of filtering options in prioritized order. These sensor filters can be changed and updated at any time with an active connection.

1. Sensor name - A sensor name, or a list of sensor names can be passed to ibm-crassd, and it will only return readings for sensors that match the name. This has the highest priority.

2. Sensor type - The sensor type, one of power, voltage, current, fan_tach, and/or temperature. These types can be provided in a list, and ibm-crassd will only send readings for those types.
3. Frequency - This option tells ibm-crassd how often to send sensor updates in seconds. This is provided as an integer greater than or equal to one.

It is very important to note that the sensor names and sensor types must be sent as a list, even if it is only one item.

Below is a python example of the client sample above sending filtering options. It's setting the frequency of updates to once every 3 seconds, and only getting sensor types of power.

```python
def crassd_client(servSocket, sn):
    # continuation from above
    sensfilters = {'frequency': 3, 'sensortypes': ['power']}
    data2send = (json.dumps(sensfilters, indent=0, separators=(',', ':')).replace('\n
↪','') +"\n").encode()
    msg = struct.pack('>I', len(data2send)) + data2send
    servSocket.sendall(msg)
```

CHAPTER 9

Building ibm-crassd

This section will provide instructions for building ibm-crassd.

CHAPTER 10

Developer's Guide

10.1 ibm-crassd.py

This is the main module of the ibm-crassd service. It is responsible for setup, loading the configuration, starting the plugins, and then monitoring the defined BMCs for new log entries.

10.2 config.py

This module contains shared data structures used by ibm-crassd and the various other modules in the project.

`config.`**`errorLogger`**`(`*severity*, *message*`)`

Used to handle creating entries in the system log for this service

Parameters

- **`severity`** – the severity of the syslog entry to create
- **`message`** – string, the message to post in the syslog

CHAPTER 11

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index