
MonAMI by example

Paul Millar <p.millar@physics.gla.ac.uk>

Table of Contents

1. Introduction	2
2. Simple periodic monitoring	6
3. Selecting and merging available data	9
4. Caching and named samples	15
5. On-demand monitoring	18
6. Plotting data with Ganglia	22
7. Using Nagios to trigger alerts	27
8. Writing data into MySQL	31

1. Introduction

This tutorial aims to teach you how to configure MonAMI. It is split into different sections, each exploring a specific example configuration file. The example files start off simple, but they get more complex as you proceed. If you find you are having difficulty, try rereading the previous section: you might have missed something there.

The subject of this tutorial is MonAMI: a data-collection framework that can provide monitoring information for different monitoring systems. MonAMI consists of a daemon and a set of plugins. The plugins provide the useful functionality, such as collecting monitoring information about resources or sending that information to monitoring systems. This will help you monitor your computer resources, allowing you to provide a reliable service.

MonAMI works in collaboration with existing monitoring systems. It collects data describing your services and sends it to a monitoring system, or multiple systems. MonAMI itself does *not* draw pretty graphs, provide trend analysis, or alert you if a service is looking suspicious. This is deliberate: there are many monitoring packages that provide these features already. Instead, it concentrates on getting information out of whatever services you are running.

Since MonAMI is designed not to favour a particular monitoring system, this tutorial will begin by concentrating on the basics without requiring any particular monitoring system. Later on, the sections will concentrate on using MonAMI in more specific situations. These section may use specific applications or monitoring systems. Wherever possible, alternatives are given.

Typographical conventions

This tutorial uses symbols and different typeface to label the different material. Sections that contain contents of a file use a slightly smaller mono-spaced font and are contained within a box.

With most file contents included, there are some small circled numbers. When copying the text files, don't copy them! They are not part of the file's contents, but are to draw your attention to specific points of interest. These points are explained immediately below the text.

The following is an example shell script. The text also includes some circled numbers:

```
#!/bin/sh ❶  
#❷ A simple example script  
  
echo❸ "Hello, world."
```

- ❶ the shell to run; in this case, the Borne shell.
- ❷ comment lines start with a hash symbol (#) and are ignored.
- ❸ the echo command produces some output.

Other sections display typical output from running a program. They are also displayed in a mono-spaced font within a box. For example, here is the result of running the above script:

```
paul@donachain:~$ ./hello.sh  
Hello, world.  
paul@donachain:~$
```



Extra notes...

Throughout the tutorial there are several notes. These are separate short sections, marked with this quill pen icon. These notes contain additional information that give a greater breadth of understanding, but are extra: you can skip them if you want to.

Prerequisites

For this tutorial, you need:

- A computer with MonAMI installed. MonAMI itself does not need `root` access for any monitoring activities: it can run quite happily as a normal user. Perhaps the easiest way of getting started is to install one of the binary packages, available from the MonAMI webpage.
- The ability to edit the MonAMI configuration files. The packaged version of MonAMI uses files in the `/etc` directory and have permission settings that require `root` privileges to edit. If you are running a version of MonAMI you have compiled for yourself then the location of the configuration files may differ and you may be able to edit these files without `root` privileges.
- about 15 minutes of spare time per section. This is an estimate of how long going through an example will take. If you have more time, you are encouraged to try altering the example configuration and seeing what happens.
- Various sections have specific requirements:
 - Section 5 *KSysGuard* or *telnet*. *KSysGuard* is a standard part of the KDE desktop and is available for GNU/Linux and Macintosh computers. There are instructions for using *telnet* instead of *KSysGuard*, allowing people who don't have *KSysGuard* to appreciate on-demand monitoring.
 - Section 6 *Ganglia*. This section is based on using MonAMI with *Ganglia*.
 - Section 7 *Nagios*. This section shows how to configure MonAMI to generate alerts within *Nagios*.
 - Section 8 *MySQL*. This section shows how to store data within a *MySQL* database. You will need an account (username and password) that has **CREATE** and **INSERT** privileges for some existing database within *MySQL*.

Installing MonAMI

As it says above, this tutorial assumes you have installed MonAMI on some computer, so the tutorial does not provide detailed information on how to install MonAMI. However, for the sake of completeness, some comments are included.

You can download MonAMI from the SourceForge download area. There are links to there from the *MonAMI home page* [<http://monami.sourceforge.net/>]. You should find the latest version available as binary and source RPMS, along with a compressed tar file containing the source.

The binary RPMs are split into a core RPM, several plugin RPMs and a documentation RPM. The core RPM provides basic infrastructure and a collection of plugins with no external dependencies. The various plugin RPMs provide additional functionality, but also include some dependencies on other libraries. The documentation RPM contains PDF and HTML versions of the User Guide.

There is also a yum and apt repository for these RPMs. These are hosted at ScotGrid. To use the yum repository, copy the following text as `/etc/yum.repos.d/monami.repo`.

```
[monami]
name=MonAMI -- your friendly monitoring daemon
# Use either "308" or "44" below.
#baseurl=http://monami.scotgrid.ac.uk/scientific/308/$basearch/
baseurl=http://monami.scotgrid.ac.uk/scientific/44/$basearch/
enabled=1
```

Configuration files

With the prepackaged version of MonAMI, configuration is held in the `/etc/monami.d` directory. In the following examples, you should create a file `/etc/monami.d/example.conf`. Each example is self-contained, so you should overwrite this file for each example.

Each section is based around a theme and each theme is usually based around a specific example configuration file. This section is no exception. However, this section's configuration file is rather boring as it will have no effect: MonAMI will run identically either with or without this file. It will run until told to stop.

If you do create the example configuration file, save the content as `/etc/monami.d/example.conf`, but make sure you don't copy the circled numbers: they indicate points of interest.

```
## ❶
## MonAMI by Example, Section 1
##
## ❷
## This file does nothing.
```

Here are some points of interest:

- ❶ Comments can be added by starting a line with a hash symbol (#). These comment lines are ignored. If a line starts with any other character, then it is not a comment line and will be processed.
- ❷ Completely blank lines are also allowed. They, too, are completely ignored, so you can include them anywhere within a configuration file.

Starting and stopping MonAMI

MonAMI will normally detach itself from the shell and redirect its output somewhere (to `syslog` by default). This is desirable behaviour for a daemon; but, for the purposes of these tutorial exercises it is better if MonAMI runs both without detaching from the shell and providing more verbose messages.

To achieve this, run the MonAMI executable directly use the command `/usr/bin/monamid -fv`. You can stop MonAMI by typing **Ctrl+C**.

When starting MonAMI, you will see output like:

```
paul@donachain:~$ /usr/bin/monamid -fv
Loading configuration file /etc/monami.conf
    plugin apache loaded
    plugin amga loaded
There will be many similar lines.
    plugin tcp loaded
    plugin tomcat loaded
Starting up...
```

When MonAMI is shutting down, you will see the following:

```
Waiting for activity to stop...
Shutting down threads...
paul@donachain:~$
```

Usually, when told to shutdown, MonAMI isn't doing anything. If so, then MonAMI will quickly exit. If MonAMI is collecting data when told to shutdown, you may see a slight delay between the “waiting for activity to stop...” and “shutting down threads...” messages. This delay is expected, and will last only for as long as MonAMI needs to finish the current activity. If the data source is slow, it may take a few seconds.

If shutting down takes longer than a minute, MonAMI will assume something has gone wrong and a bug within MonAMI or one of the plugins has been found. If this happens, it will record some debugging information and try harder to stop. Naturally, you should never see this happen!

Getting more information

The information here aims to be self-complete. However, you may be left wondering about some specific aspect and want to know more. You can get more information from:

- The packaged versions of MonAMI include manual pages for how to run monamid and the MonAMI configuration file format: monamid(8) and monami.conf(5).
- The MonAMI User Guide contains a wealth of information about MonAMI, including documentation on all the plugins and how to configure MonAMI. It is available in PDF and HTML formats, both as an RPM package and from the *MonAMI home page* [<http://monami.sourceforge.net/>].
- If you have a question that isn't answered by any of these references, feel free to join the MonAMI users mailing list and ask it there.

2. Simple periodic monitoring

In this section we will tell MonAMI to record everything it knows about something periodically. This will introduce periodic monitoring, which is perhaps the most common monitoring activity.

Plugins and Targets

Before looking at the configuration, it would help to understand two concepts within MonAMI: plugins and targets.

Plugins are fundamental to MonAMI. They allow information to be gathered, stored, or otherwise processed. Most plugins come in one of two types: either monitoring or reporting.

Monitoring plugins are those that provide information. A monitoring plugin understands how to obtain information from a specific service, program or other source of information. All the peculiarities with obtaining that information are concealed within the plugin, so they provide a uniform method of obtaining information. The *filesystem* and *apache* plugins are examples of monitoring plugins. Both are available within the default package.

Reporting plugins will accept monitoring information. They will either store this information or send it to some monitoring system. The *snapshot* plugin is an example of a reporting plugin. It will store all information it is given as a file, overwriting any existing content.

There are yet other plugins that lead a more complex life, such as the *sample* and *dispatch* plugins. Don't worry about these just yet: they'll become clear further along. Some of the aspects of the *sample* plugin are mentioned here, but both plugins will be explored more in later sections.

In all, MonAMI comes with many useful plugins and the number grows with each release. They are all described within the MonAMI User Guide and brief synopses are included within the system manual (man page) entry on the configuration file format.

Targets are configured instances of a plugin. They exist only when MonAMI is run and are described by the MonAMI configuration files; for example, whilst the *mysql* plugin knows (in principle) how to monitor a MySQL server, it is a target (that uses the *mysql* plugin) that knows how to monitor a specific MySQL server instance running on a particular machine, using a particular username and password.

As with plugins, a *monitoring target* is a target that provides MonAMI with information. It is a configured monitoring plugin. Likewise, a *reporting target* is a target that accepted information and is based on a reporting plugin.

An easy way to illustrate the difference between plugins and targets is to consider monitoring several partitions. The *filesystem* plugin will monitor a partition. To monitor multiple partitions, one would configure multiple *filesystem* targets. Each of these targets will use the *filesystem* plugin. Although there may be several *filesystem* targets, there is only one *filesystem* plugin.

Configuration file

The example configuration file used in this section will tell MonAMI to measure the current status of the root filesystem every two seconds and store all the data in the file `/tmp/monami-filesystem`. Copy the text below and store it as the `/etc/monami.d/example.conf` file.

```
##
## MonAMI by Example, Section 2
##

# Monitor our root filesystem
[filesystem] ❶
  location = /      ❷
  name = root-fs    ❸
```

```
# Record latest f/s stats every two second
[sample] ④
  read = root-fs
  write⑤ = snapshot
  interval⑥ = 2

# The current filesystem statistics
[snapshot] ⑦
  ⑧
  filename = /tmp/monami-filesystem
```

The following points are worth noting:

- ① The configuration file is split into stanzas, each of which starts with a line containing the name of a plugin in square brackets. In this case, the *filesystem* plugin is mentioned. Each stanza creates a new target that uses the specified plugin.
- ② Stanzas can have multiple attribute lines describing how the target should behave. Each attribute line has a keyword, followed by an equals sign, followed by the value (white space is also allowed). Some plugins require certain attributes to be specified; other attributes are optional.
- ③ Each target has a name, which must be unique. The name attribute allows you to configure what a target's name should be.
- ④ Sample targets collect together data and send it somewhere. They are also somewhat special: you can specify any number without explicitly giving them unique names.
- ⑤ On receiving fresh data, the *sample* target will deliver it to the targets named in the *write* attribute.
- ⑥ If a *interval* attribute is configured, this will be triggered periodically. It states how often the *sample* target will request fresh data.
- ⑦ The *snapshot* plugin accepts data and writes it to disk. The *sample* sends data to the *snapshot* target, which then writes this data to the disk.
- ⑧ If there is no name attribute, the target will take its name from the plugin. Naturally, this only works if a single target (at most) is created for each plugin.

The sample's *interval* attribute states how often the sample section will request fresh data. This happens every two seconds in this example (written as *2s* or just *2*). The *interval* can also be specified in minutes (e.g., every five minutes is *5m*), in hours (e.g., every six hours *6h*), or in combinations of these (e.g., *1h 30m*).



Fast enough?

MonAMI's internal time-keeper works with a time granularity of one second: once-per-second is the maximum rate that MonAMI can gather data. In fact, there is no particular technical reason that for forcing this maximum rate, but there is also no compelling reason to increase the maximum sampling rate.

At its heart, MonAMI is asynchronous. It can also gather and store data very quickly. When MonAMI is configured for event-based monitoring, it can store data with low latency: far faster than once per second. If you think that once-per-second is too slow, perhaps you can recast your monitoring requirement as one based on event-based monitoring.

Running the example

Make sure you run MonAMI for at least two seconds. When starting up, MonAMI attempts to spread its work evenly to reduce the impact of monitoring. It does this by starting the timed monitoring with a random fraction of the interval time. It can take up to two seconds before MonAMI will monitor the root filesystem in the example above.

Once data is collected, you should see the file `/tmp/monami-filesystem`. Depending on your local filesystem (and which version of MonAMI you are using) you should see output similar to the following:

```
"root-fs.fragment size" "1024" (B) [every 2s]
```

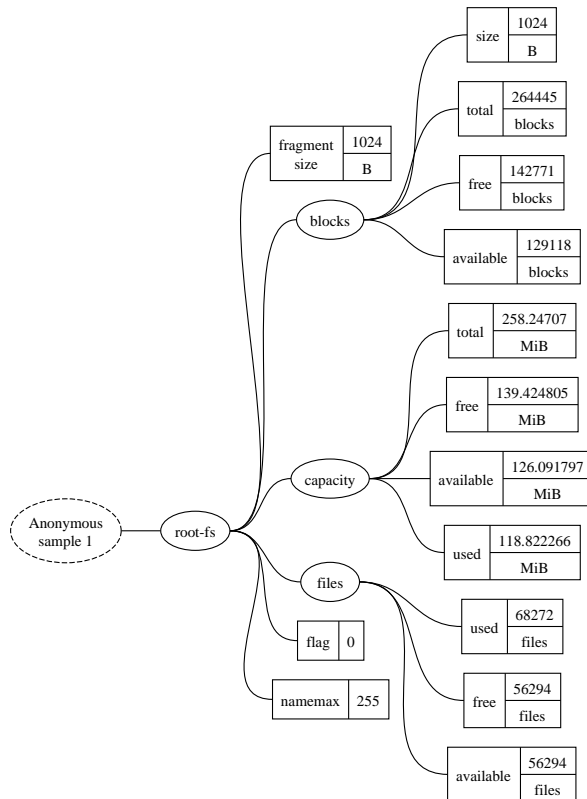
```

"root-fs.blocks.size"      "1024" (B) [every 2s]
"root-fs.blocks.total"    "264445" (blocks) [every 2s]
"root-fs.blocks.free"     "142771" (blocks) [every 2s]
"root-fs.blocks.available" "129118" (blocks) [every 2s]
"root-fs.capacity.total"  "258.24707" (MiB) [every 2s]
"root-fs.capacity.free"   "139.424805" (MiB) [every 2s]
"root-fs.capacity.available" "126.091797" (MiB) [every 2s]
"root-fs.capacity.used"   "118.822266" (MiB) [every 2s]
"root-fs.files.used"      "68272" (files) [every 2s]
"root-fs.files.free"      "56294" (files) [every 2s]
"root-fs.files.available" "56294" (files) [every 2s]
"root-fs.flag"            "0" ( ) [every 2s]
"root-fs.namemax"         "255" ( ) [every 2s]
    
```

Data and datatrees

Monitoring targets will often provide lots of information, often more than you actually want. To keep this information manageable, it is held in a tree structure, just like a filesystem: files correspond to a specific metric and directories (or folders) containing other directories and metrics.

The datatrees can be drawn as graphs. The example datatree, shown in the above snapshot output, is shown in Figure 1. The ellipses represent branches and the rectangle represents metrics.



To refer to a specific metric, you specify a path within the datatree, ignoring the very first element (shown as a dashed ellipse) as it is redundant. This is similar to how a file has an absolute path within a filesystem. Instead of using a slash (/ or \) to separate elements of a metric's path, a dot is used. So the metric in the lower right corner of the diagram is `root-fs.files.available`.

3. Selecting and merging available data

In this section, we look at how to select a subset of available information and merging different data-trees together. This allows us to be selective in what information to present and also to monitor different aspects of the system concurrently.

Configuration file

The following configuration will measure the available free space ¹ on the / (root) and /home filesystem and store the latest values in a file /tmp/monami-filesystem.

Copy the configuration below as the /etc/monami.d/example.conf file, replacing any existing file. Remember, don't try to copy the circled numbers!

```
##
## MonAMI by Example, Section 3
##

# Our root filesystem
[filesystem]
  name = root-fs
  location = /

# Our /home filesystem
[filesystem] ❶
  name = home-fs
  location = /home

# Record latest f/s stats every two seconds
[sample]
  read = ❷ root-fs❸.capacity.available, home-fs.capacity.available
  write = snapshot
  interval = 2

# The current filesystem statistics
[snapshot]
  filename = /tmp/monami-filesystem
```

A few things to note:

- ❶ The configuration creates two monitoring targets, called `root-fs` and `home-fs`. Both use the `filesystem` plugin.
- ❷ The `read` attribute is a comma-separated list of metrics or branches.
- ❸ The first element of the metric's path is the target name.

Running the example

As with the previous example, you should make sure MonAMI is running for at least two seconds to guarantee that the file /tmp/monami-filesystem has been created or updated. Depending on your filesystems this file should contain something like the following:

```
"root-fs.capacity.available"    "126.091797" (MiB) [every 2s]
"home-fs.capacity.available"    "11178.921875" (MiB) [every 2s]
```

¹ The term “available space” refers to the storage available to a non-root user whereas “free space” is the storage available to the root user. In general, the free space will be greater than the available space. More information is available in the MonAMI User Guide.

Combining different datatrees

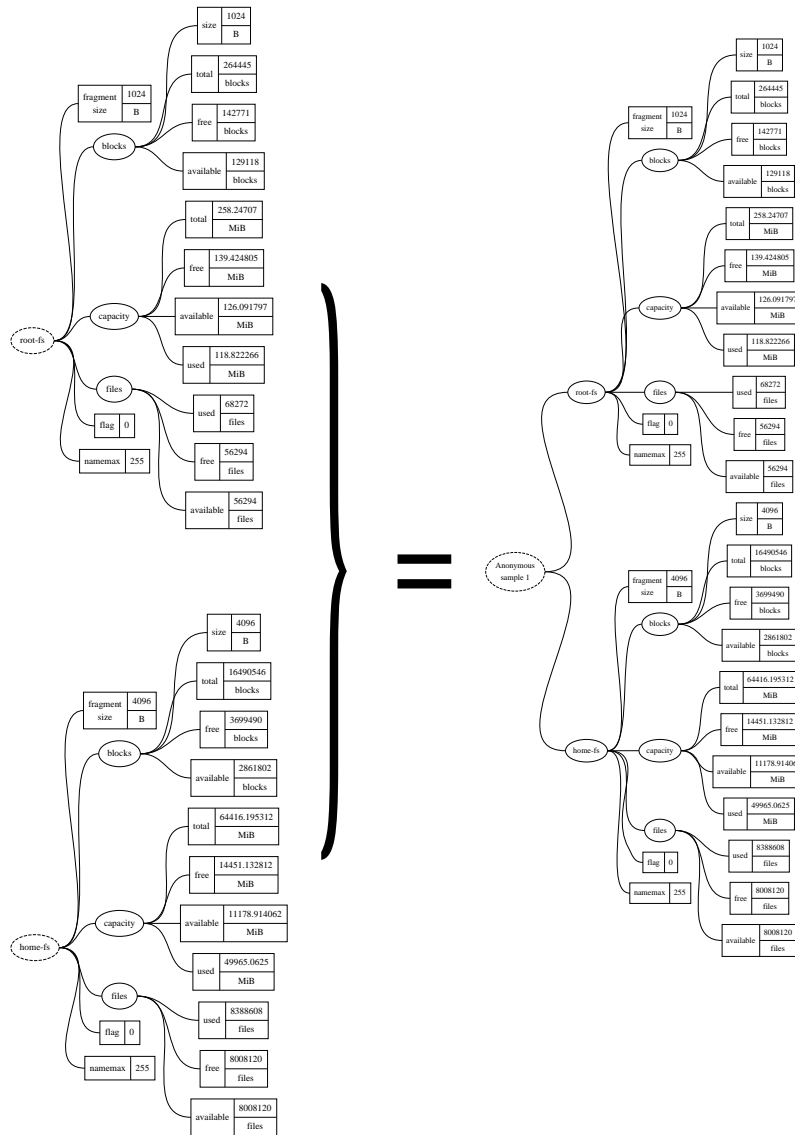
sample sections can combine different datatrees together by specifying them as a comma-separated list of sources. The simplest is to include all metrics from two sources, by simply specifying the two target names:

```
read = root-fs, home-fs
```

This will combine all data from the `root-fs` and `home-fs` targets, resulting in `/tmp/monami-filesystem` output like:

```
"root-fs.fragment size" "1024" (B) [every 2s]
"root-fs.blocks.size"   "1024" (B) [every 2s]
"root-fs.blocks.total"  "264445" (blocks) [every 2s]
"root-fs.blocks.free"   "142771" (blocks) [every 2s]
"root-fs.blocks.available" "129118" (blocks) [every 2s]
"root-fs.capacity.total" "258.24707" (MiB) [every 2s]
"root-fs.capacity.free" "139.424805" (MiB) [every 2s]
"root-fs.capacity.available" "126.091797" (MiB) [every 2s]
"root-fs.capacity.used" "118.822266" (MiB) [every 2s]
"root-fs.files.used"    "68272" (files) [every 2s]
"root-fs.files.free"    "56294" (files) [every 2s]
"root-fs.files.available" "56294" (files) [every 2s]
"root-fs.flag"         "0" () [every 2s]
"root-fs.namemax"      "255" () [every 2s]
"home-fs.fragment size" "4096" (B) [every 2s]
"home-fs.blocks.size"   "4096" (B) [every 2s]
"home-fs.blocks.total"  "16490546" (blocks) [every 2s]
"home-fs.blocks.free"   "3699490" (blocks) [every 2s]
"home-fs.blocks.available" "2861802" (blocks) [every 2s]
"home-fs.capacity.total" "64416.195312" (MiB) [every 2s]
"home-fs.capacity.free" "14451.132812" (MiB) [every 2s]
"home-fs.capacity.available" "11178.914062" (MiB) [every 2s]
"home-fs.capacity.used" "49965.0625" (MiB) [every 2s]
"home-fs.files.used"    "8388608" (files) [every 2s]
"home-fs.files.free"    "8008120" (files) [every 2s]
"home-fs.files.available" "8008120" (files) [every 2s]
"home-fs.flag"         "0" () [every 2s]
"home-fs.namemax"      "255" () [every 2s]
```

The process of combining the two datatrees is shown graphically in Figure 2.



Combining datatrees allows you to combine monitoring results from different targets. In the configuration at the beginning of this section, we merge two datatrees (each datatree has only one metric), but in general we can combine any number of datatrees, collecting data from any number of targets.

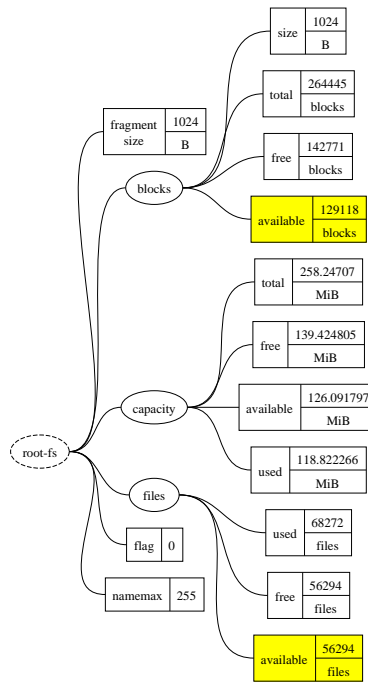
Selecting subsets of a datatree

In the above example, we select just one metric from within each target's datatree. From the `root-fs` and `home-fs` targets, we select only each target's `capacity.available` metric. This can be extended to select multiple metrics from each datatree.

Selecting metrics

To select multiple metrics, simply list the metrics you want as a comma-separated list of datatree paths. In the following example, two metrics have been selected.

```
read = root-fs.blocks.available, root-fs.files.available
```

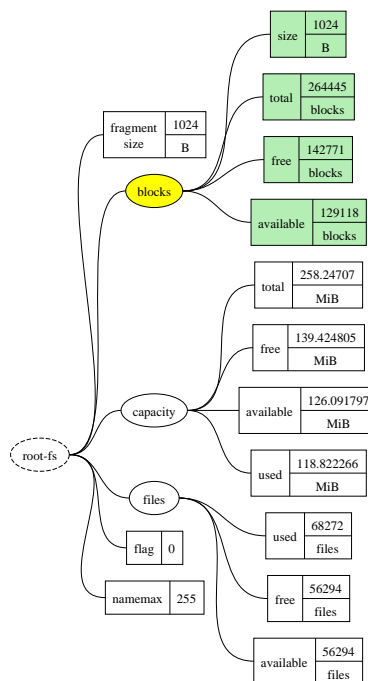


The resulting datatree will include both the `blocks.available` and `files.available` metrics.

Selecting branches

We can also specify the path of a branch to include all metrics within that branch. The following shows an example selecting the `blocks` branch from the `root-fs` target.

```
read = root-fs.blocks
```

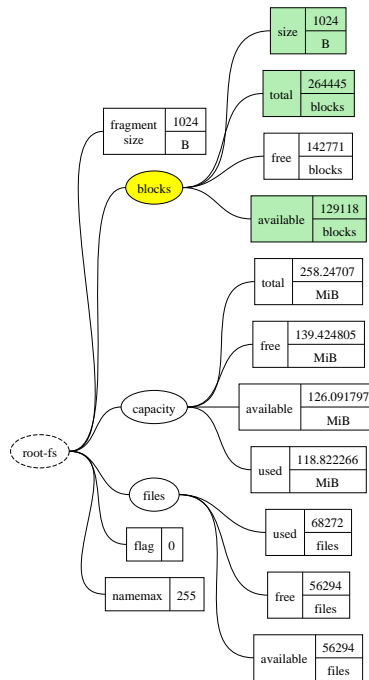


The new datatree will include the metrics `root-fs.blocks.size`, `root-fs.blocks.total`, `root-fs.blocks.free` and `root-fs.blocks.available`.

Vetoing metrics and branches

Sometimes it is easier to say what data you don't want to include. Specific metrics or branches can be excluded by listing them prefixed with an exclamation mark. The following demonstrates how to select all of the `blocks` metrics from the `root-fs` target except for the `blocks.size` metric:

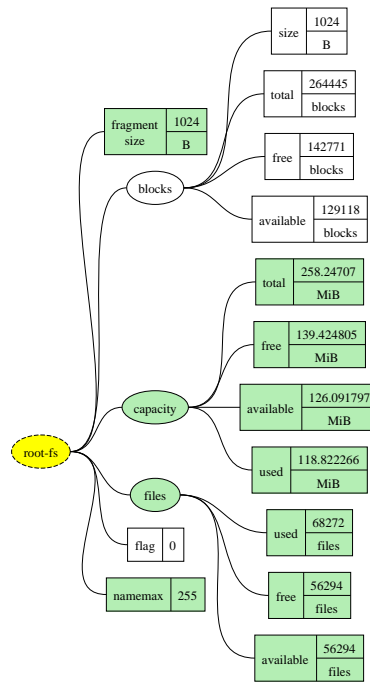
```
read = root-fs.blocks, !root-fs.blocks.free
```



This is easier than listing all the metrics individually: `root-fs.blocks.size`, `root-fs.blocks.total` and `root-fs.blocks.available`.

The final example shows selecting all metrics except for a metric and those beneath a branch.

```
read = root-fs, !root-fs.blocks, !root-fs.flag
```



By selecting metrics and branches, and by vetoing selected metrics or branches of metrics, arbitrary selection of metrics is made easy.

4. Caching and named samples

In this example, we show *caching* and *named samples*. Caching allows you to make sure you never overload a service from monitoring. Named samples allows logical grouping of related monitoring from different targets.

Configuration file

As before, copy the configuration file below as the file `/etc/monamid.d/example.conf`, overwriting any existing file.

```
##
## MonAMI by Example, Section 4
##

# Our root filesystem
[filesystem]
name = root-fs
location = /
cache = 2 ❶

# Our /home filesystem
[filesystem]
name = home-fs
location = /home
cache = 2s

# Bring together information about the two partitions
[sample]
name = partitions ❷
read = root-fs, home-fs
cache = 10

# Update our snapshot every ten seconds
[sample]
read = partitions ❸
write = snapshot
interval = 10

# Once a minute, send data to a log file.
[sample]
read = partitions.root-fs.capacity.available, \ ❹
      partitions.home-fs.capacity.available
write = filelog
interval = 1m ❺

# A file containing current filesystem information
[snapshot]
filename = /tmp/monami-fs-current

# A permanent log of a few important metrics
[filelog] ❻
filename = /tmp/monami-fs-log
```

Some points of interest:

- ❶ The `cache` attribute specifies a guaranteed minimum delay between successive requests for information. Here, there will always be at least two seconds between consecutive requests.

The value is a time-period: one or more words that specify how long the period should be. This is the same format as the `sample interval` attribute, so “5m” is five minutes and “1h 30m” is an hour and a half.

- ❷ Like all targets, this name must be unique.

- ③ This sample reads all available metrics from the `partitions` target. To gather this information, the `partitions` target will read from the two filesystem targets.
- ④ Sometimes attribute lines can get quite long. To make them easier to read and edit, long lines can be broken down into multiple shorter lines provided the last character is a backslash (`\`).
- ⑤ This interval is deliberately short to allow quick gathering of information. For normal use a much longer interval would be more appropriate.
- ⑥ The `filelog` plugin creates a file, if it does not already exist, and appends a new line for each datatree it receives. It is a simple method of archiving monitoring information.

Running MonAMI

With this example, you should leave MonAMI running for a few minutes. Whilst it is running, you can check that data is being appended to the log file (`/tmp/monami-fs-log`) correctly using, for example, the `cat` program.

Depending on which version of MonAMI you are using and the current state of your partitions, the file `/tmp/monami-fs-current` should look like:

```
"partitions.root-fs.fragment size"      "1024" (B) [every 10s]
"partitions.root-fs.blocks.size"        "1024" (B) [every 10s]
"partitions.root-fs.blocks.total"       "264445" (blocks) [every 10s]
"partitions.root-fs.blocks.free"        "142771" (blocks) [every 10s]
"partitions.root-fs.blocks.available"    "129118" (blocks) [every 10s]
"partitions.root-fs.capacity.total"     "258.24707" (MiB) [every 10s]
"partitions.root-fs.capacity.free"      "139.424805" (MiB) [every 10s]
"partitions.root-fs.capacity.available"  "126.091797" (MiB) [every 10s]
"partitions.root-fs.capacity.used"      "118.822266" (MiB) [every 10s]
"partitions.root-fs.files.used"         "68272" (files) [every 10s]
"partitions.root-fs.files.free"         "56294" (files) [every 10s]
"partitions.root-fs.files.available"     "56294" (files) [every 10s]
"partitions.root-fs.flag"               "0" () [every 10s]
"partitions.root-fs.namemax"            "255" () [every 10s]
"partitions.home-fs.fragment size"      "4096" (B) [every 10s]
"partitions.home-fs.blocks.size"        "4096" (B) [every 10s]
"partitions.home-fs.blocks.total"       "16490546" (blocks) [every 10s]
"partitions.home-fs.blocks.free"        "3699442" (blocks) [every 10s]
"partitions.home-fs.blocks.available"    "2861754" (blocks) [every 10s]
"partitions.home-fs.capacity.total"     "64416.195312" (MiB) [every 10s]
"partitions.home-fs.capacity.free"      "14450.945312" (MiB) [every 10s]
"partitions.home-fs.capacity.available"  "11178.726562" (MiB) [every 10s]
"partitions.home-fs.capacity.used"      "49965.25" (MiB) [every 10s]
"partitions.home-fs.files.used"         "8388608" (files) [every 10s]
"partitions.home-fs.files.free"         "8008117" (files) [every 10s]
"partitions.home-fs.files.available"     "8008117" (files) [every 10s]
"partitions.home-fs.flag"               "0" () [every 10s]
"partitions.home-fs.namemax"            "255" () [every 10s]
```

The file `/tmp/monami-fs-log` should look like:

```
#      time      partitions.root-fs.capacity.available  partitions.
home-fs.capacity.available
2007-10-03 11:12:59      126.091797      11178.707031
2007-10-03 11:13:59      126.091797      11178.703125
2007-10-03 11:14:59      126.091797      11178.703125
2007-10-03 11:15:59      126.091797      11178.710938
```

Named sample targets

A *named sample* target is simply a *sample* target that has a name attribute specified. In contrast, a sample without any specified name attribute is an *anonymous sample*. All the samples in previous sections are anonymous.

The main use for named samples is to allow grouping of monitoring data. Suppose you wanted to monitor multiple attributes about a service; for example, count active TCP connections, watch the application's use of the database, and count number of daemons running. You may, for ease of handling, want to build a datatree containing the combined set of metrics. A named sample allows you to do this.

Another aspect of named targets is that it allows other targets (such as anonymous samples) to request monitoring data from the named sample. Named samples can be used, in effect, as simple monitoring target (such as `root-fs` target above).



What's in a name?

In fact, anonymous sample sections *do* have a name: their name is assigned automatically when MonAMI starts. However, you should never use this name or need to know it. If you find you need to collect data from an anonymous sample, simply give the target a name.

Note that, although not illustrated in the above example, named samples will honour the `interval` attribute. This allows them to provide periodic monitoring information (in common with anonymous samples) whilst simultaneously allowing other targets to request information at other times.

Caching

Monitoring will always incur some cost (computational, memory and sometimes storage and network bandwidth usage). Sometimes this cost is sufficiently high that we might want to rate-limit any queries so, for example, a service is never monitored more than once every minute.

Within MonAMI, this is achieved with the `cache` attribute. You can configure any target to cache gathered metrics for a period. In the above example, metrics from the `partitions` named sample are cached for ten seconds. If one of the anonymous samples had the `interval` attribute set to less than 10 seconds, they would not trigger any gathering of fresh data. Instead, they would receive the previous (cached) result until the ten-second cache had expired.



Default caching policy

By default MonAMI will cache all results for one second. Since MonAMI monitoring frequency (the `interval` attribute) has a granularity of one second, this default cache will not be noticed when a target obtains data. However, if multiple targets request data from the same target at almost the same time (to within a second), the default cache ensures all the requests receive data from a single datatree.

Note that the `cache` attribute works for *sample* targets, as demonstrated in the above example. Caching targets with different cache-intervals allows a conservative level of caching for the bulk of the monitoring activity whilst retaining the possibility of adding more frequent monitoring.

Some monitoring plugins will report a different set of metrics over time; this causes the *structure* of the datatree changes due to the number of reported metrics varying. Most often this happens when the service being monitored changes availability (when a service “goes down” or “comes up”), although some services report additional metrics once they have stabilised. The Apache HTTP server is an example; after an initial delay, it provides a measure of bandwidth usage.

When a change in a datatree structure is detected, MonAMI will invalidate all its internal caches that use this datatree; independent caches are left unaffected. Subsequent requests to a target for fresh data will gather new data, either freshly cached or direct from the monitoring target. This allows the new structure to propagate independent of the `cache` attributes.

5. On-demand monitoring

This example demonstrates on-demand monitoring. On-demand monitoring is where MonAMI will not trigger gathering information internally. Instead, something outside of MonAMI requests information. Only when metrics are requested will MonAMI acquire the data.

Another characteristic of on-demand monitoring is that one can present a large number of metrics. After MonAMI has started, the user can choose which metrics they are interested in. Typically, the user can change their selection (the user decides to start monitoring some metric, or stop monitoring it) or stop monitoring altogether. MonAMI will adjust accordingly; for example, if no users are requesting on-demand monitoring information and no internally triggered monitoring is configured, MonAMI will not gather any information. Using on-demand monitoring, one can make provision for monitoring a vast number of metrics that are normally uninteresting without burdening the monitored systems.

The example used in this section is KSysGuard. It is an excellent demonstration of the benefits of on-demand monitoring: we do not specify in the MonAMI configuration which of the available metrics is to be monitored. Rather, the user (through the KSysGuard GUI) chooses what to monitor. The user can change their mind by adding or removing metrics. When the user closes the KSysGuard application, no further monitoring is triggered.

KSysGuard is not unique in using on-demand monitoring: Nagios uses it for its NRPE monitoring. However, KSysGuard is both easy to use and easy to configure, making it ideal for demonstration purposes.

Configuration file

The following example will include both internally triggered monitoring (as in the previous examples) and on-demand monitoring. As before, a log file is maintained that records the current status of two metrics. In addition, users are allowed to query any of the available metrics through the *KSysGuard* interface. *KSysGuard* was chosen as it is readily available and simple to understand, but the on-demand concept applies to some other monitoring systems.

As before, copy the following configuration file as `/etc/monami.d/example.conf`.

```
##
## MonAMI by Example, Section 5
##

# Our root filesystem
[filesystem]
name = root-fs
location = /
cache = 2 ❶

# Our /home filesystem
[filesystem]
name = home-fs
location = /home
cache = 2

# Bring together all information we want KSysGuard to see
[sample] ❷
name = ksysguard-sample
read = root-fs, home-fs
cache = 10 ❸

# Also record the root and /home available space
[sample]
read = root-fs.capacity.available, \
      home-fs.capacity.available
write = root-fs-log
interval = 1m
```

```
[filelog]
name = root-fs-log
filename = /tmp/monami-root-fs-log

# Allow KSysGuard to request information
[ksysguard]
read = ksysguard-sample ❹
```

Some points to note:

- ❶ Each of the *filesystem* targets has a two second cache. This is to catch when periodic and on-demand monitoring requests happen to coincide.
- ❷ The named sample aggregates information for the *ksysguard* target. We don't specify interval or write attributes as monitoring will be “on demand”: *ksysguard* target will request information as *KSysGuard* requires it.
- ❸ The ten-second cache is included for safety: it prevents a *KSysGuard* user from overloading the services.
- ❹ The `read` attribute declares from which target the datatree presented to *KSysGuard* is obtained.

Running the example with telnet

This section is included for those who do not have the *KSysGuard* GUI installed or for those who want to see some of the gory detail! If you have access to *KSysGuard* and don't want to type in commands in a command-line environment, feel free to move onto the next section.

To emphasise the point: people do not normally use the *telnet* client program for monitoring; we use it here but the program is included on almost every Unix-like platform. You are not expected to use this routinely. It is here to illustrate the idea of on-demand monitoring in environments that do not have *KSysGuard* installed. This hands-on approach also demonstrates that you (the user) are in control of the monitoring schedule, not MonAMI.

Start MonAMI in the usual non-detaching verbose mode: `/usr/bin/monamid -fv`. After starting MonAMI, and in a separate terminal, type `telnet localhost 3112`. This starts the telnet client so it connects to the MonAMI running on the local machine. You will see something like:

```
paul@donachain:~$ telnet localhost 3112
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^'.
MonAMI 0.9

-- Welcome to the ksysguard plugin for MonAMI --
This aims to be compatible with ksysguard's own ksysguardd daemon
and provide information for ksysguard to display.

ksysguardd>
```

The text `ksysguardd>` is a prompt, indicating you can enter a command. To start with, type the command `MONITORS`, which lists all available metrics. You will see a list of metrics, followed by the prompt.

```
ksysguardd> MONITORS
root-fs/fragment size    integer
root-fs/blocks/size     integer
root-fs/capacity/free   float
More metrics are reported.
home-fs/files/available integer
home-fs/flag            integer
home-fs/namemax        integer
ksysguardd>
```



Different separator character

As with MonAMI, KSysGuard groups metrics within a tree structure. However, KSysGuard expects path elements to be separated using the forward slash (/). The ksysguard plugin presents the data correctly for the KSysGuard GUI, so the metrics have a different separator character.

To obtain the metadata about a metric, type in the metric's full name immediately followed by a question mark. This will return a single line, containing the metric name, the minimum and maximum values for that metric (both zero if they are unknown) followed by the units. For example:

```
ksysguardd> home-fs/capacity/total?
home-fs/capacity/total 0 0 MiB
ksysguardd>
```

To obtain a metric's current value, type in the metric's full name. You can repeat this as often as you like; if correctly configured, MonAMI's caching policy will prevent queries from impacting heavily on any monitored system.

```
ksysguardd> home-fs/capacity/available
11177.742188
ksysguardd> root-fs/capacity/available
126.091797
ksysguardd>
```

To terminate the connection, either close the telnet connection (**Ctrl+] then type close and Enter**), or instruct the MonAMI ksysguard plugin to disconnect (**Ctrl+D**).

You can run multiple connections concurrently, requesting the same or different metrics simultaneously.

Running the example with KSysGuard

Start MonAMI as previously (`monamid -fv`). Whilst MonAMI is running, it will listen on port 3112 for incoming connections. This is the default port to which KSysGuard will attempt to connect.

With MonAMI running, start the KSysGuard GUI: **ksysguard**. From the menu bar, select File → Connect host... The resulting dialogue box will request information about how to connect to MonAMI. Enter the name of the host MonAMI is running on in **Host** and ensure that, under **Connection Type**, **Daemon** is selected and that **Port** is set to 3112.

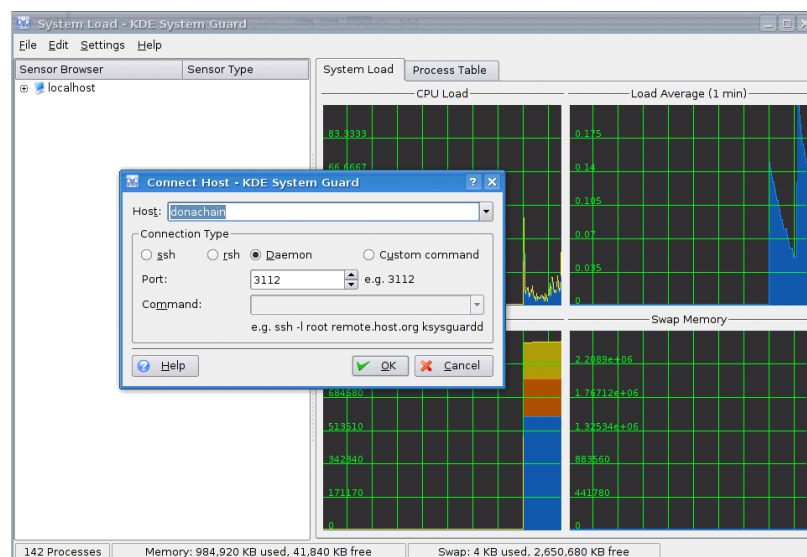


Figure 1. KSysGuard about to connect to MonAMI.



KSysGuard and host names

The GUI will not accept `localhost` as the host name for MonAMI. This is because the GUI starts its own data-gathering daemon, `ksysguardd`, automatically. It considers `localhost` always to refer to this data-gathering daemon. To connect to MonAMI, use either machine's node name (run `uname -n`) if this is different to `localhost`, or its fully-qualified domain name.

Clicking on the Ok button should result in the host name appearing in the left **Sensor Browser** pane. If this doesn't work, check that MonAMI is running, that you entered the host name correctly and there is no firewall blocking connections. If connecting to MonAMI remotely, you must authorise the remote connection using the `allow` attribute (see the MonAMI User Guide or `monami.conf(5)` manual page for further details).

Once the hostname appears on the sensor browser, click on the **+** symbol (next to the hostname) to see the available sensors. If MonAMI is reporting many metrics, expanding the tree of available metrics can be a slow process: be patient.

To begin monitoring, drag and drop a metric to a display area. It is often easiest to create a new worksheet. Select **File** → **New worksheet...** to generate a new worksheet in which you can drag-and-drop sensors.

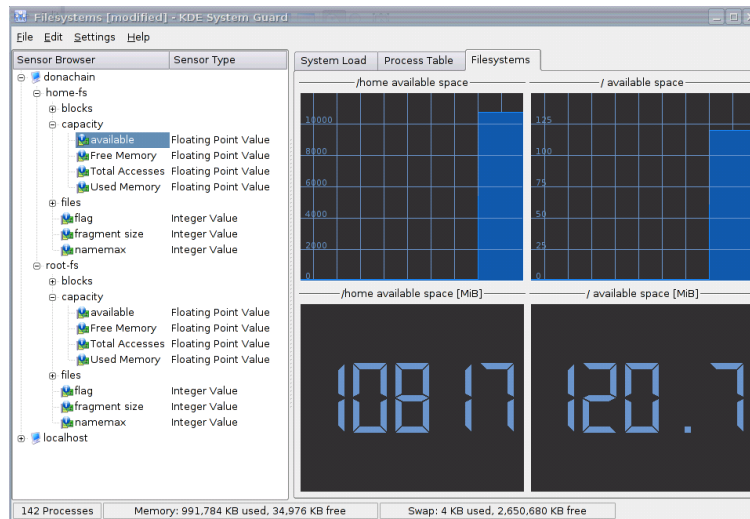


Figure 2. KSysGuard monitoring available capacity.



Wrong metric names

You may notice that some of the metrics are inappropriately labelled. For example, metrics that have a path that ends with "free" are displayed as "Free Memory", those ending with "total" are displayed as "Total Accesses" and "used" as "Used Memory". This is a bug in the current version of KSysGuard.

Mixed monitoring

The above configuration has MonAMI undertake internally triggered (or periodic) monitoring whilst allowing on-demand monitoring. MonAMI will record data to the `root-fs-log` reporting target every minute.

This file-logging of data is independent of and concurrent to any KSysGuard monitoring, except that MonAMI will honour the cache settings. Each of the two filesystem monitoring targets will cache results for two seconds, irrespective of whether the request came from KSysGuard or from the anonymous sample section.

6. Plotting data with Ganglia

This section describes interfacing MonAMI with Ganglia. To try the configuration within this section you will need to have Ganglia available. Setting up Ganglia is easy, but is outside the remit of this tutorial. Further details, including Ganglia tutorials and binary packages, are available from the *Ganglia project site* [<http://ganglia.sourceforge.net/>].

This section will start with simple filesystem monitoring and build into a demonstration of full monitoring of Torque and Maui monitoring. This example can be adapted to monitor other systems that MonAMI supports, such as DPM, MySQL, Tomcat...

Configuration file

As before, copy the following configuration file as `/etc/monami.d/example.conf`.

```
##
## MonAMI by Example, Section 6
##

# Our root filesystem
[filesystem]
name = root-fs
location = /
cache = 2 ❶

# Our /home filesystem
[filesystem]
name = home-fs
location = /home
cache = 2 ❶

# Once a minute, record / and /home available space.
[sample]
read = root-fs.capacity.available, \
      home-fs.capacity.available
write = ganglia
interval = 1m ❷

# Ganglia target that accepts data
[ganglia]
❸
```

Some points to note:

- ❶ The `cache` attributes enforce a “never more than” policy: never more than once every two seconds.
- ❷ MonAMI controls how often data in Ganglia is updated: updating data once a minute is a common choice.
- ❸ The `ganglia` plugin will attempt to read from the `gmond.conf` file. The plugin will search for this file in a couple of standard places. If found, the plugin will know how to send metric-update messages without any attributes.

Running the example

It is recommended (although not essential) that the Ganglia monitoring daemon `gmond` is running on the machine. The `gmond` daemon monitors many low-level facilities and also provides a common place for Ganglia configuration: the file `gmond.conf`. If the `gmond.conf` is not in a standard location, you can specify where to find it using the `config` attribute. If `gmond` isn't installed, you can specify how to send Ganglia metric update messages using other attributes within the `ganglia` target. Refer to the MonAMI User Guide or the `monami.conf(5)` manual page for further details.

When run, MonAMI will emit the corresponding UDP multicast packets containing the metric information. The various gmond daemons that are listening will pick up the metrics and, when queried by the gmetad daemon, will present the latest information.

The gmetad daemon will poll gmond daemons periodically. By default this is 15 seconds, although it can be altered through the gmetad.conf configuration file. This means that the new metrics defined in the above example, under the default Ganglia configuration, may take up to 75 seconds to be visible in the web front-end.

Sample results

To view the results, look at the Ganglia page (the Host-specific view) for server that is running MonAMI. You should see two additional graphs towards the bottom of the page entitled “root-fs.capacity.available” and “home-fs.capacity.available”. Here is an example:

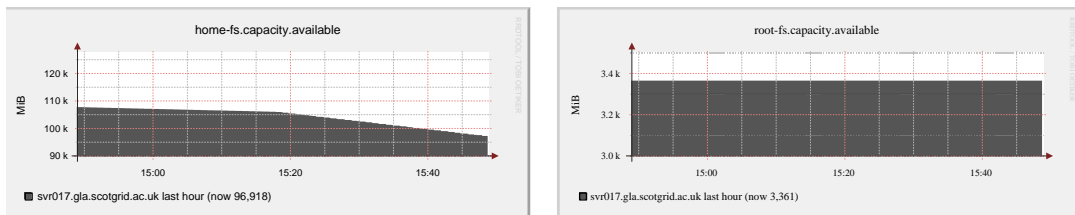



Figure 3. Ganglia generic single-metric graphs

Ganglia also provides a list of extra metrics on the Gmetrics web-page. This page has a link to from the Host view: follow the “Gmetrics” link on the left side. On the Gmetrics page, you should see two entries underneath the “User Defined Metrics (gmetrics)” title.

Grid Servers > svr017.gla.scotgrid.ac.uk



This host is up and running.

[Back to Host View](#)

User Defined Metrics (gmetrics)

TN	TMAX	DMAX	NAME	VALUE
21	60	191	root-fs.capacity.available	3360.546875 MiB
21	60	191	home-fs.capacity.available	108578.539062 MiB

Figure 4. Example Gmetric page.

Each metric reported to Ganglia is named after its path within the datatree. MonAMI keeps track of the units for each metric. This information is passed on to Ganglia, allowing it to display the current value with the appropriate units.

Dealing with old data

The figure above shows a table where the first three columns are TN, TMAX and DMAX. The TN value is the number of seconds since the metric was last updated. If the webpage is reloaded, TN will increase until a new metric value is received by Ganglia.

TMAX indicates the freshness of a metric. If TN exceeds TMAX, then Ganglia is expecting a new value. However, TMAX is only advisory: Ganglia takes no action when TN exceeds TMAX.



Delays in getting new data

Sometimes TN will exceed TMAX. Bear in mind that the PHP web-page queries gmetad to obtain information. In turn, gmetad will poll one or more gmond instances periodically (by default, every 15 seconds). This may introduce a delay between new metric values being sent and becoming visible within the web-pages.

DMAX indicates for how long an old metric should be retained. If TN exceeds DMAX then Ganglia will consider that that metric is no longer being monitored. Therefore, it will discard information about that metric. Historic data (the RRD files from which the graphs are drawn) will be kept, but the corresponding graphs will no longer be displayed. If fresh metric values become available, then Ganglia will start redisplaying the metric's graphs and the historic data may contain a gap.

Choosing a value for DMAX is a compromise. Too short an interval risks metrics being dropped accidentally if a data-source takes an unusually long time to provide information, whilst too long an interval results in unnecessary delay between a metric no longer being monitored and Ganglia dropping that metric.

Automatic dropping of old metrics can be disabled by setting DMAX to zero. If this is done, then there is no risk of Ganglia mistakenly dropping a metric. However, if a metric receives no further updates, Ganglia will continue to plot the last value indefinitely (or until `gmond` `gmetad` daemons are restarted, in that order). Unless the daemons are restarted, false data will be displayed, providing a potential source of confusion.



Calculating DMAX

MonAMI will calculate an estimate for DMAX based on observed behaviour of the monitoring targets. A fresh estimate is calculated for each metric update. If the monitoring environment changes, MonAMI will adjust the corresponding metrics' DMAX value, allowing Ganglia to adapt to changes in the underlying monitoring system's behaviour.

Preventing metric-update loss

An issue with providing monitoring information for Ganglia is how to deal with large numbers of metrics. MonAMI can provide very detailed information, resulting in a large number of metrics. This can be a problem for Ganglia.

The current Ganglia architecture requires each metric update be sent as an individual metric-update message. On a moderate-to-heavily loaded machine, there is a chance that `gmond` may not be scheduled to run as the messages arrive. If this happens, the incoming messages will be placed within the network buffer. Once the buffer is full, any subsequent metric-update messages will be lost. This places a limit on how many metric update messages can be sent in one go. For 2.4-series Linux kernels the limit is around 220 metric-update messages; for 2.6-series kernels, the limit is around 400.



Trying not to cause problems.

The *ganglia* plugin tries to minimise the risk by sending metric-update messages in bursts of 200 metric updates (so less than the 220 metric-update limit on 2.4-series kernels) with a short pause between successive bursts. The time between bursts gives the `gmond` daemons time to react. There are attributes that fine-tune this behaviour, which the User Guide discusses.

One simple solution is to split the set of metrics into subsets. If these subsets are updated independently and none have sufficient metrics to overflow the network buffer, then there will be no metric-update message loss. If more than one system is to be monitored, this splitting is easily and naturally achieved.

The following example shows a configuration for monitoring a local Torque and Maui installation. The configuration demonstrates how two sample stanzas can isolate the monitoring work. This spreads the monitoring load and reduces the impact on Ganglia.

```
##
## MonAMI by Example, Section 6.
## Torque and Maui monitoring
##

[torque]
cache = 60 ❶

[maui]
user = root ❷
cache = 60 ❶
```



```
[sample] ③
read = torque.Jobs, torque.Queues.Execution
write = ganglia
interval = 1m

[sample] ③
read = maui, !maui.Fairshare.User
write = ganglia
interval = 1m

[ganglia]
④
```

Some points:

- ① Make sure we never query the Torque or Maui services more than once a minute.
- ② The user attribute specifies as which user the MonAMI *maui* plugin should claim to be running; the value `root` is commonly authorised. It is possible to configure Maui to accept a non-`root` user, although this brings no additional security. For more details, see the MonAMI User Guide.
- ③ The Torque and Maui monitoring are done independently.
- ④ We assume that Ganglia `gmond` is installed. If this is so, the MonAMI ganglia plugin will parse the `gmond` file for information on how to send metric-update messages.

The following shows the gmetric page, showing a subset of the available metrics:

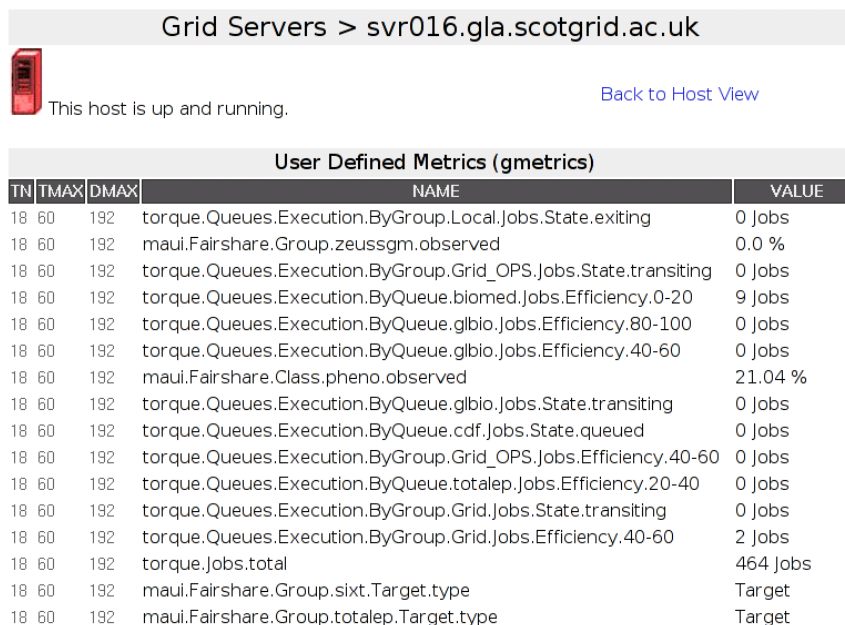


Figure 5. Ganglia gmetric page showing some metrics for Torque and Maui.

Producing complex graphs

MonAMI can produce a large number of metrics. The standard Ganglia web front-end shows a single graph (towards the bottom of page) for each metric. Although these single-metric graphs are functional, they help little towards understanding the “bigger picture” if there are a large number of metrics being recorded.

Ganglia's web front-end contains a number of graphs that aggregate metrics provided by the `gmond` daemon. One such graph shows the number of runnable processes, the 1-minute load average and the number of CPUs. Another shows the total in-core memory of the machine, split by usage. These

graphs provide a good overview of the computer's current behaviour and greatly assist in diagnosing problems as they arise.

Unfortunately, these default aggregation graphs are hard-coded into the PHP. There is currently no standard way to extend the Ganglia web front-end to include custom graphs. There is also no method to specify that certain graphs should be displayed for only one particular machine: the one that is running MonAMI collecting the interesting metrics.



The “external” package

MonAMI provides monitoring information for any number of monitoring systems. Strictly speaking, its job is done once data is within those systems. However, to get the most out of any particular system, you may need to tweak the monitoring system, or expand some scripts to better match the breadth of data MonAMI provides.

The external package contains a number of application-specific instructions, sample configuration files and modules. It is both a reference point for using MonAMI with particular monitoring systems and a platform with which to explore what is possible.

The section of the “external” package for Ganglia includes a PHP framework called *multiple-graphs*. This includes support for frames (in which multiple graphs and tables can be included) and pop-ups (allowing the display of context-specific information). The package also includes support for host-specific graphs.

The following figure shows the multiple-graphs framework in action: some of the Torque monitoring data is shown as graphs and pie-charts.

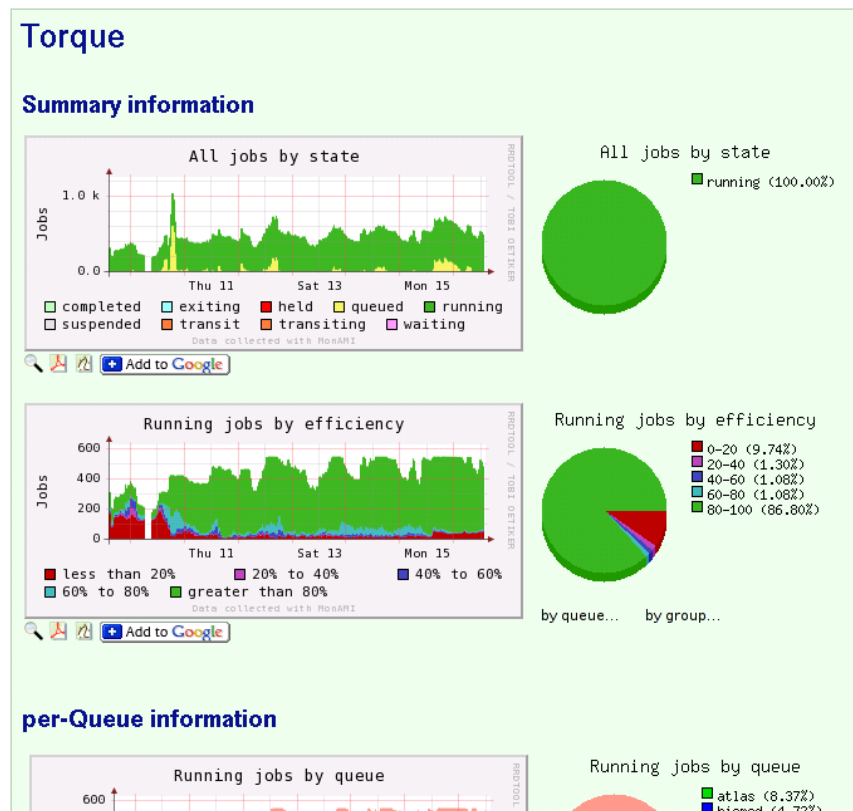


Figure 6. Some Torque graphs from the Host-view page.

The data is provided by MonAMI running with a configuration similar to the above example. The graphs and pie-charts are generated using the multiple-graphs library. The exact PHP configuration for generating the Torque and Maui frame is also available within the “externals” package as one of the examples. These are documented and include installation instructions.

7. Using Nagios to trigger alerts

Nagios is a sophisticated monitoring system. One of its major strengths is its ability to monitor many services and provide a configurable response to services going into *Warning* or *Critical* states. More information is available at the *Nagios project home page* [<http://www.nagios.org/>].

This section will show how to configure MonAMI and Nagios to alert if something is wrong. To get the most out of this section, you will need Nagios configured on your computer or computers. Nagios requires little configuration for a basic setup: the default configuration should be sufficient.

Currently, MonAMI offers only passive monitoring. This is where MonAMI sends updates to Nagios indicating the current state of the monitored services. Passive monitoring requires some extra configuration. The section Section 7, “Setting up Nagios” gives an overview of this.

Configuration file

As before, copy the following configuration file as `/etc/monami.d/example.conf`.

```
##
## MonAMI by Example, Section 7
##

# Our root filesystem
[filesystem]
name = root-fs
location = /
cache = 2

# Our /home filesystem
[filesystem]
name = home-fs
location = /home
cache = 2

# Once a minute, record / and /home available space.
[sample]
read = root-fs, home-fs
write = nagios
interval = 1m ❶

# Nagios target that sends alert data.
[nagios]
host = nagios-svr.example.org❷
port = 5668
password = NotSecretEnough

service❸ = rfs❹ : ROOT_FILESYSTEM❺
check❻ = rfs❹ : root-fs.capacity.available❼, 10❸, 0.5❹
check = rfs : root-fs.files.available, 400, 100

service = hfs : HOME_FILESYSTEM
check = hfs : home-fs.capacity.available, 10, 0.5
check = hfs : home-fs.files.available, 400, 100
```

Some points to note:

- ❶ Once a minute, this *sample* target will query the current status of the filesystems and send the new data to the *nagios* target.
- ❷ This is the host that has the *nscd* daemon. `nagios-svr.example.org` is an example FQDN and should be replaced with the hostname of your Nagios server.
- ❸ Each *nagios* target should have at least one *service* attribute. A service is what is reported back to Nagios as being *OK*, *Warning*, or *Critical* (or, if misconfigured, as *Unknown*). Without at least one service, a *nagios* target has nothing to do!

- ④ a short name used within the MonAMI configuration for this service.
- ⑤ the name of the service according to Nagios. By convention, this is in capital letters.
- ⑥ Each service should have at least one `check` attribute associated with it. The checks determine the current status of a service.
- ⑦ The name of a metric. This is the path within the supplied datatree.
- ⑧ The first value that will result in the service going into `Warning` state.
- ⑨ The first value that will result in the service going into `Critical` state.

A *service* is something that Nagios will monitor on a specific server. In general, services are abstract, (usually high-level) activity or resources that the server provides. Examples of services include a MySQL DBMS, Torque resource manager and filesystem space.

Nagios considers each service as being in one of four possible states:

OK	the service is behaving normally.
Warning	normal behaviour continues but there is an early indicator of a problem. If there is any impact on available service, it is slight.
Critical	normal behaviour is no longer possible. If any service is still available it is heavily impacted and complete failure is expected soon.
Unknown	Nagios does not know the status of this service.

Setting up Nagios

In order for a Nagios host to accept status update messages from MonAMI, it must either run the *nscad* daemon (*nscad*) or configure an *inetd*-like daemon (e.g., *inetd* or *xinetd*) to accept these connections and run the *nscad* program indirectly.

The *nscad* daemon, whether running independently or via an *inetd*-like daemon, will receive the update notice and write a command to the Nagios “external commands” socket. For this to work, the socket must be created by the Nagios daemon and the *nscad* daemon must have write access to this socket. The former is controlled by a configuration option (`check_external_commands` usually found in `nagios.cfg`) whilst the latter requires the directory in which the socket is created (as defined in the `command_file` option and typically `/var/log/nagios/rw/`) to have the correct permissions.

Each MonAMI check must have a corresponding entry in the Nagios configuration. The following creates a generic template for use with MonAMI passive updates.

```
define service {
    name                monami-service
    use                 generic-service
    active_checks_enabled 0
    passive_checks_enabled 1
    register            0
    check_command        check_monami_dummy
    notification_interval 240
    notification_period 24x7
    notification_options c,r
    check_period         24x7
    contact_groups       monami-admins
    max_check_attempts   3
    normal_check_interval 5
    retry_check_interval 1
}
```

Despite the service being purely passive, a valid `check_command` setting is still needed. We use the command `check_monami_dummy`, which is a simple command that always returns `True`:

```
define command {
```

```

command_name    check_monami_dummy
command_line    /bin/true
}

```

The final step is define the individual services that are to be monitored. The following Nagios configuration defines the two filesystem services defined above.

```

define service {
    use                monami-service
    host_name          svr017
    service_description ROOT_FILESYSTEM
}

define service {
    use                monami-service
    host_name          svr017
    service_description HOME_FILESYSTEM
}

```



Saying who you are

By default, *nagios* targets will use the local machine's fully-qualified domain name (FQDN) as the hostname. However, Nagios allows the configuration to specify the shorter hostname. In the above example, the hostname (svr017) is used instead of the longer FQDN (svr017.gla.scotgrid.ac.uk).

The localhost attribute allows you to configure what the MonAMI nagios target specifies as its identity. To correctly identify itself, the target would need localhost = svr017 within its nagios stanza.

Adjust your Nagios configuration to include the new template and checks and restart your Nagios service. Nagios is careful about checking the configuration before starting: if you have a mistake in your configuration file you must correct it before Nagios will start.

If the new services have been correctly configured you will see two new entries in Nagios' "Service Detail" web-page for the host. These will have the PASV symbol indicating that passive updates are accepted for this service, allowing MonAMI to send fresh data.

The two services describe above will be in an initial (or Pending) state when Nagios starts. They will remain in that state until MonAMI first sends data. The following figure shows these services.

svr017	HOME_FILESYSTEM	PASV ↓↓	PENDING	N/A	0d 0h 0m 12s+	1/3	Service is not scheduled to be checked...
	NTP	OK		10-17-2007 12:09:04	145d 2h 5m 13s	1/4	NTP OK - Offset 0.0002319812775 secs
	ROOT_FILESYSTEM	PASV ↓↓	PENDING	N/A	0d 0h 0m 12s+	1/3	Service is not scheduled to be checked...
	cfservd	OK		10-17-2007 12:10:36	149d 0h 16m 13s	1/4	TCP OK - 0.001 second response time on port 5308
	ssh	OK		10-17-2007 12:11:27	149d 0h 14m 33s	1/4	SSH OK - OpenSSH_3.6.1p2 (protocol 2.0)

Figure 7. Example of passively monitored services before MonAMI has sent data.

Writing checks

Within MonAMI, each service has one or more *checks* associated with it. The checks are simple tests; they determine the status of their corresponding service. For example, the filesystem service might have checks for available capacity and available inodes for the different partitions, the torque service might have checks that the torque daemon is contactable, that there aren't too many queued jobs, that there aren't jobs stuck in wait state and so on.

Numerical checks are written as a metric and two numbers separated by commas. The first number is when the check should go into *Warning* state; the second is when it should go into *Critical* state. The gradient of these numbers indicates which direction is "getting worse". If the first number is greater than the second then the metric is measuring resource exhaustion (e.g., available disk space, free memory, time spent idle); whereas, if the second number is larger than the first then the metric is

measuring resource usage (e.g., number of concurrent processes, number of jobs, number of network connections).

The following service and check attributes monitor available capacity for non-root users.

```
service = hfs : HOME_FILESYSTEM
check = hfs : home-fs.capacity.available, 10, 0.5
```

Consider a scenario where a user is downloading data, filling the /home partition. If the available capacity (i.e., home-fs.capacity.available) is 20 MiB, then HOME_FILESYSTEM is in state OK. If, when next measured, the available capacity has dropped to 10.01 MiB then HOME_FILESYSTEM is still OK. Once the measured value has dropped to 10 MiB, HOME_FILESYSTEM will be in Warning state. As the partition becomes further filled, it will stay in Warning state until, finally, the value drops to 0.5 MiB. At this point HOME_FILESYSTEM is in Critical state.

If there are multiple check attributes for a service attribute, the service's state is a combination of the different check states. The rule is simple: the most important state wins. The states, in the order of increasing importance, are: OK, Unknown, Warning and Critical. So, if all a service's checks are OK, then the service is OK. If most of the checks are OK but some are Unknown, then the service is in Unknown state. If there are some checks that are in Warning state but none yet in Critical state, then the service is in Warning state; but, if at least one check is Critical, then the service is in Critical state.

In the following example:

```
service = hfs : HOME_FILESYSTEM
check = hfs : home-fs.capacity.available, 10, 0.5
check = hfs : home-fs.files.available, 400, 100
```

If the available space drops to 10 MiB or the available files (the available inodes) drops to 400, then HOME_FILESYSTEM is Warning. If the available files drop to 100 or fewer, then HOME_FILESYSTEM becomes Critical, independent of the available space.

Running the example

As before, run MonAMI with the supplied configuration. Make sure you run MonAMI for at least one minute. There may be a slight further delay between the NSCA daemon writing the command to Nagios' socket and Nagios updating its internal state.

Once MonAMI has written data to Nagios, you will see the services' state change from Pending to one of the four normal states of a service. In the example below, both services are in state OK.

svi017	HOME_FILESYSTEM	PASSIVE OK	10-17-2007 12:34:49	0d 0h 0m 19s	1/3	MonAMI: home-fs.capacity.available = 108578.539062 MiB, home-fs.files.available = 15759607 files
	NTP	OK	10-17-2007 12:30:07	145d 2h 26m 36s	1/4	NTP OK: Offset 0.001298069954 secs
	ROOT_FILESYSTEM	PASSIVE OK	10-17-2007 12:34:49	0d 0h 0m 19s	1/3	MonAMI: root-fs.capacity.available = 3362.59375 MiB, root-fs.files.available = 663646 files
	cfservd	OK	10-17-2007 12:31:47	149d 0h 37m 36s	1/4	TCP OK - 0.000 second response time on port 5308
	sshd	OK	10-17-2007 12:33:27	149d 0h 35m 59s	1/4	SSH OK - OpenSSH_3.6.1p2 (protocol 2.0)

Figure 8. Example of passively monitored services after MonAMI has sent data.

8. Writing data into MySQL

Sometimes it is desirable to store monitoring data in a permanent and flexible form. This might be for generating reports in an automated fashion, for conducting ad-hoc trend analysis or just to see whether your site's current behaviour compares with something similar seen several months ago.

Ganglia, which uses RRDTool for data storage, allows the user to adjust the time-frame, permitting a historic view. However, the reference point is always the current time. One cannot generate a “last day” set of graphs for a period three months ago. Within the RRDTool file, aging data loses fidelity. This is a useful feature of RRDTool data storage, but it means that a graph showing a 24-hour period six months ago is necessarily less accurate.

To archive monitoring data without compromise an alternative storage mechanism is needed. One such solution is to store the data within a database and MySQL is a popular choice of database.

This section demonstrates how to configure MonAMI so it stores data within a MySQL database.

Configuration file

As before, copy the following configuration file as `/etc/monami.d/example.conf`.

```
##
## MonAMI by Example, Section 8
##

# Our root filesystem
[filesystem]
name = root-fs
location = /
cache = 2

# Our /home filesystem
[filesystem]
name = home-fs
location = /home
cache = 2

# Once a minute, record / and /home stats into MySQL database.
[sample]
read = root-fs, home-fs
write = mysql
interval = 1m

# Store data in MySQL database
[mysql]
database❶ = monitoring
user = monami-writer
password = somethingSecret
table❷ = filesystem
field❸ = root-available❹ : root-fs.capacity.available❺
field = home-available : home-fs.capacity.available
```

Some points to note:

- ❶ The database attribute must be specified when writing to a MySQL database and the corresponding database must already exist within MySQL.
- ❷ You must specify a table attribute but the table doesn't have to exist.
- ❸ The field attributes define which metrics are written to the columns.
- ❹ The name of a table column within the MySQL table.
- ❺ The path to a metric within supplied datatrees.

Setting up MySQL

Before running MonAMI, you must create the MySQL user `monami-writer` and create the monitoring database. If the user or database you intend to use already exist, then you can just reuse the existing ones. If you are using a user that already exists, make sure it has sufficient privileges to store the data.

The following SQL will create the monitoring database, create the `monami-writer` user and authorise this user to create tables within the database and to append data to those tables.

```
CREATE USER 'monami-writer' IDENTIFIED BY 'somethingSecret';
CREATE DATABASE monitoring;
GRANT CREATE,INSERT ON monitoring.* TO 'monami-writer';
```

Before appending new data, the MonAMI `mysql` plugin will check the table exists. If it doesn't the plugin will try to create the table.

The created table will be base on the datatree the `mysql` target receives: each `field` attribute will have a corresponding column in the created table and the storage type will be based on the metric. If a field attribute's metric is missing from the first datatree then MonAMI will create a `STRING` column for that metric.

You can create the table manually; the procedure is described in the MonAMI User Manual. If the table is created manually, then the MonAMI writer account can be granted less privileges; however, the increase in security is limited and manually creating the tables is a hassle, so it is easier to let MonAMI create the tables.

Running the example

Run MonAMI in the usual fashion (`/usr/bin/monamid -fv`) for at least one minute. During that minute, if the `mysql` target creates the storage table, it will display a message. So, the first time you run the plugin, you will see an extra line:

```
Starting up...
mysql> table filesystem doesn't exist, creating it...
```

As data is added to the database the row count will increase. The following shows a MySQL interactive session where the number of entries in the `filesystem` table is counted:

```
mysql> SELECT COUNT(*) from monitoring.filesystem\G
***** 1. row *****
COUNT(*): 20
1 row in set (0.00 sec)

mysql>
```

The same SQL can be executed from the command-line. For example:

```
paul@donachain:~$ mysql --skip-column-names -se \
> 'SELECT COUNT(*) FROM monitoring.filesystem;'
22
paul@donachain:~$
```

You may need to specify which MySQL user to use and (usually) request that `mysql` client prompts you for a password:

```
mysql -u user [-p] --skip-column-names -se SQL-query
```


Using the stored data

Once the monitoring data is in the MySQL database, it can be used as any other data stored in a database. This greatly increases the opportunity with which you can conduct post-analysis of the monitoring data.

Perhaps the quickest way of analysing the available data is to write SQL queries. However, this requires knowledge of SQL before one can write custom queries and other methods may be easier.



Handling awkward column names

The above example has column names of `root-available` and `home-available`. Because these names include the hyphen character (-), they must be quoted by placing these words inside back-ticks (e.g. ``root-available``) when used in SQL queries. The same is true if the column name is a MySQL reserved word, such as `SELECT` and `LIKE`.

Another option is to build dynamic web pages that display information from the database. The PHP language includes support for querying a MySQL database. It also includes support for building custom graphics through the GD library. Using these tools, it is relatively easy to build custom graphs to that are dynamic, based on the data stored in MySQL.

For the purpose of this tutorial, we shall use OpenOffice to analyse the monitoring data. There are two parts: how to obtain live data from the database within a spreadsheet and how to plot and export graphs using this data. The instructions are written for OpenOffice v2.2; other versions may require a slightly different process.

Although this tutorial uses OpenOffice, a similar procedure should work for other spreadsheet packages.

Creating a spreadsheet with monitoring data

This part of the tutorial shows how to link the data gathered in MySQL into an OpenOffice spreadsheet. This allows for easy post-analysis of the data using normal spreadsheet functions.

1. Start OpenOffice.
2. Create database document file. This document will hold OpenOffice's understanding of the MySQL database.
 - a. Select **File** → **New** → **Database** from the menu or click on the **New, Database** button. The **Database Wizard** dialogue box should appear.
 - b. Select **Connect to an existing database**, choose the **MySQL database** type and either click on the **Next>>** button or type **Alt+N**.
 - c. Select **Connect using JDBC (Java Database Connectivity)** and either click on the **Next>>** button or type **Alt+N**.
 - d. Complete the requested information:

Name of the database	<code>monitoring</code>
Server URL	<code>localhost</code> (or whatever is the MySQL server hostname). N.B. this field takes a hostname, <i>not</i> a URL.
Port number	use the default port.
MySQL JDBC driver class	<code>com.mysql.jdbc.Driver</code>

Then either click on the **Finish** button or type **Alt+F**.

- e. The **Save as** dialogue box will appear. Save the resulting OpenDocument Database document somewhere as `fs-monitoring.odb`
3. Create new spreadsheet that has data from the filesystems table imported.
 - a. Select **File** → **New** → **Spreadsheet** from the menu or click on the **New Spreadsheet** button. This will create a new, empty spreadsheet.
 - b. Select **View** → **Data Sources** from the menu or press **F4**. The **Data Sources** section should appear above the spreadsheet. The **Data Source Explorer** (left pane of the **Data Sources** section) should be visible and show a tree structure, with `fs-monitoring` as one of the root elements.
 - c. Expand the `fs-monitoring` and **Tables** branches, and select `monitoring.filesystems`. You should see data from the filesystems table appear within the right pane of the **Data Explorer** section.
 - d. Click on `monitoring.filesystems` and drag this over to a cell within the spreadsheet. You will see the data appear within the spreadsheet, which will be highlighted.

With the data from database is imported, you have the full spectrum of analysis tools available from the spreadsheet. You can perform statistical or trend analysis.



Updating the imported data

The spreadsheet remembers that the monitoring data was originally taken from a database. You can fresh the imported data by selecting one of the cells within the imported data and selecting **Data** → **Refresh Range** from the menu.

Creating graphs from monitoring data

Perhaps the most useful action is to plot a graph using the gathered data. The following procedure describes how to create a stand-alone image of a graph (as PNG-, EPS- or SVG- formatted file).

1. Start OpenOffice and load the spreadsheet containing the imported data.
2. Create a chart, using imported monitoring data.
 - a. Make sure the data is highlighted, then either select **Insert** → **Chart...** from the menu or click on the **Insert Chart** button. Then click and drag out a region on the spreadsheet where you want the graph to appear. After doing this, the **AutoFormat Chart** dialogue box will appear.
 - b. Make sure **First row as label** and **First column as label** options are both selected and either click on the **Next>>** button or press **Alt+N**.
 - c. Select the **Areas** option from **Choose a chart type** and either click on the **Next>>** button or press **Alt+N**.
 - d. Select the **Stacked** option from **Choose a variant** and either click on the **Next>>** button or press **Alt+N**.
 - e. Enter chart information:

Chart title Storage usage

X axis Available space / MB

Then click on the **Create** button or press **Alt+A**.

3. Export chart as a PNG-, EPS- or SVG- formatted file.

- a. With the chart selected, either select Edit → Copy from the menu or press **Ctrl+C**.
- b. Create a new drawing. Either select File → New → Drawing from the menu or click on the New, Drawing button. This will create a new, empty drawing window.
- c. Either select Edit → Paste from the menu or press **Ctrl+V**.
- d. With the chart still highlighted, select File → Export... from the menu.
- e. Within File format, select the appropriate format (e.g. PNG, EPS or SVG), enter a filename in the File name input and either click on Export... or press **Alt+E**
- f. For some file formats some additional information is required. If so, a dialogue box will appear requesting information. Complete this information and click on **OK**.

Watching database disk usage

Over time, the monitoring data stored within the MySQL database will increase. We must take care to keep the table sizes reasonable. MonAMI can be used to make sure sufficient space is available.

If MonAMI is running on the MySQL server is running, then the *filesystem* plugin can monitor the available space. The *nagios* plugin can provide an alert if the filesystem space becomes too low.

The MonAMI *mysql* plugin is also a monitoring plugin. In addition to stores monitoring data, it can query the current status of the MySQL database system and the tables stored within databases. The plugin does this using the MySQL API, so it can monitor the database remotely.

One of the branches (`mysql.Database`) contains information on each database. This information includes information on each of that database's tables. So the branch `mysql.Database.monitoring.Table.filesystems` contains information on our `filesystems` table, part of the monitoring database. The metric `...Table.filesystems.Datafile.current` records the current space taken up by the `filesystems` table in Bytes. Using this, one can monitor the space the table is taking up and trigger an alert once this table reaches a critical level and intervention is needed.

An example configuration is:

```
[filesystem]
name = root-fs
location = /
cache = 2

[filesystem]
name = home-fs
location = /home
cache = 2

# Once a minute, record / and /home stats into MySQL database and
# update Nagios status.
[sample]
read = root-fs, home-fs
write = mysql, nagios-fs
interval = 1m

# Every ten minutes, update Nagios status of MySQL storage usage.
[sample]
read = mysql.Database.monitoring.Table.filesystems.Datafile.current
write = nagios-mysql
interval = 10m
```

```
# Store data in MySQL database
[mysql]
database = monitoring
user = monami-writer
password = somethingSecret
table = filesystem
field = root-available : root-fs.capacity.available
field = home-available : home-fs.capacity.available

# Nagios checks for file-system
[nagios]
name = nagios-fs
host = nagios-svr.example.org
port = 5668
password = NotSecretEnough

service = rfs : ROOT_FILESYSTEM
check = rfs : root-fs.capacity-available, 10, 0.5
check = rfs : root-fs.files.available, 400, 100

service = hfs : HOME_FILESYSTEM
check = rfs : home-fs.capacity-available, 100, 10
check = rfs : home-fs.files.available, 400, 100

# Nagios checks for MySQL
[nagios]
name = nagios-mysql
host = nagios-svr.example.org
port = 5668
password = NotSecretEnough

service = ms : MYSQL_SPACE
# Warn: 20 MiB, Crit: 1 GiB
check = ms : mysql.Database.monitoring.Table.filesystems.Datafile.current,\
20971520, 1073741824
```