# Development and Implementation of a General Monitoring System for the Green Bank Telescope

Laura Leyzorek, West Virginia University
Advisor: Ramon Creager, Green Bank Observatory

July 10th, 2020

## 1 Introduction

The Green Bank Observatory (GBO) is a cutting-edge facility, made so by the state-of-the-art equipment it possesses, especially the Robert C. Byrd Green Bank Telescope (GBT). The GBT has a wide range of capability due to the variety of receivers it accommodates, its range of motion, and its sheer size, among other things. The Observatory continues to broaden its capabilities, adding new receivers such as the Ultra-Wideband Receiver, new backends, etc., allowing it to remain relevant to expanding the frontiers of radio astronomy. However, with the addition of new equipment comes increased opportunity for failures. In the interest of maintaining the GBT, both its hardware and software, engineers and technicians must constantly monitor its systems to ensure they are functioning properly. With only the existing systems in place, the necessary level of monitoring is often difficult to achieve. This project aims to create a more convenient, extensive, and informative system for monitoring the GBO equipment, specifically the GBT.

This system implements the time-series database Prometheus for receiving and recording data from the current GBT data publishing software, and Grafana as the user interface to display the data in a useful format. In order to retrieve the data from the GBT software system, and format it so that Prometheus can record it, a custom exporter was developed in the Python programming language. The exporter makes the data available to Prometheus at an HTTP endpoint on the server the exporter is launched from. Prometheus is configured to scrape that port periodically. Finally, Prometheus is set as the data source for Grafana, and the data can then be called for display directly in Grafana, where it can be easily viewed and manipulated.

## 2 Background

### 2.1 Prometheus and Grafana

Prometheus is an opensource toolkit for systems monitoring. It saves data in the form of metrics and key/value pairs as time series, meaning each value is given a timestamp. Prometheus uses its own query language, PromQL, allowing analysis and selection of the data it collects. It collects data from exporters, of which there are a wide variety to allow for data collection from almost any source. Prometheus also offers client libraries for a variety of programming languages so that custom exporters can be written if no existing exporters are sufficient.[1] Grafana is a versatile

platform for displaying timeseries data and is conveniently accessible from the web. It also offers built in support for Prometheus, making Prometheus and Grafana a widely used combination for systems monitoring applications.[2]

The large network of users and amount of available documentation for a Prometheus/Grafana monitoring system were key considerations in selecting these programs for this monitoring system. Additionally, a system similar to the one detailed in this report was created for use with the Breakthrough Listen (BTL) project when it established itself at the Green Bank Observatory. BTL's system served as a model and inspiration for this GBT monitoring system.

### 2.2 Existing GBT Systems

Understanding the software architecture responsible for collecting and publishing the relevant data was an integral requirement of this project. Each device or set of related devices associated with the GBT has a corresponding manager. Each manager has child managers under it, further specifying the device it represents, and each child manager oversees a set of samplers and parameters. All samplers and parameters are unique in their respective namespaces. That is, each Parameter is uniquely identifiable in the system's Parameter namespace, and each Sampler in the system's Sampler namespace. Each can therefore be identified by a unique key specifying which of the namespaces it is in, which manager it belongs to, and its name.[3] Parameters can contain a wide variety of information, including state of device, units, and other information about the device, while samplers refer to sets of raw numerical data which continually updates at regular intervals.[4]

Previous options in place for monitoring the Green Bank Telescope include gbtlogview and sampler2og. Logview offers a graphical user interface allowing users an option to select samplers to see data from and a time interval over which to plot it. However, it is inconveniently slow to plot relatively long time intervals or large data sets. It is complicated to access and require a familiarity with Linux operating systems and the GBO network. Sampler2log collects data from various samplers and stores it as FITS files in a specified directory, but offers no way to visually display it. However, both of these applications operate similarly to the final version of this monitoring system in how they gather GBT data. They provided valuable examples of how such a system might collect and manage data within the telescope's monitoring and control system.

### 2.3 Data Publisher

The managers equipped to do so publish their own data using a library, Data Publisher, created to allow them to use ZMQ as a transport and Protocol Buffers as the wire serialization protocol. Data Publisher offers several methods for retrieving the data from the managers. For use, it is imported into the virtual environment as data_pub. To retrieve data from the managers using Data Publisher, a key is needed. The key is constructed in the following manner: "MajorDevice.MinorDevice:S|P:key", where S or P refers to whether the particular data set is a sampler or a parameter. If there is a child manager, the parent manager is the major device, and the child manager is the minor device. If there are no child managers, both the major and minor device names are the same; the name of the parent manager.

One method for retrieving data through data_pub is to use the function get_data_snapshot(<key>). This will provide a snapshot of the current data from that key, as one would expect. However, this method also returns lots of metadata that may be considered irrelevant for the purposes of this project. In order to get only the sampler values themselves, another function, get_parameter_value() can be used in conjunction with get_data_snapshot. This will return only the numerical values of the sampler or parameter with a limited amount of metadata. There is also another method to get the desired data; by subscribing to a particular key. Subscribing uses the Portal class provided by data_pub to open a new thread accessing the desired manager, so that the most recent data sample will always be retrieved, via a custom callback function, which determines what is done with new data.

### 2.4 Prometheus-Client

Prometheus offers modules for several programming languages for writing custom exporters. The python module, Prometheus-Client, allows the exporter to start up an instance on a particular port on the local server, configures data into types Prometheus understands (gauges, counters, summaries, and histograms), and creates metrics from the data. Each metric can refer to multiple values, which can be differentiated by labels. Summaries and histograms refer to multiple values by nature, while counters and gauges can refer to a set of values or a single value.

## 3 The Exporter – deLIVER

A capable exporter was crucial to the success of this system, and its development constituted the majority of the efforts of the project. Its main function is to gather whatever data is desired for monitoring and make it available to Prometheus. Many considerations were made about how to best accomplish this goal, and to determine any additional abilities the exporter should have. Several iterations preceded the current version, each improving its functionality and adding to its usefulness.

The exporter, deLIVER, was developed in a virtual environment in the language python. The modules installed for its functionality included data_pub, a library of functions and classes used to access data from the GBT managers, prometheus-client, the python module for writing custom Prometheus exporters.

### 3.1 First Version

An exporter must create metrics, create an HTTP endpoint on a particular port, and set values to the metrics. The metrics must be in such a format that Prometheus is able to collect their values. Where the exporter gets the data and how it parses them is determined by custom functions in the exporter code.

To access the GBT data, a Portal to the managers was established using Data Publisher. The functions get_data_snapshot() and get_parameter_value() were selected as the means of

retrieving the data for simplicity, since that method didn't require writing a separate callback function for subscribing. These functions were put in the main loop, so that with every iteration they were called again and would retrieve whatever data was available. In order that the exporter wouldn't run continuously and use excessive processing power, it was set to sleep for one second after every iteration of the loop. The key ServoMonitor.ServoMonitor:S:elServo1Hz was selected as the first key to export, since the servo motor information is one of the most frequently monitored data sets. This key refers to a set of tachometer and motor current values for each of the servo motors responsible for moving the GBT in elevation, labeled "Tach1-8" and "Torq1-8." Each name under that key was manually typed into the exporter code to set the values to the metric. A single metric was generated for the key, and the multiple Tach and Torq values were differentiated by labels.

Generating the metric names from the data structure returned was arguably the most challenging aspect of the initial program design. The combination of functions dp.get_parameter_value(dp.get_data_snapshot()) returned the data in a recursive format as nested dictionaries. Each metric name was built from the result of the former functions, which meant using some of the dictionary keys rather than only their values. To retrieve the actual values, and the relevant dictionary keys to create the metric names, the initial exporter converted successive dictionaries into lists and used indexing in conjunction with dictionary keys to access the necessary information. However, this method was inefficient, and relied on every set of data having the same format.

To streamline the process, a python module called flatten-dict was imported. This module allowed the nested dictionaries to be flattened into a single dictionary, with all preceding keys turned into a single key for each value. In order for the flatten-dict module to be usable, data_pub had to be slightly modified and then reinstalled in the virtual environment. The function flatten(<nested dictionary>) was used to simplify the data structure, making each key a tuple containing all the nested keys that would have originally led to that value. This allowed the tuple to be converted into a list and whichever key was required to be selected through indexing in a single step.

### 3.2 Second Version

Once the first version was proved to function as expected, exporting the data for the given key on the specified port (initially port 8000 on the test host, devenv-hpc1), the next step was to make it generic. Initially, the goal was for the exporter gather all data available from every manager and export it all to a single port at once. Prometheus would scrape that port regularly, and any particular sampler could be selected for graphing in Grafana. The exporter would have code that would be generic to any sampler, and it would run through that code for every manager and every key at a regular interval.

However, after some discussion, it was decided that it would be more efficient and useful for the exporter to deal with one key, and only one. This would mean that a separate instance of the exporter would run on its own port for each sampler monitored. This would allow each sampler to be updated at a unique interval and would ensure that any unimportant data wouldn't be

exported unnecessarily. Due to the sheer volume of data available from Data Publisher, this would make the whole system much more streamlined and efficient.

Another possibility discussed was to have one instance of the exporter for each manager, meaning multiple keys and sampler sets would be up on a single port, but not all managers. This would lower the number of instances needed, while keeping the amount of data per port more manageable. However, this was decided against as it would not allow separate scraping intervals for each sampler and would also end up exporting unnecessary data. The extra time and effort required to launch a new instance of the exporter is negligible, and the number of available ports far exceeds the expected number to be used, meaning a single exporter instance per key would be sufficient.

The method by which the exporter receives its data from the managers was also altered in this iteration. It was decided that subscribing to the managers using the Portal class provided in data_pub was more desirable than using the functions get_data_snapshot and get_data_parameter. This would mean that the exporter would automatically update when there was new data available, rather than running at an arbitrary interval set using the sleep function. This aids in the exporter being generic, while also making it more efficient since it would only update when necessary. This required the addition of a callback function to determine what was to be done with each new value that the exporter received. Using python's queue class, a queue of length 1 was established to hold the most recent sampler value. This meant that only the most recent value would be held, and only until a more recent one was available.

Once the new version of the exporter was functional, it was broken into small functions and formatted to be runnable from the command line, allow for signal handling, and logging. It was configured to take a sampler key and a port number as arguments, given by flags in the command line. After testing it with multiple keys to ensure it was truly generic, two instances were launched from the command line in a tmux session on the machine devenv-hpc1, which could be detached to allow them to continue running even if the connection to the server was closed. They exported data for the azimuth and elevation servo motors on the GBT, to be used in further development of Prometheus and Grafana.

### 3.3 Final Version

The final modifications to deLIVER allowed it to respond to potential errors. Some anticipated errors were as follows: the exporter being called with an incorrect key, the given port already being in use, the manager or sampler going down, or the manager already being down when the exporter is called.

A new function was written to check the given key against the keys listed by Portal.list_keys() before starting up the port. This allowed the exporter to log an error message and close itself gracefully without getting any farther than necessary in the process if a key was incorrect, or if a manager was down at the time it was called.

Handling the case of a manager going down while deLIVER was subscribed to one of its keys posed a more significant challenge. Several solutions were considered, one being to have a separate exporter to monitor the health of all the managers. Another being to set the exporter to watch the result of the command mp.subscibe(), which returns either “True” or “False.” A separate exporter was decided against because several other applications already exist to monitor the states of the managers, and it was determined that developing a separate exporter for that purpose would be outside the scope of this project at the time. Simply monitoring the Boolean outputs from the subscribe function was determined to be insufficient and would involve too much added complexity to the exporter.
It was decided that the best solution would be to have the exporter keep track of the interval at which it receives data, allowing it to detect if it stopped receiving, which would indicate a problem with the manager it was subscribed to. This method takes advantage of the fact that samplers in the current GBT software system publish values at regular intervals unique to each sampler. The exporter would determine this interval for each instance, by averaging a sample of the most recent intervals. A separate watchdog function would be set to alert if no data was received after some multiple of the average interval.

This was attempted through writing functions to use datetime or timers. However, errors with the scope of variables arose, since certain portions of the code only ran under certain conditions, causing some variables to be referenced before assignment, among other errors. Due to the level of expertise of the developers and minimal experience with timing and related functions, this approach was abandoned. Instead, it was determined that developing a new python class to determine the average publishing period would be the most effective method. The class developed for this purpose was named MovingAverage. It calculated the current interval using the timestamps published with the data and appends this value to a list of set length. To determine the average, the mean of the list is taken.

MovingAverage was implemented in the exporter code by initializing an instance of it globally and adding a new interval to the list each time new data was published. The next step was to add a watchdog which would compare the average period to the current period, making sure that the current period was not significantly higher than the average. However, in the process of developing the watchdog, it was discovered that the function Queue.get(), used to retrieve data from the managers, blocked the code. This meant that no other code would run until new data was available. If a manager went down, this would result in the exporter waiting until the manager came back up without ever even running a watchdog. One possible solution considered was to run the watchdog in a separate thread, however this was disregarded to minimize complexity. Instead, the exporter was modified to use Queue.get_nowait(), since this does not block; rather it raises an exception when the queue is empty.

Since Queue.get_nowait() raises an exception, deLIVER had to be modified to include exception handling. Using the python try and except statements, the gathering data portion of the code was put under “try.” If no new data was available, the queue would be empty, and an exception would be raised. When this occurred, the exporter was set to sleep for half the average publishing period, so that no data samples would be missed, but the exporter would not continuously run unnecessarily. Using exception handling also provided a convenient place to put a watchdog function to allow deLIVER to create alerts when exceptions were raised. However, upon further

consideration, it was determined that the exporter itself did not need to have the ability to send alerts, and therefore no need for a watchdog function.

Once the average period was calculated, all that was needed was a way to see how many times the exporter ran without returning data. If this number was larger than two, since the exporter waited half the period every time no data was available, that would mean something had gone wrong. This value could be monitored by Prometheus and Grafana, and either of those could take care of alerting. This would keep the exporter as simple as possible while still achieving the goals. In order to monitor the number of times the exporter ran without receiving data, a counter class was implemented to reset every time the exporter ran without raising an exception, so that the counter never increases past two under normal conditions. This value was exported as a gauge, along with the average publishing period, so that this information could be available to Grafana.

### 3.4 Circus Manager

The final version of deLIVER.py was designed to be runnable from the command line, however, in order to ensure any instances of it would survive system restarts and would log messages appropriately, it was given a manager of its own, an open-source protram manager. Circus can run as a Linux Service, which means it can be started automatically when the machine starts. An instance of Circus was configured to run the exporter. Adding a new instance of deLIVER is a simple matter of adding to the configuration file of Circus, having Circus re-read it, and subsequently Circus takes care of ensuring its health.[5]

## 4 Prometheus

Prometheus runs as a service, listening on port 9090 by default, and was installed on the machine galileo. It stores the data it collects as a timeseries on the server it is installed on. For this project, all of Prometheus' data is stored in the directory /var/lib/Prometheus/data. The data is formatted into two-hour chunks, and one write-ahead-log so that Prometheus can still access the most recent data if the system were to go down for any reason.

Prometheus gathers data as metrics, by scraping desired targets. For the purposes of this project, the targets scraped by Prometheus are HTTP endpoints on which data is exported via deLIVER or other exporters. Prometheus defaults to the metrics path for scraping, but can also be set to other paths such as probe for use with other exporters. For example, Prometheus automatically makes metrics on itself, which are available for scraping at http://localhost:9090/metrics. This target is entered simply as 'localhost:9090,' so that a path is not designated here, but as a separate setting.

Prometheus' configurable settings are defined in the configuration file, prometheus.yml, found in /etc/prometheus on galileo. A variety of options are available for the way in which prometheus collects data. Targets, scrape intervals, paths, and alerting rules, among other things, are configurable.[6] For use in the General Telescope Monitoring System, Prometheus was configured as follows.

When a new instance of the exporter was launched, a new job was added. The job was named after the sampler it corresponded to. In general, each new sampler was added as a separate job. However, if the samplers were versions of the same information and shared a sampling rate, they were grouped into a single job to limit complexity. The port number which the exporter listened on was added to the job as a target with the format "devenv-hpc1:<port>," since all instances of deLIVER were run from devenv-hpc1. A scrape interval and a scrape timeout were set for each job corresponding to the publishing rate of the sampler. The scrape timeout was set to be the same as or shorter than the scrape interval, to ensure that data samples remained in the proper order. If a scrape interval was not set for job, it would be the default scrape interval, 15 seconds for this project, which was set at the beginning of the configuration file. The minimum scrape interval used was one second, as it was unclear if a more rapid sampling rate could be supported. This highlights one shortcoming of this system, and offers an opportunity for further research and development, since several key samplers on the GBT have sampling rates significantly higher than 1Hz.

Once modifications were been made to the configuration file, Prometheus had to be restarted, or a SIGHUP signal given to the PID to trigger a re-read. Any added samplers Prometheus monitors are listed at http://galileo:9090/targets, since Prometheus runs on galileo. All targets are listed there, showing their status (up, down, or unknown), URL, job, and how long ago each target was last scraped.

# 5 Grafana

## 5.1 Data Display

Grafana was also installed on galileo, and runs as a service, listening on port 3000. As the front end of the monitoring system, Grafana provides the user interface, and was configured to be as easily accessible and user-friendly as possible. Prometheus was set as the data source with an updating interval of one second. Since Grafana runs on a GBO machine, the webpage was only available on the internal GBO network, which is sufficiently secure that requiring users to log in would be unnecessary. Thus, the configuration was modified to allow anyone view-only access without logging in. To edit the displays, however, logging in as admin is still necessary.

Grafana's displays are organized into dashboards, each of which contain one or more panels, on which the data is graphed. In general, each Prometheus job is allotted one dashboard, meaning that the samplers from that job provide the data displayed on that dashboard. The data from the samplers is then graphed in as many panels as was deemed most logical for the type of data they represented.

The data for each panel is designated by a query, given in Prometheus' query language, promQL.[7] An example of such a query would be 'ServoMonitor_ServoMonitor_S_elServo1Hz{name="Torq1"}.' Since the metric name, ServoMonitor_ServoMonitor_S_elServo1Hz, corresponds to a whole set of values, specifying

the “name” selects only the data set with “name” equal to “Torq1.” Once a query is set for a panel, that panel will display only that data. In order to make the displays more customizable by the user, and subsequently more useful, a separate panel titled “Selected Data” was created within each dashboard. This panel implements variables, a powerful tool offered through Grafana, to allow users to select the data to be displayed thereon.

Each dashboard is created with its own set of variables, corresponding to the types of data it contains, as shown in figure 1. The variables are generated from the “name” label of the corresponding metric. For example, there are three weather samplers, weather1, weather2, and weather3. Each one returns the same set of values, including temperature, humidity, and wind velocity, each labeled accordingly. In order to allow the user to compare any of these values from any of the three samplers on a single graph, three variables were created, each one corresponding to one of the three weather samplers. Each variable can be set to any value or set of values from that sampler. The variables are displayed at the top of the dashboard as dropdown menus, where the desired data is easily selectable by the user.

The variables are set to a particular value by the user’s selection. For example, the user might select “Temperature” for the variable “Weather2.” This is only useful, however, if the graph changes to reflect the selected data. This was accomplished by using the variable within the query for the “Selected Data” panel. In the query for that panel, instead of setting the name label equal to a particular label value, it is set to the variable, so that it will change every time a new value is selected for that variable.

### 5.2 Alerting

Grafana also has the ability to send alerts through a variety of channels, including email, slack, discord, and many others. This is very desirable for this monitoring system, so that engineers or technicians can be notified of problems as quickly as possible. There were several possibilities as to what part of the system should be responsible for triggering alerts. The options included the exporter itself, Prometheus, or Grafana. The exporter was ruled out in order to keep it as simple as possible. Prometheus required the installation of a separate alert manager to send alerts. For simplicity’s sake, Grafana was chosen as the source for alerts in the system. To this end, it was configured to send emails from the account monctrl@gb.nrao.edu by modifying the SMTP section of Grafana’s configuration file. These email alerts can be sent to any desired email address set as a notification channel in Grafana.[8]

A dashboard was set up with a panel for the Times_No_Data metric, shown in figure 2, and an alert for that panel was configured to send an email alert whenever any of the counters are larger than 9. This would indicate that a manager has skipped 5 periods of publishing data. The alert includes a message explaining the significance of the alert and the name of the Prometheus job that has triggered it.

## 6 All Together

All components of the system work together to allow easy viewing of GBT data on Grafana, as visualized in figure 3. New samplers are added to the system through the following steps.

First, the key for the desired sampler must be determined so that the data can be accessed via Data Publisher.

Once the key is determined, a new instance of deLIVER must be launched by adding a watcher to the `circus.ini` file with the proper key and a new port number. Each new watcher must have a unique name. Current instances are named by the port they are running on since only one instance can run with each port number. Once a new watcher is added, circus must be restarted. To ensure the new instance of the exporter is running, one can `curl` http://devenv-hpc1:<port> to see the metrics being exported.

Once a new instance of the exporter is launched, that new port number must be added to Prometheus' configuration file as a new target. A new job with "server:port" as the target must be added. Any job-specific settings, such as scrape interval, must also be set. Once modifications have been made to the configuration file, Prometheus must be restarted, or a SIGHUP signal can be given to the PID to trigger a re-read.

To ensure that it has been properly configured, http://galileo:9090/targets can be checked. The new instance of the exporter should be listed as a target with status "up."

To plot the data from the newly exported key in Grafana, a new dashboard or a new panel in an existing dashboard must be added and a query referencing the new metric entered where directed. The query is formatted using promQL, Prometheus' querying language, which offers a wide variety of options for formatting data, including arithmetic operations, filtering, and time offsetting, among other things.

## 7 Future Work

In order for this monitoring system to be of maximum benefit to the observatory, there are many opportunities for improvement and expansion. Continuation of the project will aim to include statistics on the performance of many GBO machines, including network, CPU, GPU, and disk usage, among other things. This would provide valuable information for diagnosing and preventing software errors. With more successful additions to this monitoring system, further expansion may be warranted.

One shortcoming of the custom exporter written for this project, deLIVER, is that it requires a manager to use Data Publisher in order to receive data from that manager. Not all managers use data publisher, meaning that deLIVER is sometimes not usable without patching the desired manager. A possible solution would be to design a new exporter, similar to deLIVER, but using another method to access the data from the managers, instead of data publisher. However, it may be more progressive to simply update all managers to use data publisher. Existing exporters, such as the node-exporter and nvidia_gpu_prometheus_exporter, could be used for monitoring statistics related to GBO machines. The wide variety of Prometheus exporters allows for the possibility of monitoring almost anything.

As more data sources are added to the monitoring system, the volume of data retained by Prometheus will continue to grow, potentially increasing past what is practical. Setting up a time limit on data retention may be a desirable next step in order to limit the disk space utilized by Prometheus. This is a common step taken with Prometheus monitoring systems; the Breakthrough Listen project, for example, limits their data retention to the last 90 days. Alternatively, if it is determined that long term data storage through Prometheus is desirable, external data storage options may be considered.

In order to improve the functionality of Grafana as an aid in prevention and rapid detection of problems, several upgrades to its notification abilities could be put into place. Grafana's alerts can be set up simply from the webpage without the installation of any separate alert manager. However, if the number and complexity of alerts were to increase significantly, installation of an alert manager might become more desirable. The alert manager for Grafana is also compatible with Prometheus. Currently, Grafana is only configured to send alerts through email. The email alerts contain a message describing what triggered the alerts and listing the relevant metric or metrics. These alert messages could be improved with the addition of an image processor to allow Grafana to send an image of the alerting graph with the message. More channels could be added in the future, such as Slack, Discord, or even telephone calls. Alerts for certain dashboards could be configured to be sent to different people depending on who is most interested in that particular device.

Another feature offered by Grafana is the ability to add annotations to displays. If implemented, this would allow notes to be added to clearly show events or useful information along with the graph so that data could be correlated with certain events. This could be investigated more in future development of this monitoring system.

# 8 Conclusion

This project has successfully demonstrated the viability of using prometheus and grafana to provide a capable and user-friendly option for monitoring various components of the Green Bank Telescope. The development of deLIVER exporter allowed for all relevant data to be accessed through this system, and demonstrated that any desired data could be collected, even if a new exporter might need to be written. Grafana, with Prometheus as a data source, allowed for all data to be displayed in clear and informative graphs easily accessible from any operating system. It also provided the ability to send timely alerts when abnormalities in the data were detected. With continued development and upgrades, this monitoring system will provide technicians, engineers, and software developers with valuable information about many systems around the Green Bank Observatory and serve as an invaluable tool in failure prevention or diagnostics.

# 9 References

[1] https://prometheus.io/docs/introduction/overview/

[2] https://grafana.com/docs/grafana/latest/features/datasources/prometheus/

[3] https://www.gb.nrao.edu/~rcreager/DataPublisher/tutorial.html

[4] https://www.gb.nrao.edu/~rcreager/Survey/index.html#introduction

[5] https://circus.readthedocs.io/en/latest/

[6] https://prometheus.io/docs/prometheus/latest/configuration/configuration/.

[7] https://prometheus.io/docs/prometheus/latest/querying/basics/.

[8] https://grafana.com/docs/grafana/latest/alerting/create-alerts/.


**Figure 1**. A screenshot of the Weather panel. Variables specific to the panel are displayed at the top, allowing users to select their desired data. The Time_Offset box is a variable allowing users to type any time interval desired. These variables determine the data displayed in the first two panels.

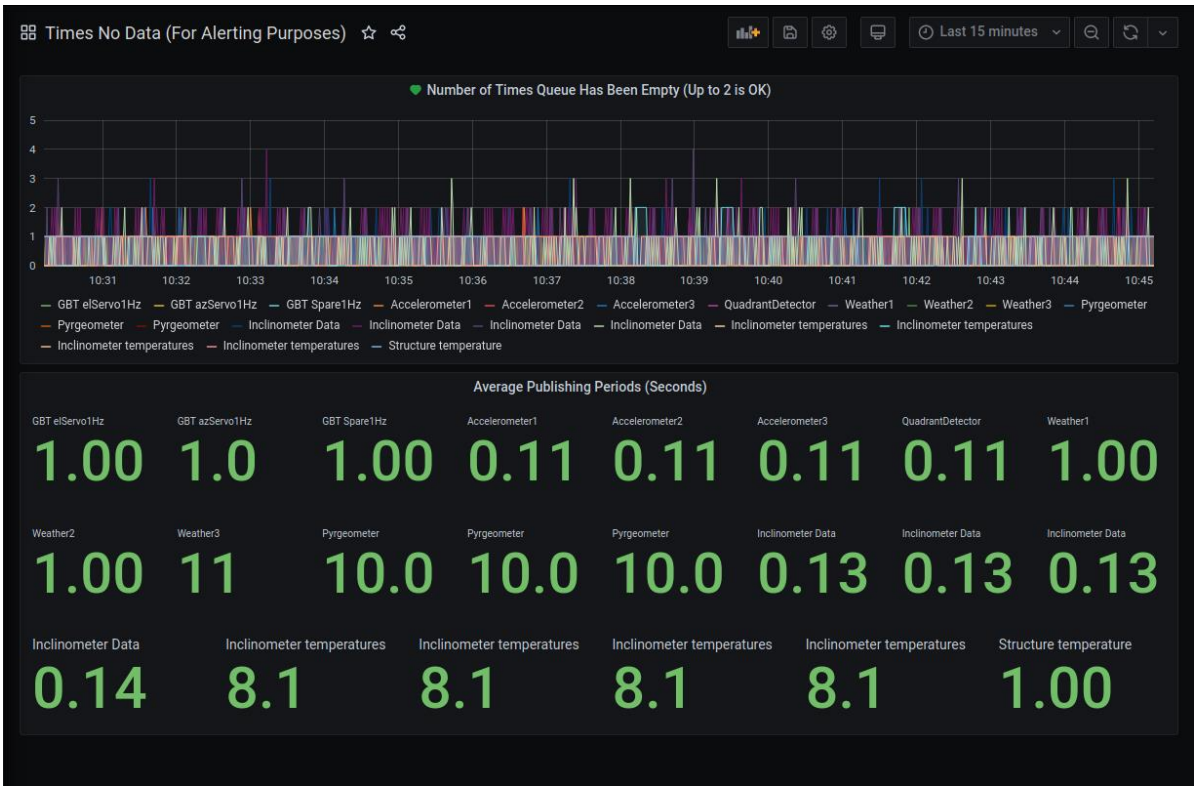


**Figure 2.** The Grafana panel dedicated to displaying the Times_No_Data metric created by deLIVER. If a sampler were perfectly regular, this value would never be greater than 2. Notice that occasionally it is as high as 3 or 4, indicating significant irregularity of those samplers. However, the alert is not triggered until it reaches 10, indicating 5 periods have been skipped.

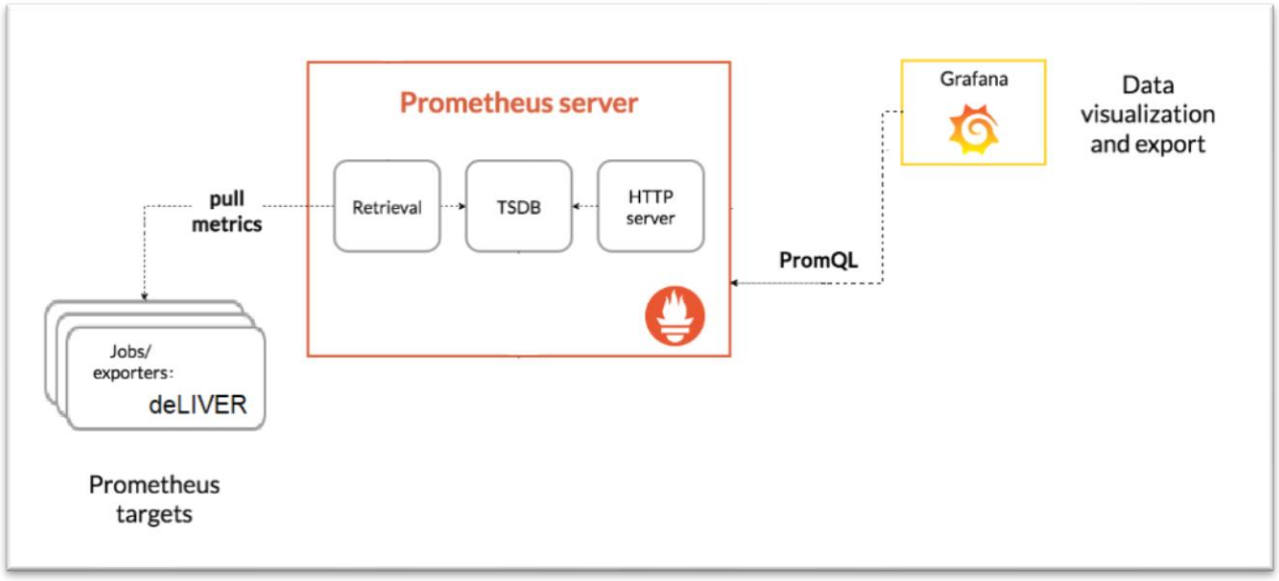


**Figure 3.** A simple diagram of the structure of this monitoring system demonstrating the flow of data.