Factory Monitoring for the 21st Century

Jeffrey Dost[1,*], *Edgar* Fajardo[1,**], *Brian* Bockelman[2], *Leonardo* Hernández-Cano[3], *Caitlin* Hung[1], *Naveen* Kashyap[1], *Frank* Würthwein[1], and *Marian* Zvada[2]

[1]University of California, San Diego, 9500 Gilman Dr, La Jolla, CA 92093, USA
[2]University of Nebraska-Lincoln, 1400 R Street, Lincoln, NE 68588, USA
[3]National Autonomous University of Mexico, Av Universidad 3000, Copilco Universidad, Coyoacán, Ciudad de México, Distrito Federal 04510, Mexico

Abstract. A key aspect of pilot-based grid operations are the GlideinWMS pilot factories. A proper and efficient use of any central block in the grid infrastructure for operations is inevitable, and GlideinWMS factories are no exception. The monitoring package for the GlideinWMS factory was originally developed when the factories were serving a couple of VOs and tens of sites. Today with the factories serving tens of VOs and hundreds of sites around the globe an update of the monitoring is due. Moreover with the new availability of industry open source storage and graphing packages an opportunity remains open. In this work we present the changes made to the factory monitoring to leverage different technologies: Elasticsearch, RabbitMQ, Grafana, and InfluxDB to provide a centralized view of the status and work of several GlideinWMS factories located in different continents around the globe.

1 Introduction

The OSG Glidein Factory [1]is an integral service for CMS and OSG [2]. The factory operations team depends on monitoring information for a variety of tasks such as ensuring the health of the service, making sure pilots are being submitted to sites the end users are requesting, and ensuring the pilots run at sites without issue. The GlideinWMS software [3] comes with standard web-based monitoring pages built in. When the OSG Glidein factory first went live, it was a single instance, and served about four scientific communities or Virtual Organizations(VO). Pilots were all configured to run only in single core mode. Over time, as more VOs joined OSG and adapted GlideinWMS, the scale of the OSG Factory grew to the point where more than one instance was required to run in parallel to serve the load. In addition to scaling in size, eventually CMS and other VOs began experimenting with multicore pilots. This is just one example of new resource types the factory software had to adapt to. Over the lifetime of the OSG Factory as a service, these evolutionary changes were needed to keep up with the demands of the users, but at the same time disrupted the monitoring, and code had to be adjusted to keep up with the changes.

Fundamental changes such as providing multicore pilot metrics were eventually added, but the updates were limited to the case where pilots requested a fixed number of cores, and

*e-mail: jdost@ucsd.edu
**e-mail: emfajard@ucsd.edu

the user jobs either still ran in single core mode, or a single user job would take over all cores of a given pilot. When more sophisticated modes of multicore were adapted such as partitionable slots – which allowed users to dynamically select a subset of cores of a given pilot – again, the monitoring needed to be changed to reflect this new mode of operation.

In addition to proper accounting of core usage, as more factories were spun up, it became evident to the operations team that having individual web pages for each factory would not be sufficient to give a global picture of the OSG Factory as a whole. A long standing request has been made to the GlideinWMS developers to provide an aggregate monitoring solution that combined usage of all factories. However in practice, other more urgent feature requests kept the global monitoring request at a lower priority, and it was never implemented. Over the years it was becoming apparent that a better development workflow was needed to keep up with factory monitoring needs. In 2017 the Open Science Grid agreed to take ownership of the factory monitoring. This would allow the GlideinWMS developers to focus on non-monitoring related development, they would merely need to provide a way to export the raw metrics in a parsable format. At the same time, dedicated staff at UCSD and UNL could work with the Factory Operations team and ensure monitoring needs are met as disruptive changes occur in the evolution of the Glidein factory.

2 Architecture

The new factory monitoring architecture is iteratively being improved and revised, but this section will focus on two implementations, the pre-CHEP version (Sect. 2.2), and the current post-CHEP version (Sect. 2.3). The pre-CHEP version was presented as a poster at CHEP 2018. Significant improvements have been made since, so the discussion on the post-CHEP version will explain what was changed, the reasons for the changes, and how the current implementation has improved as a result. Before getting into the new architecture, first the original GlideinWMS factory monitoring will be discussed in the next section.

2.1 GlideinWMS built-in monitoring

The GlideinWMS built-in factory monitoring is based around the RRDtool database, which is a round robin database intended to store time based series for fixed amounts of time. RRDs are schema based, so unfortunately if any metrics are added or removed to the series, all of the existing RRDs must be updated to account for the change. The factory provides a web server that hosts plots over time of statistics of pilots running at sites all over the globe. For each site queue a corresponding RRD database has to be maintained in the backend. The plots are generated by parsing the RRDs using JavaScript libraries. All of the web pages were created from the ground up, mostly by students. Because of this, it is difficult to make changes to the existing pages. Metrics along with their views, such as layout, colors, and line types are all hardcoded values in the web code, they cannot be customized without committing changes to the GlideinWMS codebase. Also, as stated previously, each factory instance has its own web server containing monitoring pages describing itself only. One factory monitor does not contain information about other factories.

2.2 New monitoring: Pre-CHEP version

The first step toward decoupling factory monitoring from particular instances was to split the data store and user interface from the factory code. For the data store we initially chose InfluxDB because it was created to hold time series data. Its advantage over RRD is that

it is schema-less, so the data series don't need to be defined in advance; we can add or remove metrics as we see fit without committing to a particular layout. On the user interace end, we chose Grafana because it is web-based like the previous monitoring, it supports InfluxDB natively, and its ability to create views including graphs and tables is very flexible and customizable. Both InfluxDB and Grafana were installed on a new monitoring host at UCSD separate from the factories.

The next step was to devise a method to push the data from the factories into InfluxDB. The GlideinWMS factory code already published some pilot statistics in a parsable xml file called `schedd_status.xml`. This xml file is published on the factory web servers and is periodically updated along side the RRD files. It contains current statistics such as counts of running, idle, and held at various site queues. In order to leverage this, we devised a Python script that would periodically run as a cron job on each factory called `factory_monitor.py`. The `factory_monitor.py` script reads in the current values from `schedd_status.xml` and updates the relevant InfluxDB databases on the monitoring host over HTTP. We abstracted this functionality in what we call a messenger module. The idea is that we can write a specific messenger module that knows how to push data to InfluxDB, but then later we can go back and write other messenger modules that can push to other database types, and they would all be envoked the same way.

The above solution works well for plotting the queued pilot statuses over time, however the old GlideinWMS monitoring also contained plots based on completion statistics that were directly pushed to the RRD files without being exported to a status xml file. To reproduce these missing plots, we worked with the GlideinWMS development team and submitted a GitHub pull request to also export the missing completed stats into a new file. These contain information such as how much walltime the pilots spent running, counts of pilot passing or failing validation, the number of user jobs ran inside the pilots, and other runtime characteristics of the user jobs. We export these metrics in a file called `completed_data.json`. We chose JSON format for the missing file because it has become a more commonly used file format for sending data to modern database solutions such as InFlux and Elasticsearch. This allows us to avoid the extra conversion step we need to perform when parsing and sending the `schedd_status.xml` metrics. We have requested that longer term the GlideinWMS team also convert `schedd_status.xml` to a JSON file. Then we can begin using it as the native format for exporting future metrics, so we do not have to perform this extra conversion step whenever we parse monitoring information from the files.

2.3 New monitoring: Post-CHEP updates

The pre-CHEP setup was enough to give us a proof of concept prototype, and we used it to successfully decouple the factory monitoring from the factory code. However, we wanted to take the new monitoring a step further to see if we can push the data to more than one backend. The GRACC is the official monitoring Elasticsearch instance for OSG. It is a natural place to also back up all factory statistics. Planning to also feed the GRACC Elasticsearch revealed limitations in the current setup. In order to publish data to it, one must first coordinate with UNL to push data to a RabbitMQ instance hosted at Nebraska. The UNL operations team handles the path from RabbitMQ into the GRACC Elasticsearch backend. The requirements in updating the new factory monitoring setup is to modify the `factory_monitor.py` script to also push the data to the Nebraska hosted RabbitMQ instance.

To prototype the changes needed to interface with Elasticseach via RabbitMQ, we installed a test Elasticsearch instance on the Grafana monitoring host at UCSD. We set up our own test RabbitMQ instance on a different test machine also hosted at UCSD. Once we had our Elasticsearch test infrastructure ready, the limitation of our architecture became apparent.

Given the task to write and maintain the `factory_monitor.py` script was delegated to students, any time changes were made, the scripts would all have to be updated accordingly on each production factory by the service owners. This is because students aren't allowed root access to the production factories. In order to better streamline the development pipeline, we decided to move the script off of the factories, and installed it on a separate test machine at UCSD. This machine happened to be the same sharing the UCSD test RabbitMQ instance but that is a temporary implementation detail. The reason why this still works is because the respective `schedd_status.xml` and `completed_data.json` files that publish the raw data on the production factories are also be accessible via HTTP from any external host.

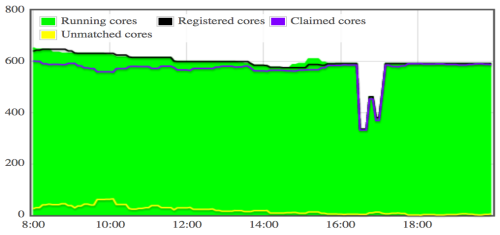
Besides moving the `factory_monitor.py` script out of the factories and onto its own host, we also updated the code to handle multiple factory sources, and the ability to export the data to multiple database destinations. The script now simply loops over a list of factories, and parses the `completed_data.json` and `schedd_status.xml` files for each. The list of factories is configurable. In order to support multiple destinations, we simply added a messenger module that understands how to push the monitoring data to a RabbitMQ host. This module is invoked along with the already existing InfluxDB messenger module from the pre-CHEP implementation, and currently also sends the monitoring metrics to the UCSD test RabbitMQ instance. Work is still being done to reuse the RabbitMQ messenger module and also feed the Nebraska RabbitMQ instance, and ultimately to send the data to the OSG GRACC system.

3 Results

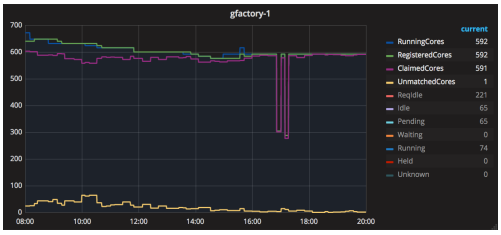
The fundamental test to ensure our new monitoring system is a viable candidate to replace the built-in GlideinWMS monitoring is to try and replicate the most used plots. Figure 1 and Figure 2 show before and after snapshots comparing the built-in monitoring with new plots created in Grafana. The data source of these plots is the `schedd_status.xml` file. Similarly, Figure 3 and Figure 4 show the before and after for plots derived from the `completed_data.json` source. Aside from slight differences in color choices more appropriate to the default dark theme of Grafana, for the purposes of daily factory operations, these plots are one to one replicas.

An important fundamental difference in the way the new plots are generated vs the old monitoring is that the rules used to generate them are completely decoupled from any code. The user interface in Grafana lets you add and modify plots arbitrarily on the fly. Conversely the built-in GlideinWMS plot definitions are hard-coded in the web monitor JavaScript. Arbitrary changes are not possible without pull requests to the GlideinWMS developers, and then waiting for the patches to be released in the next version of GlideinWMS rpms.

Another result worth noting is that we have successfully replicated the data graphs using both the InfluxDB and Elasticsearch data store backends. Grafana has built in support for both, so it is trivial to point it to either database type. The query languages are different, but Grafana allows you to write the queries required based on the configured backend, and generate visualizations with either. Our test Grafana instance is currently configured to interface with both backends, so we have separate dashboards containing the respective plots generated from Influx and Elasticsearch. The important implication here is that we are not stuck using a particular implementation of the data store, our new monitoring system is flexible to handle a variety of options.

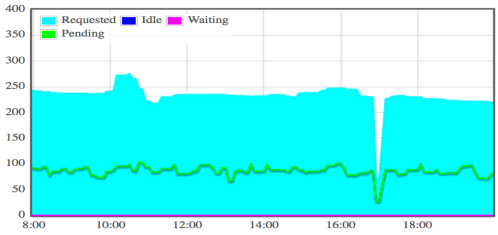


(a) built-in

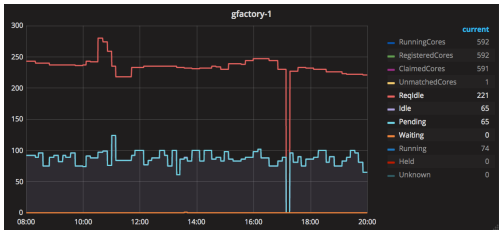


(b) Grafana

Figure 1. Comparison of running pilots at a particular compute element from built-in GlideinWMS vs new Grafana monitoring

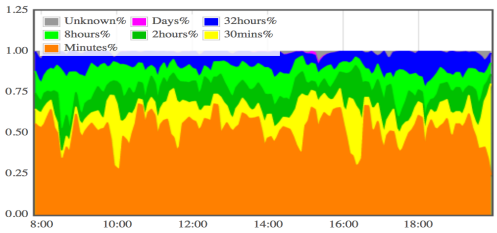


(a) built-in

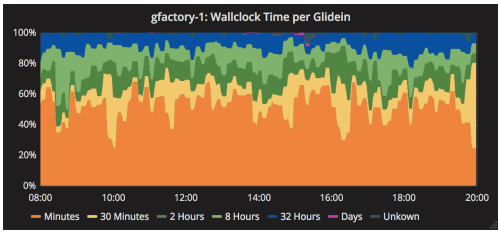


(b) Grafana

Figure 2. Comparison of idle pilots at a particular compute element from built-in GlideinWMS vs new Grafana monitoring

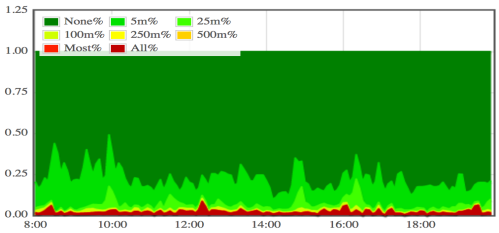


(a) built-in

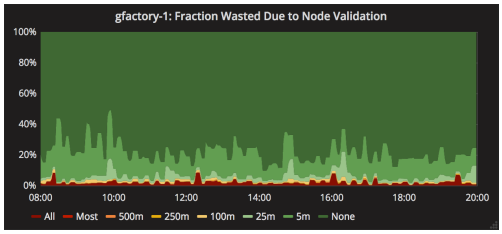


(b) Grafana

Figure 3. Comparison of pilot walltime at a particular factory from built-in GlideinWMS vs new Grafana monitoring



(a) built-in



(b) Grafana

Figure 4. Comparison of pilot validation failures at a particular factory from built-in GlideinWMS vs new Grafana monitoring

4 Conclusions

The new factory monitoring builds on and improves the original GlideinWMS factory monitoring in a few fundamental ways: it decouples the publishing of raw data from the monitor database backends, pulls the implementation out of the factory code and onto a separate host allowing aggregation of data from multiple factories, and modularizes the code to take

advantage of a variety of quickly-becoming standard monitoring tools used in industry and throughout the HEP computing community.

References

[1] I. Sfiligoi, J.M. Dost, M. Zvada, I. Butenas, B. Holzman, F. Wuerthwein, P. Kreuzer, S.W. Teige, R. Quick, J.M. Hernndez et al., Journal of Physics: Conference Series **396**, 032103 (2012)

[2] R. Pordes, D. Petravick, B. Kramer, D. Olson, M. Livny, A. Roy, P. Avery, K. Blackburn, T. Wenaus, F. Würthwein et al., Journal of Physics: Conference Series **78**, 012057 (2007)

[3] P. Mhashilkar, M. Mambelli, igor sfiligoi, holzman, klarson1, jdost321, ddbox, M. Mascheroni, J. Weigand, L. Lobato et al., *glideinwms/glideinwms: v3.4* (2018), https://doi.org/10.5281/zenodo.1309679