

CERN Summer Student Project Report

Author: Jozsef Makai

Supervisor: Pedro Andrade

Team: IT Monitoring team (IT department)

September 2015

IT Monitoring team

CERN is running two large data centres, one in Geneva and a second in Budapest, providing over 120.000 cores, 180 PB of disk space, 100 PB of tape space, and 150 high performance tape drives to serve CERN's computing and storage needs. The IT Computing Facilities (CF) group is in charge of these data centres and in particular the monitoring of the data centre resources. The IT monitoring team is working on modernising the technologies used for monitoring by integrating and developing open source solutions.

The IT monitoring team is responsible for the supervision of data centers, the main challenge of the team is to provide reliable computing platforms for the organization. In order to achieve this, the team provides various monitoring frameworks and tools that can be easily utilized by any other computing related teams to improve the quality and availability of their services.

First task: HEPSEC data collection

[HEPSEC](#) is a benchmark specialized for the measurement of high energy physics (HEP) computing performance of machines. Today, CERN utilizes this benchmark for the initial testing of machines. However, with the appearance of multi-level virtualization and environment changes (new version of software, hardware aging) the HEPSEC value of a certain configuration can change by time, so we want to measure and collect the HEPSEC value for every machines in the CERN data centers during its lifetime. My task was to collect the data from the machines and pass the data to the data collecting and visualizing systems of the IT Monitoring team. For that, I have implemented a [Lemon sensor](#) that collects the measurement data from the hosts it runs on. The repository for the sensor can be found [here](#). The sensor deletes the content of the file after the data was transmitted to the IT Monitoring systems. This solution may be improved after specification with the teams providing the HEPSEC data.

Second task: cAdvisor

My second task was to investigate the cAdvisor system and to implement a storage driver that sends the measurement data collected by cAdvisor to the backend systems of the IT Monitoring team, like Elasticsearch and HDFS.

Installation

First, the Go development and runtime environment has to be installed for the machine *cAdvisor* will run on. Go can be downloaded for most platforms from [here](#) (but also the *golang* package can be installed by any Linux package managers) can be in and [the following](#) environment variables should be set up. Finally, follow the [instructions](#) to download and build *cAdvisor*, as well (this step can be skipped with our initial [repository](#) set up with *cAdvisor*).

Patching and Additions

Unfortunately, the supported and runnable storage drivers are hard-coded to the [storage_driver.go](#) source file. In order to work with a new, officially not supported storage, we have to apply a [patch](#) (with the initialization of our new driver) to this file and *cAdvisor* should be recompiled. With our initial [repository](#), just download the repository, apply the patch with the *patch* tool and [compile](#) it.

Then *cAdvisor* can be run with the new driver named *flume_http* and with 10 seconds of sampling interval:

```
sudo ./cadvisor -storage_driver=flume_http --global_housekeeping_interval=1m0s
```

```
--housekeeping_interval=10s
```

A brief description of these parameters can be found [here](#) and [here](#).

We also have to add our storage driver implementation to the appropriate folder which in this case should be *storage/flume_http/flume_http.go* from the root directory of *cAdvisor*. If we implement the appropriate methods provided by the *StorageDriver* interface in [storage.go](#) source file then we have our own storage driver and we can start publishing the provided data.

Driver Implementation

In the driver source file, as a minimum requirement, you have to create your own *struct* type and implement the two methods of the *StorageDriver* interface in *storage.go* which in case of *flumeSource* type looks like:

```
func (self *flumeSource) AddStats(ref info.ContainerReference, stats *info.ContainerStats) error
```

This function will be called periodically for each running containers by *cAdvisor* and we get the recent measurement data in the *stats* variable. We send the metric data in this function to *Flume*.

```
func (self *flumeSource) Close() error
```

In this function, we can dispose every resources (e.g. database connections) we used, it will be called when *cAdvisor* is stopped.

The only tricky thing about the JSON we have to send to *Flume* is that the *body* part should be a JSON string instead of JSON object. Therefore, we create a JSON object of the appropriate metrics with the built-in JSON marshaller, convert it to string, escape all “ characters and then place this string to the final JSON string.

The source code of the initial implementation, which works with the *dev Flume* cluster, can be found [here](#).

Metrics

cAdvisor provides metrics about CPU, memory, disk and network usage of the containers. Informations about the metrics can be found in the [source code](#) (search for *InterfaceStats*, *FsStats*, *MemoryStats*, *CpuStats/CpuUsage*, *DiskIoStats*).

The most important CPU metrics are CPU usage of the containers in total, per CPU, in user space and in kernel space. All of them are measured in nanoseconds.

The most important memory metrics are memory usage and memory consumed by the working set of a container, both measured in bytes.

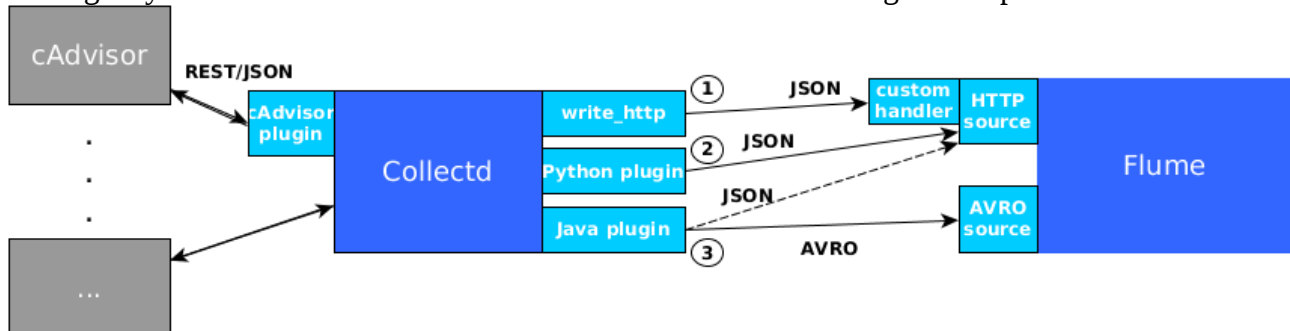
The most important network metrics are bytes/packets sent and received per network interface.

The most important filesystem metrics are the capacity, used/available disk space, all measured in bytes, time spent with IO operations measured in milliseconds.

In this initial prototype, we used the CPU total usage and the memory usage (including working set) metrics as one *lemon* metric.

Third task: Collectd cAdvisor integration and Collectd Flume integration

My third task was to investigate the possibilities to integrate cAdvisor with Collectd and then finding ways to send measurement data from Collectd to Flume using AVRO protocol.



1) write_http plugin and custom Flume handler/interceptor

The *write_http* plugin is a built-in Collectd plugin which can send data in a predefined JSON format. The JSON format the plugin produces is incompatible with the JSON format that the default Flume handler expects. To overcome this problem, we can implement a custom handler for the Flume HTTP source which accepts the data and passes it to the custom interceptor which can process and forward it accordingly.

Custom handler and interceptor can be configured:

```
httpagent.sources.http-source.handler =
ch.cern.itmon.cadvisor.flumeinterceptor.CollectdJsonHandler
httpagent.sources.http-source.interceptors = i1
httpagent.sources.http-source.interceptors.i1.type =
ch.cern.itmon.cadvisor.flumeinterceptor.CustomHttpInterceptor$Builder
```

- Pros:
 - 1) Very easy to implement and to install: dependency handling is easy with Maven and the required JARs (which can be copied to a directory by Maven) are only needed to be copied to the lib directory of Flume.
 - 2) Easy to [configure](#) on Collectd side, basically just need to set the address of the Flume HTTP source.
- Cons:
 - 1) Data processing and transformation is central and not distributed.
 - 2) Data filtering (which metrics to send) can only be done by disabling the collection of certain metrics.
 - 3) Didn't work for the examined cAdvisor Collectd plugin because *write_http* plugin wasn't installed in the container and the package manager didn't work either.

2) Collectd Python plugin with Flume HTTP source

Collectd now embeds a Python interpreter which allows us to forward data to any storage systems. In order to [configure](#), one should load the *python* Collectd plugin, specify the directory location where the implemented plugin resides and import the module.

For the plugin, we have to import the *collectd* module which will be installed with newer Collectd distributions. Furthermore, we have to register callback functions (in a strict order) as it is [documented](#). The *register_write* function allows us to register a callback function that gets each collected metric data and can forward those.

- Pros:
 - 1) Easy to configure, to deploy, easiest dependency handling (just need to install required Python libraries on the system).
 - 2) Distributed data processing/filtering.
- Cons:
 - 1) Only works with JSON, doesn't work with AVRO because of Flume AVRO source and Python AVRO library [incompatibilities](#).

3) Collectd Java plugin with AVRO source

The [Java plugin](#) works similarly to the Python plugin but instead of providing pointers to callback functions, we need to implement the appropriate interfaces provided by the [Collectd API](#). The classes of the API reside in a JAR (usually `/usr/share/collectd/java/collectd-api.jar`) which is also installed with Collectd distributions.

A little trick which is not mentioned but without it the plugin may does not work. We need to set the `LD_LIBRARY_PATH` environment variable to contain the path to the installed `libjvm.so` file. In order to let Collectd automatically do that for us, we can add an `export` command to the `/etc/default/collectd` file.

For example:

```
export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:/usr/lib/jvm/java-7-oracle/jre/lib/amd64/server/
```

It is worth to use the [Flume AVRO client library](#) which provides a higher level API than the [AVRO Java API](#). In that case we can avoid code generation from schema if we use the default Event object of Flume.

To utilize this library, we can add the following Maven dependency (which is in Maven central) to our pom.xml:

```
<dependency>
  <groupId>org.apache.flume</groupId>
  <artifactId>flume-ng-core</artifactId>
  <version>1.6.0</version>
</dependency>
```

- Pros:
 - 1) Works with AVRO and JSON/HTTP.
 - 2) Distributed.
- Cons:
 - 1) More code, need to implement interfaces in Java (it is acceptable with the Flume AVRO client).
 - 2) We need to use a [schema](#) and may need to generate classes according to it (it is no more a problem if we use the default Event schema of Flume and the Flume AVRO client).
 - 3) Hardest to configure, deploy and handle dependencies as you need to list each dependency JARs one by one like (as setting the `$CLASSPATH` environment variable didn't work either):

```
JVMArg "-Djava.class.path=/home/jmakai/collectd/plugins/java/collectd-api.jar:/home/jmakai/collectd/plugins/java/collectd.avro-0.0.1-SNAPSHOT.jar"
```

The “dirty” solution for this problem is to create a fat JAR (importing all dependencies into a huge single jar) with Maven. In that case we have everything packaged into one JAR file, so we have to include a single JAR in the configuration. In order to utilize this feature, we can use the [Maven assembly plugin](#) or for more

control, the [Maven shade plugin](#). Then we can easily use the created fat JAR along with the Collectd provided API like:

```
JVMArg "-Djava.class.path=/home/jmakai/workspace/ch.cern.itmon.collectd.avro/target/collectd.avro-0.0.1-SNAPSHOT-jar-with-dependencies.jar:/usr/share/collectd/java/collectd-api.jar"
```

Cadvisor Collectd plugin

It is a [Collectd plugin](#) written in [Python](#) which reads the data from the REST API of cAdvisor and dispatches the measured data for the *writer* plugins (plugins with registered *write* functions, like *write_http* and other data transmitter plugins). The plugin has its own [configuration file](#) that allows us to precisely control which metrics the plugin collects from cAdvisor, to enable Docker container monitoring, etc.

Selected solution

We have selected the third option (Collectd Java plugin with AVRO source) as a possible solution for the future. While the Java plugin had some drawbacks first, I have managed to find workaround solutions for each of them and managed to find relatively easy ways to deploy and configure that setup. On the other hand, it is a distributed solution which helps to maintain the scalability of the IT Monitoring teams systems. It is also an important advantage of this solution that it supports the AVRO protocol which is the preferred protocol of Flume and the IT Monitoring systems.

Prototype installation details

After investigating and consulting all the possible solutions, we have chosen to create a prototype of the setup with cAdvisor and Java Collectd plugins using Docker containers. The installation, how-to details, and the limitation of the prototype were fully documented on the [documentation site](#) of IT Monitoring team in GitLab.