How to Implement the Sonexion REST API and Correlate it with SEDC and Other Data

Cary Whitney
National Energy Research Scientific Computing (NERSC)
Lawrence Berkeley National Laboratory
Berkeley, CA USA
clwhitney@lbl.gov

***Abstract*—This document describes how to write a python plugin to the Sonexion SeaStream Lustre filesystem. Also discussed is how to correlate the Lustre data with Cray's System Environment Data Collections (SEDC) data to start and give a better view into what is happening on the system.**

Keywords-component; lustre; SeaStream; Sonexion; metric; plugin

I. INTRODUCTION

SeaStream is the REpresentational State Transfer (REST) Application Programming Interface (API) that Seagate created for the Sonexion system. This was the replacement for the Lustre Monitoring Tool (LMT) data collection method. At this time, there is a basic Software Development Kit (SDK) to access this data. I will be presenting a bit on the structure on how to access the data and what we have done with it. I've included a lot from the setup README into this paper for completeness.

II. GENERAL DESCRIPTION

First, there are collectors on every Object Storage Service (OSS) and Metadata Service (MDS) within the filesystem structure. Data is then sent to the Cluster head node. This is the node that the SDK connects to gather the data. LMT also used a similar method of having a collector on each system and sending its data to a central location. Under LMT, that central store tends to be a Structured Query Language (SQL) server and connecting to that database was the way to access the data. The SQL server tends to be the pinch point in large systems. Now with SeaStream, the stored data is held for a shorter period of time and the SDK just polls the data store.

Figure 1 presents our desired end result. This dashboard allows one to select between the three different filesystems available on Edison while presenting all the jobs being run by the listed user ID. The start and end time for each job is listed along with the number of nodes the job is running and the node list. Unfortunately, there can be no more that eye correlation between job execution and Lustre performance. That correlation will require jobstats to be available.

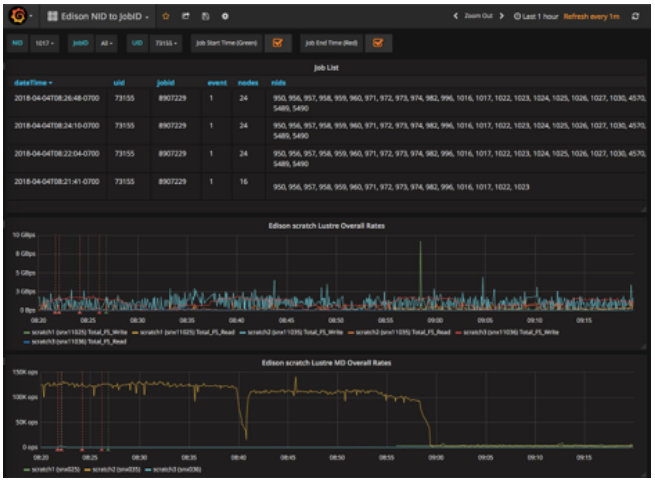

Figure 1. Combined Grafana Dashboard

III. SEASTREAM SDK

A. Getting the SDK

First, ask for the latest SDK from Cray. This is reviewed on a case by case basis since the site needs to take on a little more responsibility. The current version is Release 2.1.25 RAS-685 SDK. Once downloaded it comes with a good README file and three examples. The SDK is written in Python and the examples plugins are for Graphite, OpenTSDB and InfluxDB. I have used the OpenTSDB example extensively as my base.

SeaStream was enable in 2.0 SUXX and 3.0 SUXX. This is the initial release levels. We have discovered an issue with gathering data from Cori's filesystem which should be fixed in 2.0 SU28 and 3.0 SU11. Follow the README file or the instructions here to set up the Sonexion.

B. Setting up the Sonexion Cluster

In Figure 2 we first have to check to see if the Cluster Store admins module is present. If the command displays 'Invalid command "admins"' the admins module is not present and the alternative method listed in the SDK's README file should be used to create the streaming user.

Since the admins module does exist, the few commands listed in Figure 2 will create and enable the SeaStream service.

C. Installing the SDK

Unzip the source into a location where it will be installed into.

```
#> cscli admins
cscli: Invalid command "admins"

#> cscli admins add --username=streamapi --role=readonly --disable-ssh —enable-web
Enter the password :
Confirm the password :

#> cscli service_console configure rest_api enable
REST API has been enabled
#> cscli service_console configure rest_api  user_add --username=streamapi
User 'streamapi' has been added to REST API authorized users list
#> cscli service_console configure rest_api show
REST API access: enabled

REST API authorized users:

  streamapi

Enabling seastream.
```

Figure 4. Streaming account creation

```
Required
  python 2.7.5 or greater
  python-twisted-core-12.2.0-4.el7.x86_64.rpm

Possible additional software
  m2crypto-0.21.1-17.el7.x86_64.rpm
  pyserial-2.6-6.el7.noarch.rpm
  python-fpconst-0.7.3-12.el7.noarch.rpm
  python-twisted-web-12.1.0-5.el7_2.x86_64.rpm
  SOAPpy-0.11.6-17.el7.noarch.rpm
```

Figure 3. Software Requirements

```
{
    "hosts": ["https://1.2.3.4", “https://1.2.3.5”],
    "port": 443,
    "user": "streamapi",
    "password": "SuperSecretPassword",
    "stream": "fs_stats",
    "plugin_mode": "test_formatter",
    "dbhost": "ipaddress",
    "dbport": “4242"
}
```

Figure 2. Opentsdb_fs_stats.json

Next Figure 3 lists the requirements for SeaStream; the additional software was required since I was installing the

```
#> cd seastream
#> python setup.py install
#> cd ..
#> cd plugins/plugin_support
#> python setup.py install
#> cd ..
```

Figure 5. SeaStream Installation Commands

SDK on the ES management server. Also Figure 4 lists the process to install the SDK.

D. Verifying the configuration and software

At this point, SeaStream has been enabled and the SDK installed. We now want to test to make sure we have a valid data path. We will use the OpenTSDB example plugin to verify that we can see data from the Sonexion, but first we need to modify the OpenTSDB configuration file a little.

Figure 5 shows the OpenTSDB configuration file. The black section is the SeaStream connection section. The red line is the debug statement and the blue lines are for the plugin use.

Changes to the OpenTSDB configuration file are:

1. Change the “hosts” IP address to the IP address of the Sonexion head node.
2. Change the “user” and “password” to the account and password that was created earlier.
3. Make sure “plugin_mode” is in the configuration file and set to “test_formatter”. This debug statement disables all functions of the plugin portion of the configuration file. We do not need to edit the “dbhost” or “dbport” lines.
4. Now run the plugin, seastream_opentsdb.py
5. Figure 6 shows an example of output that should be seen.

E. Configuration file decoded

Figure 7 is what will become the RabbitMQ configuration file and we will use it to expand what we already know about the file.

```
#> python seastream_opentsdb.py -c opentsdb_fs_stats.json
Streaming data from the fs_stats stream
[{'timestamp': 1522785624, 'metric': u'notify', 'value': 0.0, 'tags': {'category': 'fs_stats', 'host': u'snx11039n003', 'target':
'MDT0000', 'filesystem_name': u'snx11039', 'target_type': u'MDT', 'system': '', 'unit': u'Ops/sec', 'filesystem_type':
'lustre'}}]
[{'timestamp': 1522785624, 'metric': u'mknod', 'value': 0.0, 'tags': {'category': 'fs_stats', 'host': u'snx11039n003', 'target':
'MDT0000', 'filesystem_name': u'snx11039', 'target_type': u'MDT', 'system': '', 'unit': u'Ops/sec', 'filesystem_type':
'lustre'}}]
[{'timestamp': 1522785624, 'metric': u'mkdir', 'value': 0.0, 'tags': {'category': 'fs_stats', 'host': u'snx11039n003', 'target':
'MDT0000', 'filesystem_name': u'snx11039', 'target_type': u'MDT', 'system': '', 'unit': u'Ops/sec', 'filesystem_type':
'lustre'}}]
```

Figure 6. Sample debug output from OpenTSDB plugin

1) Black section

This is the common block of configuration data associated with connecting to the Sonexion cluster.

- hosts is the Cluster's head node. Listing multiple hosts as a list allows the plugin to follow the data if the head nodes fails over to it backup.
- port is the default port to query the data.
- user/password are the account and password create in Cluster Store and allow access to the data stream.
- stream is what data is being desired from Cluster Store.
    - fs_stats are the filesystem statistics.
    - node_stats are the hosts metrics.
    - jobstats will be the extended filesystem metrics based on job ID.

```
{
    "hosts": ["https://1.2.3.4"],
    "port": 443,
    "user": "streamapi",
    "password": "SuperSecretPassword",
    "stream": "node_stats",
    "plugin_mode": "xtest_formatter",
    "mqhostname": "rabbit.server.domain",
    "mqport": 1234,
    "exchange": "rabbit-exchange",
    "routingkey": "rabbit.routing.key",
    "mqsystem": "system name",
    "mount": "mount point label",
    "type": "lustre",
    "login": "rabbitAccount",
    "mqpassword": "rabbitPassword"
}
```

Figure 7. RabbitMQ rabbitmq_node_stats.json

2) Red section

There is currently only one line in this section and that is the plugin_mode which is set to test_formatter when enabled. I insert an 'x' as the first character of the string to disable debugging but allow me to remember the configuration line.

3) Blue section

This section lists the configuration entries needed for the plugin. We are using the configuration file to pass in RabbitMQ information and other data to enrich the data stream.

- mqhostname/mqport is the RabbitMQ host and port.
- exchange is the RabbitMQ exchange the data will be sent to.
- routingkey this will be the path through RabbitMQ.
- login/mqpassword is needed since we authenticate to our RabbitMQ instance, we need to pass this also into pika.
- mqsystem/mount/type: all three of these are data enrichment points that are added to each data point being sent. We use this to distinguish the data.
    - type is our data types. This is 'lustre'.
    - mount is the common mount point.
    - mqsystem is the Lustre Cluster name since we have four different Clusters.

F. Debug out of RabbitMQ

At this stage let's take a look at what the RabbitMQ plugin will give us. Figure 8 shows the output in JavaScript Object Notation (JSON) form of the node_stats instead of fs_state from OpenTSDB. The purple metrics are items that we are enriching the data via the configuration file.

G. Available metric fields

The current available metrics for each stream for SeaStream are node_stats listed in Table I, while fs_stats are in Table II in two parts with target_type Object Storage Target (OST) metric data for the storage side and target_type MetaData Target (MDT) for metadata operations. One caveat is the data can change between versions since the software is still in development. With Elastic free form storage, the plugin will pick up any changes in number of metrics presented or metric name changes and store them. The issue is them pushed to the display device, aka the graphic labels.

```
#> python seastream_rabbitmq.py -c rabbitmq_snx11039_node_stats.json
Streaming data from the node_stats stream
{"category": "node_stats", "host": "snx11039n003", "mount": "scratch1", "unit": "avg", "timestamp": "2018-04-
03T14:45:13+0000", "metric": "Processor_load_5_min", "identifier": "", "system": "snx11039", "value": 0.0194}
{"category": "node_stats", "host": "snx11039n003", "mount": "scratch1", "unit": "percent", "timestamp": "2018-04-
03T14:45:56+0000", "metric": "CPU_Usage", "identifier": "", "system": "snx11039", "value": 0.0}
{"category": "node_stats", "host": "snx11039n003", "mount": "scratch1", "unit": "percent", "timestamp": "2018-04-
03T14:45:43+0000", "metric": "CPU_idle_time", "identifier": "", "system": "snx11039", "value": 99.6218}
```

Figure 8. Debug output from RabbitMQ plugin showing the JSON output

TABLE I. NODE_STATS METRICS

Metric	Unit
Memory_Usage	percent
Memory_Free	bytes
Memory_Cached	bytes
Memory_Buffers	bytes
CPU_Usage	percent
CPU_idle_time	percent
CPU_iowait_time	percent
CPU_nice_time	percent
CPU_system_time	percent
CPU_softirq_time	percent
CPU_user_time	percent
CPU_Interrupts	ips
Processor_load_1_min	avg
Processor_load_5_min	avg
Processor_load_15_min	avg

TABLE II. FS_STATS METRICS

OST Metric	Unit	MDT Metric	Unit
read	bytes/sec	available_space	bytes
write	bytes/sec	free_indes	inodes
available_space	bytes	free_space	bytes
total_space	bytes	open	ops/sec
free_space	bytes	close	ops/sec
free_inodes	inodes	mkdir	ops/sec
total_inodes	inodes	rmdir	ops/sec
		link	ops/sec
		unlink	ops/sec
		create	ops/sec
		destroy	ops/sec
		connect	ops/sec
		disconnect	ops/sec
		getattr	ops/sec
		getxattr	ops/sec
		setattr	ops/sec
		rename	ops/sec
		notify	ops/sec
		mknod	ops/sec
		statfs	ops/sec
		quotactl	ops/sec
		process_config	ops/sec
		llog_init	ops/sec

IV. RABBITMQ PLUGIN

Since the SDK has several good examples of plugin code, I basically started with the OpenTSDB plugin and modified it to present RabbitMQ. This ends up being a fairly easy task. The plugin name is: seastream_rabbitmq.py.

First, we need to make sure to include the pika module to gain access to the RabbitMQ libraries for python.

```
import pika
```

Now the example plugin has only two basic classes:

- PluginStreamerConsumer - this does not change since the RabbitMQ plugin only writes out the text similar to the debug statement.
- SeaStreamRabbitMQ - which is renamed from SeaStreamOpenTSDB

A. *PluginStreamerConsumer*

Little if anything changes in this class. For our purposes, it mainly provides a hook to call the metric parsing functions.

1) class PluginStreamerConsumer(AbstractStreamerConsumer):

- def __init__(self):
- Gathers system information from the Cluster.
- def insert(self, element):
- Hook to perform any action that may need to be done when a new data type is seen by the plugin.
- def update(self, element):
- This hook would be called when any new data is received for a metric that is already being collected.
- def delete(self, element):
- The delete hook removes a collected element from the stream.
- def process_metrics(self, element):
- The process_metrics hook has these functions:
  1. If the stream type is ‘fs_stats’, call **parse_fs_stats**.
  2. If the stream type is ‘node_stats’, call **parse_node_stats**.
  3. If the stream type is ‘jobstats’, call **parse_jobstat**.
  4. Since only one of these will be called at any one time, if it succeeds then the process **send_metric** is called.

B. SeaStreamRabbitMQ

This class is where the changes take place. Most of the changes is in send_metric with a few changes to deal with extra data in each of the parse definitions.

- < is SeaStreamRabbitMQ
- > is SeaStreamOpenTSDB

1) class SeaStreamRabbitMQ(object):

- def __init__(self, options):
  Figure 9 sets up the plugin variables

```
def __init__(self, options):
144,152c142,143
<         self.mqhostname = options.get('mqhostname')
<         self.mqport = options.get('mqport')
<         self.exchange = options.get('exchange')
<         self.routingkey = options.get('routingkey')
<         self.mqsystem = options.get('mqsystem')
<         self.mount = options.get('mount')
<         self.type = options.get('type')
<         self.login = options.get('login')
<         self.mqpassword = options.get('mqpassword')
---
>         self.dbhost = options.get('dbhost')
>         self.dbport = options.get('dbport')
155c146
<         for var in ['stream', 'mqhostname', 'mqport',
'exchange', 'routingkey', 'mqsystem', 'mount', 'type',
'login', 'mqpassword']:
---
>         for var in ['stream', 'dbhost', 'dbport']:
```

Figure 10. Setting up the variables for the plugin

- def get_stream_type(self):
- def get_systemname(self):
- def get_mount(self):
  These three definitions return their associated values. The systemname and mount have been added to address the additional information needed for RabbitMQ and data enrichment.
- def connect(self):
  Figure 10 is where connection checking was removed and should be added back in. Right now it just connects to RabbitMQ via pika.
- def disconnect(self):
  No change.
- def send_metrics(self, metrics):
  Figure 11 is part of the debug print out when test_formatter is set. The lower section is the publish command to send the data to RabbitMQ. There is no reconnect code yet. This will be the way to kill the existing process and restart it if RabbitMQ goes away.
- def parse_node_stats(self, data, system_identifier):
  Figure 12 Set our enrichment variables and our timestamp format. Then converts the output data format to remove the OpenTSDB tag structure.

```
def connect(self):
199,233c184,194
<         # loop until mqhostname is reachable removed
<
<         self.creds = pika.PlainCredentials (self.login,
self.mqpassword)
<         self.conn = pika.BlockingConnection
(pika.ConnectionParameters(host = self.mqhostname,
credentials = self.creds))
<         self.channel = self.conn.channel()
<         self.channel.exchange_declare
(exchange=self.exchange, exchange_type='topic',
durable=True)
<
235,236c196,208
<         # CLW this is forced True and will be changed
when the above gets fixed.
<         self.connected = True
```

Figure 9. Pika connection setup

```
def send_metrics(self, metrics):
250,253c222
<         #print (metrics)
<         for metric in metrics:
<             data=json.dumps(metric)
<             print (data)
---
>         print (metrics)
259,268c228,233
<             data=json.dumps(metric)
<             self.channel.basic_publish (exchange='%s'
% (self.exchange),
<                 routing_key='%s' % (self.routingkey),
<                 body = '%s' % (data))
<         # CLW Dropped OpenTSDB code
272,276c237
<             format(self.mqhostname, self.mqport),
'Trying to reconnect')
<         # CLW Added sleep here instead of in the
connection area since I do not have it working there
<         #  Here I am assuming (maybe incorrectly) that
the connection dropped since the service is
<         #  temporarily down.  Thus wait and then try
the reconnect.
<         time.sleep(10)
---
>             format(self.dbhost, self.dbport), 'Trying to
reconnect')
```

Figure 11. RabbitMQ publishing code

- def parse_fs_stats(self, data, system_identifier, fs_object_stores):

```
def parse_node_stats(self, data, system_identifier):
291,292d251
<         mqsystem = plugin.get_systemname()
<         mount = plugin.get_mount()
297,299d255
<         # 2016-10-31T09:43:56-0700
<         mqtime = time.strftime("%Y-%m-
%dT%H:%M:%S%z", time.localtime(timestamp))
<
321c276
<             'timestamp': mqtime,
---
>             'timestamp': timestamp,
323,328c278,283
<             'system': mqsystem,
<             'mount': mount,
<             'host': hostname,
<             'category': 'node_stats',
<             'unit': units,
<             'identifier': system_identifier,
---
>             'tags': {
>                 'system': system_identifier,
>                 'host': hostname,
>                 'category': 'node_stats',
>                 'unit': units
>                 }
```

Figure 13. Changes to node_stats

- Figure 13 The same logic is used by fs_stats as above for node_stats. There are two stanzas in this since there looks to be a format change in one of the SDK versions. We will follow along.
- def parse_jobstat(self, data, system_identifier):
- Figure 14 is the placeholder for the jobstats section of the code and should work when jobstats is available.

V. RESULTS

Figures 15 and 16 are using Grafana to display all of the available data from the plugin. This is a long dashboard and is viewed in the two graphics. I have removed some dashboard panels since the data is not very interesting. I can also select between the different filesystem by the pull-down box in the upper left corner.

Figure 1 at the beginning is also from Grafana and is annotated with SEDC job data. Even though we use SLURM, the alps library is still currently used underneath things. The alps library still generates a job start/stop entry and the nodes used in the SEDC data. I can then query that index and add it to the dashboard. (Caveat, I cannot correlate what data profile is used by what job. One would need jobstats for that. I can only visualize it.)

```
def parse_fs_stats(self, data, system_identifier,
fs_object_stores):
343,347d297
<         mqsystem = plugin.get_systemname()
<         mount = plugin.get_mount()
<
<         # 2016-10-31T09:43:56-0700
<         mqtime = time.strftime("%Y-%m-
%dT%H:%M:%S%z", time.localtime(timestamp))
361,363c310
<                 'system': mqsystem,
<                 'mount': mount,
<                 'timestamp': mqtime,
---
>                 'timestamp': timestamp,
365,372c312,321
<                 'indentifier': system_identifier,
<                 'filesystem_name': filesystem_name,
<                 'filesystem_type': 'lustre',
<                 'target': target,
<                 'target_type': target_type,
<                 'category': 'fs_stats',
<                 'host': host,
<                 'unit': units
---
>                 'tags': {
>                    'system': system_identifier,
>                    'filesystem_name': filesystem_name,
>                    'filesystem_type': 'lustre',
>                    'target': target,
>                    'target_type': target_type,
>                    'category': 'fs_stats',
>                    'host': host,
>                    'unit': units
>                 }
381,383c330
<             'timestamp': mqtime,
<             'system': mqsystem,
<             'mount': mount,
---
>             'timestamp': timestamp,
385,389c332,338
<             'identifier': system_identifier,
<             'filesystem_name': filesystem_name,
<             'filesystem_type': 'lustre',
<             'category': 'fs_stats',
<             'unit': units
---
>             'tags': {
>                 'system': system_identifier,
>                 'filesystem_name': filesystem_name,
>                 'filesystem_type': 'lustre',
>                 'category': 'fs_stats',
>                 'unit': units
>             }
```

Figure 12. Changes to fs_stats

```
def parse_jobstat(self, data, system_identifier):
404,405d352
<         mqsystem = plugin.get_systemname()
<         mount = plugin.get_mount()
408d354
<         # CLW
411c357
<             job_name, attribute, mqtime, metric_value,
units = result
---
>             job_name, attribute, timestamp, metric_value,
units = result
413c359
<             new_metric = [{
---
>             new_metric = {
415c361
<                 'timestamp': mqtime,
---
>                 'timestamp': int(timestamp),
417,429d362
<                 'identifier': system_identifier,
<                 'system': mqsystem,
<                 'mount': mount,
<                 'filesystem_name': filesystem_name,
<                 'filesystem_type': 'lustre',
<                 'target': target, 'jobid': job_name,
<                 'category': 'jobstats',
<                 'unit': units,
<                 'ndc': {
<                     'system': mqsystem,
<                     'mount': mount,
<                     'type': 'lustre'
<                     },
431,434c364,369
<                 'ndc',
<                 'lustre',
<                 mqsystem,
<                 mount
---
>                 'system': system_identifier,
>                 'filesystem_name': filesystem_name,
>                 'filesystem_type': 'lustre',
>                 'target': target, 'jobid': job_name,
>                 'category': 'jobstats',
>                 'unit': units
436c371
<             }]
---
>             }
```

Figure 14. Changes to parse_jobstat

Figure 15. First half of the dashboard showing most of the information from SeaStream

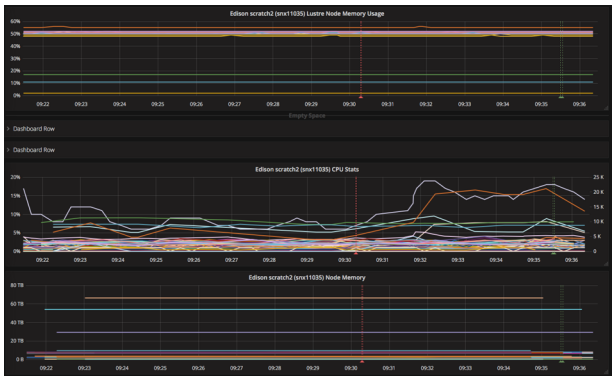

Figure 16. Second half of the dashboard showing most of the information from SeaStream

VI. Bringing it All Together

The main point here is; can others use this? Can others create something that they get value from? Hopefully with the following resources to help one create a SEDC plugin and this for Lustre data, a Grafana dashboard like the one opening in this paper can be created. The JSON is listed at the end of the paper and will be available for download. Since Grafana has several data sources that it supports, one could with minimal effort, create a SEDC plugin to store that data into OpenTSDB and just use the example OpenTSDB plugin from SeaStream, take the JSON dashboard and either create data sources of the same name or create the data sources and change the name in the JSON. Then a functioning dashboard can be view and compared with other organizations.

VII. The Future

A. SEDC Resources

SEDC plugin example is a CSV. [1] is a reference to get more information on how to write one. [2] refers to a paper talking about how to write a SEDC plugin that sends data to either RabbitMQ or Kafka.

```
{
 "annotations": {
  "list": [
   {
    "builtIn": 1,
    "datasource": "-- Grafana --",
    "enable": true,
    "hide": true,
    "iconColor": "rgba(0, 211, 255, 1)",
    "name": "Annotations & Alerts",
    "type": "dashboard"
   },
   {
    "datasource": "Edison-SEDC",
    "enable": true,
    "hide": false,
    "iconColor": "#629e51",
    "limit": 100,
    "name": "Job Start Time (Green)",
    "query": "event:1 AND jobid:$jobid",
    "showIn": 0,
    "tags": [],
    "textField": "jobid",
    "timeField":
```


Figure 17. Beginning of the Grafana JSON exported dashboard.

B. *Additional Metrics*

I've started to investigate additional metrics to gather in the hope to build a better understanding of Lustre and a way to debug and troubleshoot filesystem issues.

C. *Grafana JSON Dashboard*

The Grafana JSON dashboard file for the annotated dashboard in Figure 1 is displayed in two partial part in Figures 17 and 18. The full dashboard file is 10 pages long. This JSON code could be imported into another Grafana instance and possibly the only changes would be the "datasource" in Figure 17 to point to what is available at your site. Then the multiple "query" also in Figure 17 may have to be change to make sure the same data is gathered at each panel. This would be easier than creating from scratch and would offer a consistent view between organizations.


```
  "timepicker": {
   "refresh_intervals": [
    "5s",
    "10s",
    "30s",
    "1m",
    "5m",
    "15m",
    "30m",
    "1h",
    "2h",
    "1d"
   ],
   "time_options": [
    "5m",
    "15m",
    "1h",
    "6h",
    "12h",
    "24h",
    "2d",
    "7d",
    "30d"
   ]
  },
  "timezone": "",
  "title": "Edison NID to JobID",
  "version": 11
}
```


Figure 18. Ending of the Grafana JSON exported dashboard.

REFERENCE

[1] S. Martin, "Cray XC Power Monitoring and Management Tutorial", CUG 2016, https://ssl.linklings.net/conferences/cug/cug2016_program/views/includes/files/tut103s2-file1.pdf

[2] S. Martin, D Rush, M Kappel and C Whitney, "How-to write a plugin to export job, power, energy, and system environmental data from your Cray XC system", https://cug.org/proceedings/cug2017_proceedings/includes/files/pap147s2-file1.pdf
https://cug.org/proceedings/cug2017_proceedings/includes/files/pap147s2-file2.pdf