



Munin Documentation

Release 2.999.6-HEAD-2017-08-28-c56-gdb39385

The Munin project and its contributors

Aug 28, 2017

Contents

1	Preface	3
1.1	What is Munin ?	3
1.2	Conventions	3
1.3	Further Information	3
1.4	Bug Reporting Guidelines	5
1.5	Documenting Munin	7
2	Tutorial	11
2.1	Getting Started	11
2.2	Wildcard Plugins	13
2.3	Using SNMP plugins	14
2.4	Let Munin croak alarm	15
2.5	Munin and Nagios	20
2.6	Advanced Features	24
2.7	Extraordinary Usage	24
2.8	Monitoring the “unreachable” hosts	25
2.9	Troubleshooting	26
3	Munin’s Architecture	31
3.1	Overview	31
3.2	Components	32
3.3	Protocols	56
3.4	Syntax	58
3.5	API	60
4	Munin installation	61
4.1	Prerequisites	61
4.2	Installing Munin	62
4.3	Initial configuration	64
4.4	Upgrade notes	66
5	Munin Advanced	67
5.1	Advanced Network	67
6	Developing Munin	69
6.1	Munin development environment	69
6.2	Munin development tests	71
7	Developing Plugins	73
7.1	How to write Munin plugins	73
7.2	How To Write SNMP plugins	80

8	Reference	83
8.1	Nomenclature	83
8.2	Man pages	87
8.3	Examples and Templates	106
8.4	Other reference material	122
9	Other docs	139
9.1	Contributing	139
9.2	Developer Talk	139
10	Indices and tables	141

Contents:

What is Munin ?

Munin is a networked resource monitoring tool (*started in 2002*) that can help analyze resource trends and *what just happened to kill our performance?* problems. It is designed to be very plug and play.

A default installation provides a lot of graphs with almost no work.

Note: In Norse mythology Hugin and Munin are the ravens of the god king Odin. They flew all over Midgard for him, seeing and remembering, and later telling him. *Munin* means *memory* in old Norse.

Conventions

Example Consistency

Examples should be consistent throughout the documentation. We should standardize on a set of values/examples used, e.g.:

- Example domains: example.com, example.org, example.net
- Example hosts: foo.example.com, bar.example.com
- Example IP addresses: 192.0.2.0/24 (pick from the range 192.0.2.0 - 192.0.2.255)
- Example plugins: FIXME
- Example datasources: FIXME

[RFC2606](#) details which hostnames are reserved for such usage.

[RFC3330](#) details special-use IPv4 addresses.

Further Information

Besides the official documentation, that is this guide, there are many other resources about Munin.

The fact that info is scattered around makes it sometimes difficult to find relevant one. Each source has its purpose, and is usually only well-suited for some kind of documentation.

Munin Guide

These are the pages you are currently reading. It is aimed at the first read. The chapters are designed as a walk-through of Munin's components in a very guided manner. Its read constitutes the **basis** of every documentation available. Specially when asking live (*IRC*, *mailing-lists*) channels, users there will expect that you read the Guide prior to asking.

It is regularly updated, as its sources are directly in the munin source directory, the last version can always be accessed online at <http://munin.readthedocs.org/> thanks to [ReadTheDoc](#).

It is specially designed for easy contribution and distribution thanks to [Sphinx](#). That aspect will be handled in [Contributing](#).

Web

The [Munin web site](#) is the other main source of information. It has a wiki format, but as spammers have become lately very clever, all content is now added by registered users only.

Information there has the tendency of being rather extensive, but old. This is mostly due to the fact that it was the first and only way of documenting Munin. So in case there is conflicting information on the wiki and on the Guide, better trust the Guide. We are obviously very glad if you can pinpoint the conflicting infos so we can correct the wrong one.

Still, a very important part is the [FAQ](#) (Frequently Asked Questions), which contains many answers to a wide array of questions. It is the only part of the documentation in the wiki that is still regularly updated.

GitHub

The [Munin GitHub](#) has slowly become the center of all the community-driven development. It is a very solid platform, and despite its drawback of delegation of control, given the importance it has today, no-one can ignore it. The mere fact that we opened a presence there has increased the amount of small contributions by an order of magnitude. Given that those are the meat of a global improvement, it ranked as a success.

Main Repository

Therefore, we will move more and more services to cloud platforms as GitHub, as it enables us to focus on delivering software and not caring about so much infrastructure.

We already moved all code pull requests there, and new issues should be opened there also. We obviously still accept any contribution by other means, such as email, but as we couldn't resist the move from SVN to GIT, we are moving from our Trac to GitHub.

Contrib Repository

The [contrib](#) part is even more live than before. It has very successfully replaced the old [MuninExchange](#) site. Now, together with the [Plugin Gallery](#) it offer all the useful features the old site offered, and is much easier to contribute to. It also ease the integration work, and therefore shortens the time it takes for your contributions to be reviewed and merged.

Mailing Lists

If you don't find a specific answer to your question in the various documentations, the mailing lists are a very good place to have your questions shared with other users.

- [subscribe to the munin-users list \(English\)](#)
- [subscribe to the munin-users-de \(German\)](#)
- [subscribe to the munin-users-jp \(Japanese\)](#)

Please also consult the list archives. Your Munin issue may have been discussed already.

- [munin-users list archive \(English\)](#)
- [munin-users-de list archive \(German\)](#)
- [munin-users-jp list archive \(Japanese\)](#)

It happens that they were much more used in the previous years, but nowadays it is much more common to seek an immediate answer on a specific issue, which is best handled by *IRC*. Therefore the mailing lists do appear very quiet, as most users go on other channels.

IRC

The most immediate way to get hold of us is to join our IRC channel:

```
#munin on server irc.oftc.net
```

The main timezone of the channel is Europe+America.

If you can explain your problem in a few clear sentences, without too much copy&paste, IRC is a good way to try to get help. If you do need to paste log files, configuration snippets, scripts and so on, please use a [pastebin](#).

If the channel is all quiet, try again some time later, we do have lives, families and jobs to deal with also.

You are more than welcome to just hang out, and while we don't mind the occasional intrusion of the real world into the flow, keep it mostly on topic, and don't paste random links unless they are *really* spectacular and intelligent.

Note that `m-r-b` is our beloved `munin-relay-bot` that bridges the `#munin` channel on various IRC networks, such as Freenode.

Yourself!

Munin is an open-source project.

As such, it depends on the user community for ongoing support. As you begin to use Munin, you will rely on others for help, either through the documentation or through the mailing lists. Consider contributing your knowledge back. Read the mailing lists and answer questions.

If you learn something which is not in the documentation, write it up and contribute it. If you add features to the code, contribute them.

Bug Reporting Guidelines

When you find a bug in Munin we want to hear about it. Your bug reports play an important part in making Munin more reliable because even the utmost care cannot guarantee that every part of Munin will work on every platform under every circumstance.

The following suggestions are intended to assist you in forming bug reports that can be handled in an effective fashion. No one is required to follow them but doing so tends to be to everyone's advantage.

We cannot promise to fix every bug right away. If the bug is obvious, critical, or affects a lot of users, chances are good that someone will look into it. It could also happen that we tell you to update to a newer version to see if

the bug happens there. Or we might decide that the bug cannot be fixed before some major rewrite we might be planning is done. Or perhaps it is simply too hard and there are more important things on the agenda. If you need help immediately, consider obtaining a commercial support contract.

Identifying Bugs

Before you report a bug, please read and re-read the documentation to verify that you can really do whatever it is you are trying. If it is not clear from the documentation whether you can do something or not, please report that too; it is a bug in the documentation. If it turns out that a program does something different from what the documentation says, that is a bug. That might include, but is not limited to, the following circumstances:

- A program terminates with a fatal signal or an operating system error message that would point to a problem in the program. (A counterexample might be a “disk full” message, since you have to fix that yourself.)
- A program produces the wrong output for any given input.
- A program refuses to accept valid input (as defined in the documentation).
- A program accepts invalid input without a notice or error message. But keep in mind that your idea of invalid input might be our idea of an extension or compatibility with traditional practice.
- Munin fails to compile, build, or install according to the instructions on supported platforms.

Here “program” refers to any executable, not only the back-end process.

Being slow or resource-hogging is not necessarily a bug. Read the documentation or ask on one of the mailing lists for help in tuning your applications.

Before you continue, check on the TODO list and in the FAQ to see if your bug is already known. If you cannot decode the information on the TODO list, report your problem. The least we can do is make the TODO list clearer.

What to Report

The most important thing to remember about bug reporting is to state all the facts and only facts. Do not speculate what you think went wrong, what “it seemed to do”, or which part of the program has a fault. If you are not familiar with the implementation you would probably guess wrong and not help us a bit. And even if you are, educated explanations are a great supplement to but no substitute for facts. If we are going to fix the bug we still have to see it happen for ourselves first. Reporting the bare facts is relatively straightforward (you can probably copy and paste them from the screen) but all too often important details are left out because someone thought it does not matter or the report would be understood anyway.

The following items should be contained in every bug report:

- The exact sequence of steps from program start-up necessary to reproduce the problem. This should be self-contained; it is not enough to send in a bare log output without the plugin *config* and *fetch* statements.
- The best format for a test case for a restitution issue (graphing or HTML) is a sample plugin that can be run through a single munin install that shows the problem. (Be sure to not depend on anything outside your sample plugin). You are encouraged to minimize the size of your example, but this is not absolutely necessary. If the bug is reproducible, we will find it either way.
- The output you got. Please do not say that it “didn’t work” or “crashed”. If there is an error message, show it, even if you do not understand it. If the program terminates with an operating system error, say which. If nothing at all happens, say so. Even if the result of your test case is a program crash or otherwise obvious it might not happen on our platform. The easiest thing is to copy the output from the terminal, if possible.

Note: If you are reporting an error message, please obtain the most verbose form of the message. Use the *-debug* command line arg.

- The output you expected is very important to state. If you just write “This command gives me that output.” or “This is not what I expected.”, we might run it ourselves, scan the output, and think it looks OK and is

exactly what we expected. We should not have to spend the time to decode the exact semantics behind your commands. Especially refrain from merely saying that “This is not what Cacti/Collectd/... does.”

- Any command line options and other start-up options, including any relevant environment variables or configuration files that you changed from the default. Again, please provide exact information. If you are using a prepackaged distribution that starts the database server at boot time, you should try to find out how that is done.
- Anything you did at all differently from the installation instructions.
- The Munin version. If you run a prepackaged version, such as RPMs, say so, including any subversion the package might have. If you are talking about a Git snapshot, mention that, including the commit hash.
- If your version is older than 2.0.x we will almost certainly tell you to upgrade. There are many bug fixes and improvements in each new release, so it is quite possible that a bug you have encountered in an older release of Munin has already been fixed. We can only provide limited support for sites using older releases of Munin; if you require more than we can provide, consider acquiring a commercial support contract.
- Platform information. This includes the kernel name and version, perl version, processor, memory information, and so on. In most cases it is sufficient to report the vendor and version, but do not assume everyone knows what exactly “Debian” contains or that everyone runs on amd64.

Where to Report

In general fill in the bug report web-form available at the project’s [GitHub](#).

If your bug report has security implications and you’d prefer that it not become immediately visible in public archives, don’t send it to bugs. Security issues can be reported privately to <security@munin-monitoring.org>.

Do not send bug reports to any of the [user mailing lists](#). These mailing lists are for answering user questions, and their subscribers normally do not wish to receive bug reports. More importantly, they are unlikely to fix them. If you have some doubts about your issue being a bug, just drop by on [IRC](#) and ask there first.

If you have a problem with the documentation, the best place to report it is on [IRC](#) where most of the devs hang out. Please be specific about what part of the documentation you are unhappy with.

Note: Due to the unfortunate amount of spam going around, all of the above email addresses are closed mailing lists. That is, you need to be subscribed to a list to be allowed to post on it.

If you would like to send mail but do not want to receive list traffic, you can subscribe and set your subscription option to nomail.

Documenting Munin

Munin Guide

The pages you are just viewing ;)

The guide contains documentation about the Munin software.

It is written using reStructuredText¹.

If you have a [GitHub](#) account, you can even edit the pages online and send a pull request to contribute your work to the official Munin repository.

¹ Pod is a simple-to-use markup language used for writing documentation for Perl, Perl programs, and Perl modules. (And Munin plugins) See the [perlpod Manual](#) for help on the syntax.

Plugin Documentation

Plugin documentation is included in each plugin, written using the POD² style format

The command line utility `munindoc` can be used to display the info pages about the plugins. Call `munindoc buddyinfo` to get the documentation for plugin `buddyinfo`.

Have a look at the [munindoc instruction page](#) in our Trac wiki and edit or add the pod section in the plugins code file accordingly.

Finally send a patch or a pull request on github to help us improve the plugins documentation.

Munin Gallery

The plugin documentation is also included in the [Munin Gallery](#).

See our [Wiki page](#) for instructions how to contribute also example images for the gallery.

Unix Manual Pages

The manual pages are included in, and generated from, the *man pages in the Munin Guide*.

Munin's Wiki

The [wiki](#) contains documentation concerning anything *around* munin, whilst the documentation of the *Munin software* is here in the [Munin Guide](#).

This guide is **the** official documentation of Munin. It has been written by the Munin developers and other volunteers in parallel to the development of the Munin software. It aims to describe all the functionality that the current version of Munin officially supports.

To make the large amount of information about Munin manageable, this guide has been organized in several parts. Each part is targeted at a different class of users, or at users in different stages of their Munin experience. It is nevertheless still designed to be read as a book, sequentially, for the 3 first parts.

Further parts can be read in a more random manner, as one will search for a specific thing in it. Extensive care has been taken to fully leverage the hyperlinking abilities of modern documentation readers.

Preface

This is the part you are currently reading.

It focus on very generic information about Munin, and also gives some guidelines on how to interact with the Munin ecosystem.

Every Munin user should read this.

Part I - Tutorial

This part is an informal introduction for new users. It will try to cover most of what a user is expected to know about Munin. The focus here is really about taking the user by the hand and showing him around, while getting his hands a little wet.

Every Munin user should read this.

Part II - Architecture

² The reStructuredText (frequently abbreviated as reST) project is part of the Python programming language Docutils project of the Python Doc-SIG (Documentation Special Interest Group). Source Wikipedia: <http://en.wikipedia.org/wiki/ReStructuredText>
See the [reStructuredText Primer](#) for help on the syntax.

This part documents the various syntax that are used all throughout Munin. It is about every thing a normal user can look himself without being considered as a developer. It means the syntax of the various config files, and the protocol on the network.

Every Munin user should read this.

Part III - Install

This part describes the installation and administration of the server. It is about the OS part of Munin, which would be UID, path for the various components, how upgrades should be handled. It is also about how Munin interacts with its environment, such as how to secure a Munin install¹, how to enhance the performance of it.

Everyone who runs a Munin server, be it for private use or for others, should read this part.

Part IV - API

This part describes the programming interfaces for Munin for advanced users, such as the SQL schema of the metadata, or the structure of the spool directories. It should cover everything that isn't covered by *Part II*.

Part V - Advanced use

This part contains information for really advanced users about the obscure capabilities of Munin. Topics include undocumented stuff or even unwritten stuff that is still only in RFC phase.

Part VI - Reference

This part contains reference information about Munin commands, client and server programs. This part supports the other parts with structured information sorted by command or program. This also serves as a repository for the full sample configs that are studied in the *Part I*

Part VII - Others

This part contains assorted information that might be of use to Munin developers. This section serves usually as incubator for elements before they migrate to the previous parts.

Note: If you think that our Guide looks quite familiar, it is done on purpose, as we took a great inspiration of PostgreSQL's Manual. We even copied some generic sentences that were already very well worded.

In fact, the PostgreSQL project was, and still is, of a great guidance, as it does so many things right. The parts that were *imported* from PostgreSQL are obviously still under the PostgreSQL license².

¹ Even how to configure SELinux with Munin !

² We are not license experts, so if a PostgreSQL license guru has some issues with that, we'll be happy to resolve them together.

Welcome to the Munin Tutorial. The following few chapters are intended to give a simple introduction to Munin, monitoring concepts, and the Munin protocol to those who are new to any one of these aspects. We only assume some general knowledge about how to use computers. No particular Unix or programming experience is required. This part is mainly intended to give you some hands-on experience with important aspects of the Munin system. It makes no attempt to be a complete or thorough treatment of the topics it covers.

After you have worked through this tutorial you might want to move on to reading *Part II* to gain a more formal knowledge of the Munin protocol, or *Part IV* for information about developing applications for Munin. Those who set up and manage their own server should also read *Part III*.

Getting Started

Please refer to the *Nomenclature part* to understand the terms used in this guide.

Installation

Before you can use Munin you need to install it, of course. It is possible that Munin is already installed at your site, either because it was included in your operating system distribution or because the system administrator already installed it. If that is the case, you should obtain information from the operating system documentation or your system administrator about how to access Munin.

If you are installing Munin yourself, then refer to *Install Chapter* for instructions on installation, and return to this guide when the installation is complete. Be sure to follow closely the section about setting up the appropriate configuration files.

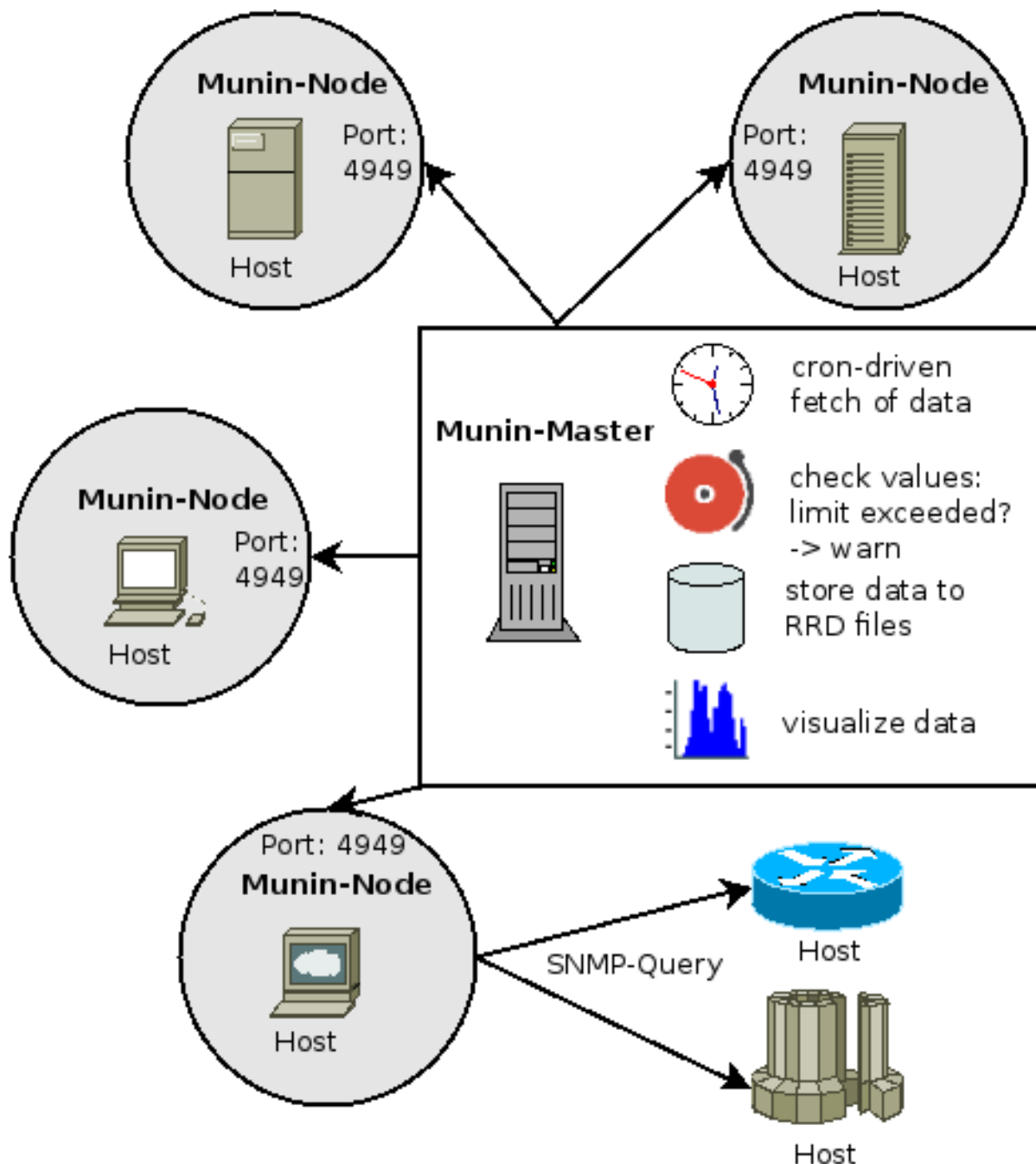
All the tutorial will assume a Debian installation, so all the commands are suited to the Debian package management system. As the one in Ubuntu is mostly the same, examples should work unchanged. For RPM-based systems, the equivalent yum command is left as an exercise to the reader, but should not be very hard to get.

We cannot speak about every other OS, but any UNIX-like have been reported to work. Your safest best should still to stick to a supported OS if you don't feel adventurous.

Also, you should need a dedicated server for the master role, as it mostly requires root access. Again, it is not required, but safety, and ability to copy/paste the samples, advise you to stick to these guidelines.

Architectural Fundamentals

Munin has a master-nodes architecture. See *Munin's Architecture* for the details.



Adding a Node

Thanks to the plug-and-play architecture of Munin, this is very easy. You obviously have to install the node part on the host you want to monitor.

```
$ apt-get install munin-node
```

This will install the node, some default plugins and launch it.

As the node runs as the root user in order to run plugins as any needed user, it now only listens on localhost as a security measure. You have to edit `munin-node.conf` in order to listen to the network, and add the master's IP on

the authorized list.

And don't forget to install `munin-node` also on the "Munin master" machine to monitor Munin's activities :-)

Wildcard Plugins

Wildcard plugins are plugins designed to be able to monitor more than one resource. By symlinking the plugin to different identifiers, the exact same plugin will be executed several times and give the associated output.

Operation & Naming Convention

Our standard example plugin is the `if_` plugin, which will collect data from the different network interfaces on a system. By symlinking `if_` to `if_eth0` and `if_eth1`, both interfaces will be monitored, and creating separate graphs, using the same plugin.

Wildcard plugins should, by nomenclature standards, end with an underscore (`_`).

Installation

Because a wildcard plugin normally relies on the symlink name to describe what item of data it is graphing, the plugin itself should be installed in the system-wide plugin dir (`/usr/share/munin/plugins` in Linux). Then via the `munin-node-configure` command, your munin-node will suggest shell commands to setup the required symlinks in the `servicedir` under `/etc/munin/plugins`.

For 3rd-Party wildcard plugins We recommend to install them into an own directory e.g. `/usr/local/munin/lib/plugins` and call `munin-node-configure` with flag `--libdir <your 3rd-party directory>`.

E.g.:

Before the plugin is installed:

```
# munin-node-configure --shell
```

Install the new plugin:

```
# mv /tmp/smart_ /usr/share/munin/plugins/smart_
```

Rescan for installed plugin:

```
# munin-node-configure --shell
ln -s /usr/share/munin/plugins/smart_ /etc/munin/plugins/smart_hda
ln -s /usr/share/munin/plugins/smart_ /etc/munin/plugins/smart_hdc
```

You can now either manually paste the symlink commands into a shell, or pipe the output of `munin-node-configure --shell` to a shell to update in one sequence of commands.

SNMP Wildcard Plugins

SNMP plugins are a special case, as they have not only one but two parts of the symlinked filename replaced with host-specific identifiers.

SNMP plugins follow this standard: `snmp_[hostname]_something_[resource to be monitored]`

E.g.: `snmp_10.0.0.1_if_6`

which will monitor interface 6 on the host 10.0.0.1. The *unlinked* filename for this plugin is `snmp__if_` (note two underscores between `snmp` and `if`).

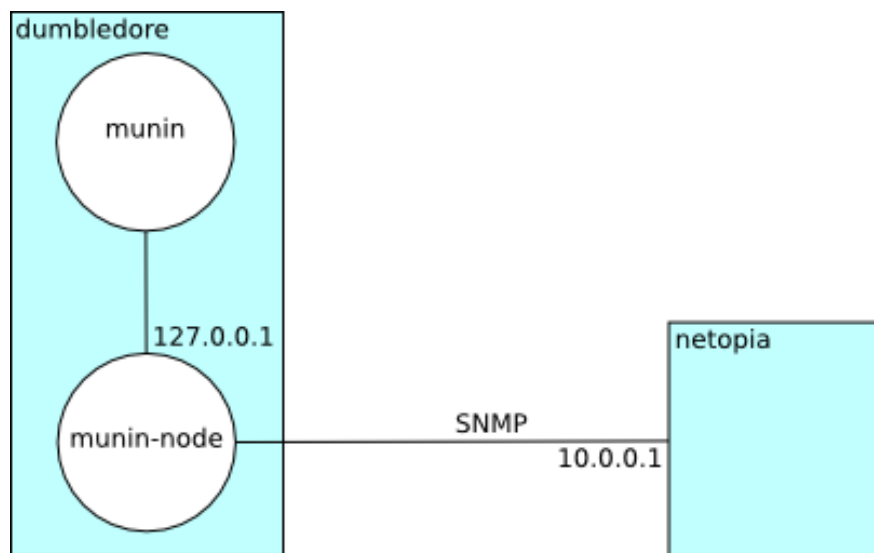
See *Using SNMP plugins* for information about configuring SNMP plugins.

Using SNMP plugins

(Cribbed from an e-mail written by Rune Nordbøe Skillingstad)

The easy way to configure SNMP plugins in Munin is to use *munin-node-configure*.

In this setup, both munin and munin-node runs on the server “dumbledore”, and we also want to monitor the router “netopia” using SNMP plugins. The setup is shown below:



On the node you want to use as an SNMP gateway (“dumbledore”), run the configure script against your SNMP-enabled device (“netopia”).

```
dumbledore:~# munin-node-configure --shell --snmp netopia
ln -s /usr/share/munin/plugins/snmp__if_ /etc/munin/plugins/snmp_netopia_if_1
ln -s /usr/share/munin/plugins/snmp__if_err_ /etc/munin/plugins/snmp_netopia_if_
↪err_1
```

Note that *munin-node-configure* also accepts other switches, namely `--snmpversion` and `--snmpcommunity`:

```
munin-node-configure --shell --snmp <host|cidr> --snmpversion <ver> --
↪snmpcommunity <comm>
```

This process will check each plugin in your Munin plugin directory for the *magic markers* `family=snmpauto` and `capabilities=snmpconf`, and then run each of these plugins against the given host or CIDR network.

Cut and paste the suggested `ln` commands and restart your node.

The node will then present multiple virtual nodes:

```
dumbledore:~# telnet localhost 4949
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
# munin node at dumbledore
nodes
netopia
dumbledore
.
list netopia
```

```
snmp_netopia_if_1
snmp_netopia_if_err_1
```

On your master server (where you gather the information into rrd files) you add this virtual node to your *munin.conf* (example contains both real node and the virtual one – both with the same address line)

```
[dumbledore]
  address 127.0.0.1
  use_node_name yes

[netopia]
  address 127.0.0.1
  use_node_name no
```

Next time *munin-cron* runs, the virtual node should start showing up in your Munin website.

You cannot easily set the SNMP community if it is different from the default `public`.

Recommended solution:

```
# munin-node-configure --snmp your.host.domain.tld --snmpcommunity "seacrat_
↪community"
```

Note that the community strings are not automatically saved anywhere. You will have to store them yourself to a file under `/etc/munin/plugin-conf.d/`. This file should not be world readable.

Example file `/etc/munin/plugin-conf.d/snmp_communities`:

```
[snmp_netopia_*]
env.community seacrat community

[snmp_some.other.device_*]
env.community frnpeng pbzzhavgl
```

Always provide your community name unquoted. In fact, if you do quote it, it will tread the quote as part of the community name, and that will usually not work. Just note that any prefix or trailing white space is stripped out, so you **cannot** currently configure a community name with a prefix or trailing white space.

To probe SNMP hosts over IPv6, use `--snmpdomain udp6` with `munin-node-configure`. To have the SNMP plugins poll devices over IPv6, set the `domain` environment variable to `udp6` in the plugin configuration file. Other transports are available; see the `Net::SNMP` perldoc for more options.

Let Munin croak alarm

As of Munin 1.2 there is a generic interface for sending warnings and errors from Munin. If a Munin plugin discovers that a plugin has a data source breaching its defined limits, Munin is able to alert the administrator either through simple command line invocations or through a monitoring system like Nagios or Icinga.

Note that if the receiving system can cope with only a limited number of messages at the time, the configuration directive `contact.contact.max_messages` may be useful.

When sending alerts, you might find good use in the *Munin alert variables*.

Note: Alerts not working? For some versions 1.4 and less, note that having [more than one contact defined](#) can cause `munin-limits` to hang.

Sending alerts through Nagios

How to set up Nagios and Munin to communicate has been thoroughly described in *Munin and Nagios*.

Alerts send by local system tools

Email Alert

To send email alerts directly from Munin use a command such as this:

```
contact.email.command mail -s "Munin-notification for ${var:group} :: ${var:host}"  
→your@email.address.here
```

Syslog Alert

To send syslog message with priority use a command such as this:

```
contact.syslog.command logger -p user.crit -t "Munin-Alert"
```

Alerts to or through external scripts

To run a script (in this example, 'script') from Munin use a command such as this in your munin.conf.

Make sure that:

1. There is NO space between the '>' and the first 'script'
2. 'script' is listed twice and
3. The munin user can find the script – by either using an absolute path or putting the script somewhere on the PATH – and has permission to execute the script.

```
contact.person.command >script script
```

This syntax also will work (this time, it doesn't matter if there is a space between 'l' and the first 'script' ... otherwise, all the above recommendations apply):

```
contact.person.command | script script
```

Either of the above will pipe all of Munin's warning/critical output to the specified script. Below is an example script to handle this input and write it to a file:

```
#!/usr/bin/env ruby  
  
File.open('/tmp/munin_alerts.log', 'a') do |f| #append  
  f.puts Time.now  
  for line in $stdin  
    f.puts line  
  end  
end
```

The alerts getting piped into your script will look something like this:

```
localhost :: localdomain :: Inode table usage  
CRITICALS: open inodes is 32046.00 (outside range [:6]).
```

Syntax of warning and critical

The `plugin.warning` and `plugin.critical` values supplied by a plugin can be overwritten by the Munin master configuration in `munin.conf`.

Note that the warning/critical exception is raised only if the value is outside the defined value. E.g. `foo.warning 100:200` will raise a warning only if the value is outside the range of 100 to 200.

Reformatting the output message

You can redefine the format of the output message by setting *Global Directive* `contact.<something>.text` in `munin.conf` using *Munin alert variables*.

Something like:

```
contact.pipevia.command | /path/to/script /path/to/script \
  --cmdlineargs="${var:group} ${var:host} ${var:graph_category} '${var:graph_
  ↵title}'"

contact.pipevia.always_send warning critical

contact.pipevia.text <munin group="${var:group}" host="${var:host}"\
  graph_category="${var:graph_category}" graph_title="${var:graph_title}" >\
  ${loop< >:wfields <warning label="${var:label}" value="${var:value}"\
  w="${var:wrange}" c="${var:crange}" extra="${var:extinfo}" /> }\
  ${loop< >:cfields <critical label="${var:label}" value="${var:value}"\
  w="${var:wrange}" c="${var:crange}" extra="${var:extinfo}" /> }\
  ${loop< >:ufields <unknown label="${var:label}" value="${var:value}"\
  w="${var:wrange}" c="${var:crange}" extra="${var:extinfo}" /> }\
</munin>
```

Calls the script with the command line arguments (as a python list):

```
['/path/to/script', '/path/to/script', '--cmdlineargs="example.com', 'test.example.
  ↵com', 'disk', 'Disk usage in percent', '']
```

and the input sent to the script is (whitespace added to break long line):

```
'<munin group="example.com" host="test.example.com" graph_category="disk" graph_
  ↵title="Disk usage in percent" >
  <critical label="/home" value="98.41" w=":92" c=":98" extra="" />
</munin> '
```

(need for the second `/path/to/script` may vary, but this document says it is required)

If something goes wrong:

- check the log file for `munin-limits.log`.
- remember this script will run as the same user as the cron job that starts `munin-cron`.

For more examples see section *Example usage* below.

Munin Alert Variables

When using Munin's built-in alert mechanisms, lots of variables are available. Generally, all directives recognized in the *configuration protocol* and in `munin.conf` are available as `${var:directive}`. We list some frequently used in the following section.

Group or host or plugin related variables

These are directly available.

Variable group

Syntax `${var:group}`

Reference Group name as declared in `munin.conf`.

Variable host

Syntax `${var:host}`

Reference Host name as declared in munin.conf.

Variable graph_title

Syntax `${var:graph_title}`

Reference Plugin's title as declared via config protocol or set in munin.conf.

Variable plugin

Syntax `${var:plugin}`

Reference Plugin's name as declared via config protocol or set in munin.conf.

Variable graph_category

Syntax `${var:graph_category}`

Reference Plugin's category as declared via config protocol or set in munin.conf.

Data source related variables

The below table lists some variables related to the data fields in a plugin. To extract these, they must be iterated over, even if there is only one field. Iteration follows the syntax defined in the Perl module `Text::Balanced` (sample below the table).

Variable {fieldname}.label

Syntax `${var:label}`

Reference Label of the data field as declared via plugin's config protocol or set in munin.conf.

Variable {fieldname}.value

Syntax `${var:value}`

Reference Value of the data field as delivered by data fetch

Variable {fieldname}.extinfo

Syntax `${var:extinfo}`

Reference Extended info of the field, if declared via plugin's config protocol or set in munin.conf.

Variable {fieldname}.warning

Syntax `${var:wrange}`

Reference Numeric range for warning alerts of the field, if declared via plugin's config protocol or set in munin.conf.

Variable {fieldname}.critical**Syntax** `${var:crange}`**Reference** Numeric range for critical alerts of the field, if declared via plugin's config protocol or set in `munin.conf`.**Variable wfields****Syntax** `${var:wfields}`**Reference** Space separated list of fieldnames with a value outside the warning range as detected by `munin-limit`.**Variable cfields****Syntax** `${var:cfields}`**Reference** Space separated list of fieldnames with a value outside the critical range as detected by `munin-limit`.**Variable ufields****Syntax** `${var:ufields}`**Reference** Space separated list of fieldnames with an unknown value as detected by `munin-limit`.

How variables are expanded

The `${var:value}` variables get the correct values from `munin-limits` prior to expansion of the variable.

Then, the `${var:*range}` variables are set from `{fieldname}.warning` and `{fieldname}.critical`.

Based on those, `{fieldname}.label` occurrences where warning or critical levels are breached or unknown are summarized into the `${var:*fields}` variables.

Example usage

Note that the sample command lines are wrapped for readability.

Example 1, iterating through warnings and criticals

```
contact.mail.command mail -s "[${var:group};${var:host}] -> ${var:graph_title} ->
                               warnings: ${loop<,>:wfields} ${var:label}=${var:value}
↵ } /
                               criticals: ${loop<,>:cfields} ${var:label}=${
↵ ${var:value}}" me@example.com
```

This stanza results in an e-mail with a subject like this:

```
[example.com;foo] -> HDD temperature -> warnings: sde=29.00,sda=26.00,sdc=25.00,
↵ sdd=26.00,sdb=26.05 / criticals:
```

Note that there are no breaches of critical level temperatures, only of warning level temperatures.

Example 2, reading `${var:wfields}`, `${var:cfields}` and `${var:ufields}` directly

```
contact.mail.command mail -s "[${var:group};${var:host}] -> ${var:graph_title} ->
warnings: ${var:wfields} /
criticals: ${var:cfields} /
unknowns: ${var:ufields}" me@example.com
```

The result of this is the following:

```
[example.com;foo] -> HDD temperature -> warnings: sde sda sdc sdd sdb / criticals:
->/ unknowns:
```

Iteration using Text::Balanced

The Text::Balanced iteration syntax used in munin-limits is as follows (extra spaces added for readability):

```
${ loop < join character > : list of words ${var:label} = ${var:value} }
```

Given a space separated list of words “a b c”, and the join character “,” (comma), the output from the above will equal

```
a.label = a.value,b.label = b.value,c.label = c.value
```

in which the label and value variables will be substituted by their Munin values.

Please consult the [Text::Balanced](#) documentation for more details.

Munin and Nagios

Munin integrates perfectly with Nagios. There are, however, a few things of which to take notice. This article shows example configurations and explains the communication between the systems.

Setting up Nagios passive checks

Receiving messages in Nagios

First you need a way for Nagios to accept messages from Munin. Nagios has exactly such a thing, namely the NSCA which is documented here: [NSCA](#).

NSCA consists of a client (a binary usually named `send_nasca` and a server usually run from `inetd`. We recommend that you enable encryption on NSCA communication.

You also need to configure Nagios to accept messages via NSCA. NSCA is, unfortunately, not very well documented in Nagios’ official documentation. We’ll cover writing the needed service check configuration further down in this document.

Configuring Nagios

In the main config file, make sure that the `command_file` directive is set and that it works. See [External Command File](#) for details.

Below is a sample extract from `nagios.cfg`:

```
command_file=/var/run/nagios/nagios.cmd
```

The `/var/run/nagios` directory is owned by the user `nagios` runs as. The `nagios.cmd` is a named pipe on which Nagios accepts external input.

Configuring NSCA, server side

NSCA is run through some kind of (x)inetd.

Using inetd

the line below enables NSCA listening on port 5667:

```
5667          stream tcp      nowait  nagios  /usr/sbin/tcpd  /usr/sbin/nsca -c /
↳etc/nsca.cfg --inetd
```

Using xinetd

the lines below enables NSCA listening on port 5667, allowing connections only from the local host:

```
# description: NSCA (Nagios Service Check Acceptor)
service nsca
{
  flags          = REUSE
  type          = UNLISTED
  port          = 5667
  socket_type   = stream
  wait         = no

  server        = /usr/sbin/nsca
  server_args   = -c /etc/nagios/nsca.cfg --inetd
  user          = nagios
  group         = nagios

  log_on_failure += USERID

  only_from     = 127.0.0.1
}
```

Common

The file `/etc/nsca.cfg` defines how NSCA behaves. Check in particular the `nsca_user` and `command_file` directives, these should correspond to the file permissions and the location of the named pipe described in `nagios.cfg`.

```
nsca_user=nagios
command_file=/var/run/nagios/nagios.cmd
```

Configuring NSCA, client side

The NSCA client is a binary that submits to an NSCA server whatever it received as arguments. Its behaviour is controlled by the file `/etc/send_nsca.cfg`, which mainly controls encryption.

You should now be able to test the communication between the NSCA client and the NSCA server, and consequently whether Nagios picks up the message. NSCA requires a defined format for messages. For service checks, it's like this:

```
<host_name> [tab] <svc_description> [tab] <return_code> [tab] <plugin_output> [newline]
```

Below is shown how to test NSCA.

```
$ echo -e "foo.example.com\ttest\t0\t0" | /usr/sbin/send_nsga -H localhost -c /etc/
↪send_nsga.cfg
1 data packet(s) sent to host successfully.
```

This caused the following to appear in `/var/log/nagios/nagios.log`:

```
[1159868622] Warning: Message queue contained results for service 'test' on host
↪'foo.example.com'. The service could not be found!
```

Sending messages from Munin

Messages are sent by *munin-limits* based on the state of a monitored data source: OK, Warning, Critical and Unkown (O/W/C/U).

Configuring `munin.conf`

Nagios uses the above mentioned `send_nsga` binary to send messages to Nagios. In `/etc/munin/munin.conf`, enter this:

```
contacts nagios
contact.nagios.command /usr/bin/send_nsga -H your.nagios-host.here -c /etc/send_
↪nsga.cfg
```

Note: Be aware that the `-H` switch to `send_nsga` appeared sometime after `send_nsga` version 2.1. Always check `send_nsga --help`!

Configuring Munin plugins

Lots of Munin plugins have (hopefully reasonable) values for Warning and Critical levels. To set or override these, you can change the values in *munin.conf*.

Configuring Nagios services

Now Nagios needs to recognize the messages from Munin as messages about services it monitors. To accomplish this, every message Munin sends to Nagios requires a matching (passive) service defined or Nagios will ignore the message (but it will log that something tried).

A passive service is defined through these directives in the proper Nagios configuration file:

```
active_checks_enabled      0
passive_checks_enabled     1
```

A working solution is to create a template for passive services, like the one below:

```
define service {
    name                passive-service
    active_checks_enabled 0
    passive_checks_enabled 1
    parallelize_check    1
    notifications_enabled 1
    event_handler_enabled 1
    register             0
    is_volatile          1
}
```

When the template is registered, each Munin plugin should be registered as per below:

```
define service {
    use                passive-service
    host_name          foo
    service_description bar
    check_period       24x7
    max_check_attempts 3
    normal_check_interval 3
    retry_check_interval 1
    contact_groups     linux-admins
    notification_interval 120
    notification_period 24x7
    notification_options w,u,c,r
    check_command       check_dummy!0
}
```

Notes

- `host_name` is either the FQDN of the `host_name` registered to the Nagios plugin, or the host alias corresponding to Munin's `notify_alias` directive. The `host_name` must be registered as a host in Nagios.
- `service_description` must correspond to the plugin's name, and for Nagios to be happy it shouldn't have any special characters. If you'd like to change the service description from Munin, use `notify_alias` on the data source. Available in Munin-1.2.5 and later.

A working example is shown below:

```
[foo.example.com]
    address foo.example.com
    df.notify_alias Filesystem usage
    # The above changes from Munin's default "Filesystem usage (in %)"
```

What characters are allowed in a Nagios service definition?

See Nagios docs on [Illegal Object Name Characters](#)

`service_description`: This directive is used to define the description of the service, which may contain spaces, dashes, and colons (semicolons, apostrophes, and quotation marks should be avoided). No two services associated with the same host can have the same description. Services are uniquely identified with their `host_name` and `service_description` directives.

Note: This means that lots of Munin plugins will not be accepted by Nagios. This limitation impacts every plugin with special characters in them, e.g. `'(, ')`, and `'%`. Workarounds are described in [ticket #34](#) and the bug has been fixed in the Munin code in changeset 1081.

Alternatively you can use `check_munin.pl` to gather fresh data from nagios instead of `check_dummy`.

Sample munin.conf

To illustrate, a (familiar) sample `munin.conf` configuration file shows the usage:

```
contact.nagios.command /usr/local/nagios/bin/send_nsca nagioshost.example.com -c /
↳usr/local/nagios/etc/send_nsca.cfg -to 60

contacts no                # Disables warning on a system-wide basis.

[example.com;]
    contacts nagios        # Enables warning through the "nagios" contact for
↳the group example.com
```

```
[foo.example.com]
  address localhost
  contacts no # Disables warning for all plugins on the host foo.
↳example.com.

[example.com;bar.example.com]
  address bar.example.com
  df.contacts no # Disables warning on the df plugin only.
  df.notify_alias Disk usage # Uses the title "Disk usage" when sending warnings.
↳through munin-limits
# Useful if the receiving end does not accept all
↳kinds of characters
# NB: Only available in Munin-1.2.5 or with the
↳patch described in ticket 34.
```

Setting up Nagios active checks

Use `check_munin.p` to get data from munin-node directly into nagios and then use it as a regular check plugin. Basically munin-node become a kind of snmp agent with a lot of preconfigured plugins.

Advanced Features

Introduction

Alias

Aggregate graphs

Now and again, the need to combine one or more graphs shows up.

Munin facilitates this through the use of *fieldname.sum* and/or *fieldname.stack*.

See *Graph aggregation stacking example* and *Graph aggregation by example* for details.

Supersampling

Conclusion

Extraordinary Usage

Introduction

Most of us use Munin to monitor network traffic and service activities in IT environment, but as it can monitor anything that is expressible in numbers. There are a lot more operational scenarios one can think of. Here we collect reports and links about these unusual use cases ;-)

Examples

- Monitoring the temperature values of a Ökofen Pellematic
- How to write a plugin to monitor departures from Munich airport

Monitoring the “unreachable” hosts

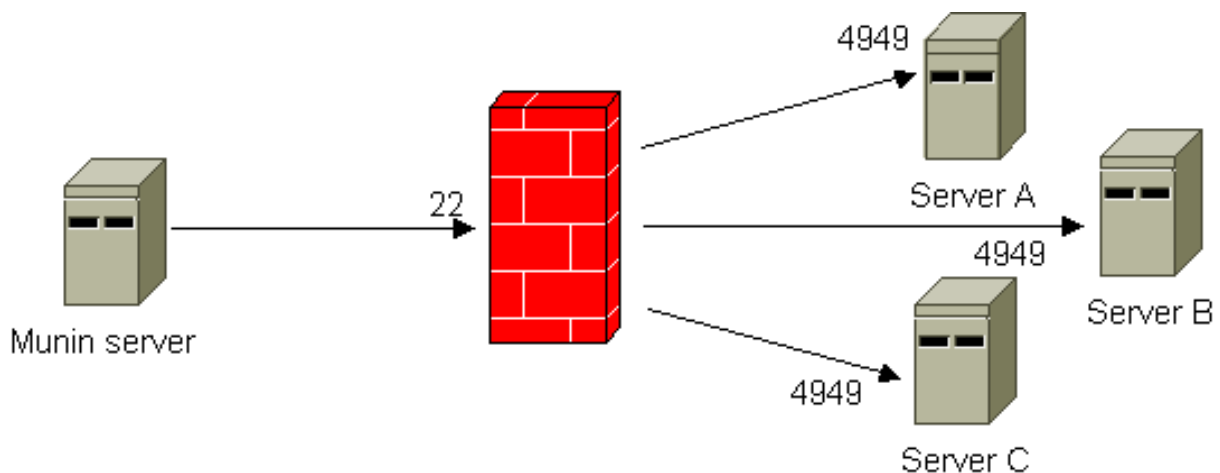
There are a number of situations where you’d like to run munin-node on hosts not directly available to the Munin server. This article describes a few scenarios and different alternatives to set up monitoring. Monitoring hosts behind a non-routing server.

In this scenario, a *nix server sits between the Munin server and one or more Munin nodes. The server in-between reaches both the Munin server and the Munin node, but the Munin server does not reach the Munin node or vice versa.

To enable for Munin monitoring, there are several approaches, but mainly either using SSH tunneling or “bouncing” via the in-between server.

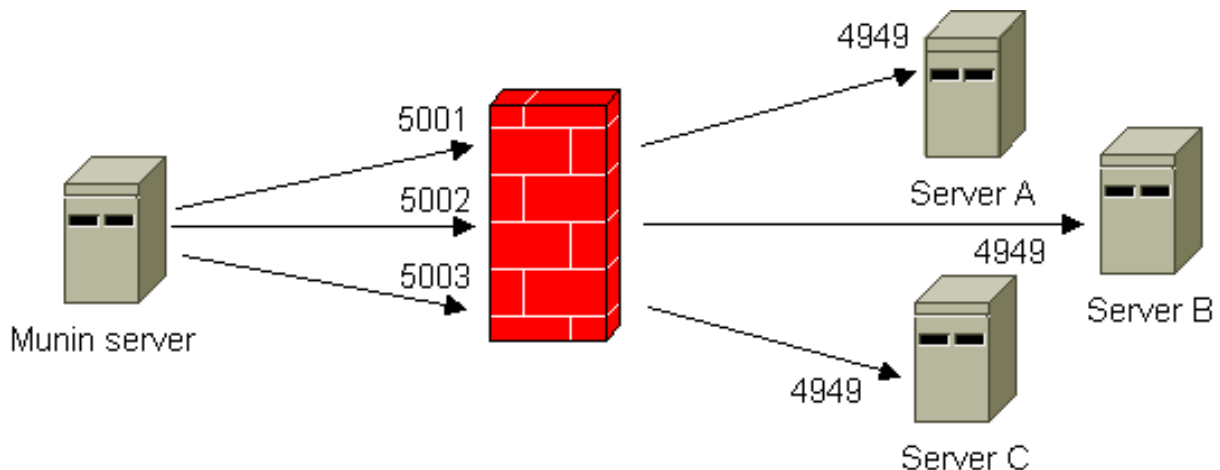
SSH tunneling

The illustration below shows the principle. By using SSH tunneling only one SSH connection is required, even if you need to reach several hosts on “the other side”. The Munin server listens to different ports on the localhost interface. A [configuration example](#) is included. Note that there is also a [FAQ entry on using SSH](#) that contains very useful information.



Bouncing

This workaround uses netcat and inetd/xinetd to forward the queries from the Munin server. All incoming connections to defined ports are automatically forwarded to the Munin node using netcat.

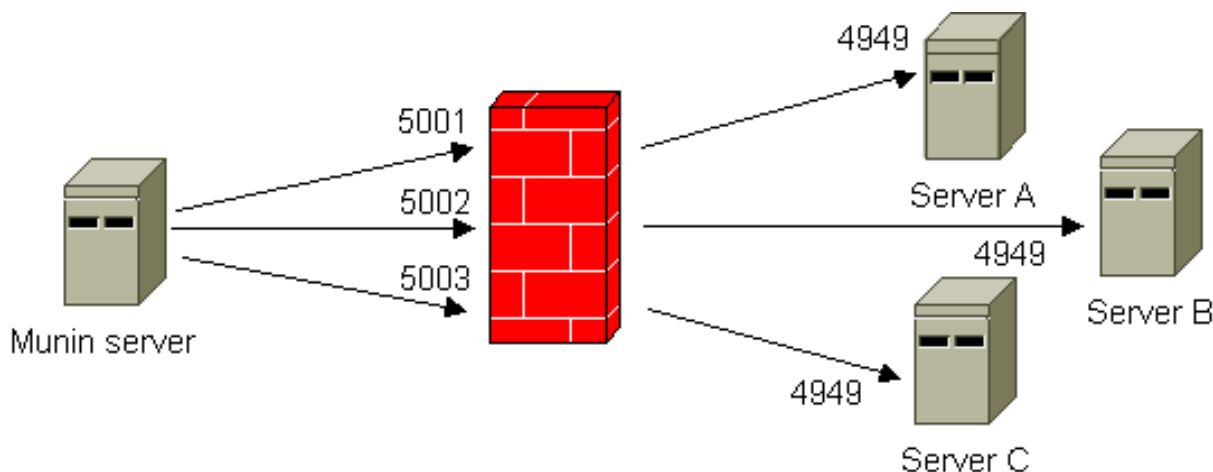


Behind a NAT device

Monitoring hosts behind a NAT device (e.g. DSL router or firewall)

If you have one or more Munin nodes on the “inside” of a NAT device, port forwarding is probably the easiest way to do it. Configuring port forwarding on all kinds of network units and firewall flavours is way beyond the scope of the Munin documentation, but the illustration below show the principle. A [sample configuration](#) including `munin.conf` is found [here](#).

Note that if the NAT device is a *nix system, you may also use the two approaches described above.



Troubleshooting

This page lists some general troubleshooting strategies and methods for Munin.

Check node agent

Is the `munin-node` process (daemon) running on the host you want to monitor?

Did you restart the `munin-node` process after you made changes to its configuration?

Check connectivity

The examples show a `munin-node` agent running on `127.0.0.1`; replace it with your node address.

Note: You can use `netcat` to port 4949.

Using `telnet` was the previous recommended way as it was a fairly standard install. We don't recommend it anymore since `netcat` is now almost as ubiquitous as `telnet` and it offers a real native TCP connection, whereas `telnet` does not. Note that using `socket` also works perfectly, but it is not as mainstream.

Does the `munin-node` agent allow connections from your munin master?

Here we try to connect manually to the `munin-node` that runs on the Munin master host. It can be reached via IP address `127.0.0.1` or hostname `localhost` and port 4949.

Output of a `netcat` session should be something like this:

```
# nc localhost 4949
Trying 127.0.0.1...
Connected to localhost.
```

```
Escape character is '^]'.
# munin node at [your hostname]
```

Does the above output give the same hostname that should be expected upon configuration in *munin.conf*?

Note: If you have a fully qualified domain name (FQDN) in *munin-node.conf*, the host you're monitoring has to identify itself with FQDN as well.

E.g. if the masters node tree has the following entry:

```
[foo.example.com]
  address foo.example.com
```

...then a netcat session to the node should give you the following output:

```
# munin node at foo.example.com
```

Note: If the connection test fails, check the *allow directive* in *munin-node.conf* and make sure any firewalls allow contact on destination port 4949.

Debugging Plugins

Which plugins are enabled on the node?

Does *munin-node* recognize any plugins? Try issuing the command `list` (being connected to the agent) and a (long) list of plugins should show.

```
# nc localhost 4949
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
# munin node at foo.example.com
list
open_inodes irqstats if_eth0 df uptime [...]
```

Check a particular plugin

Check on agent host

Note: All the commands here need to be run as user `root`. A common method of becoming `root` is via the `sudo` command, but refer to your local documentation for a more specific instruction.

Restart *munin-node*, as it only reads the plugin list upon start. (Good to test a plugin with *munin-run*, without enabling it right away.)

```
/etc/init.d/munin-node restart
```

Call *munin-run* on the monitored host to see whether the plugin runs through .

Try with and without the `config` plugin argument. Both runs should not emit any error message.

Note: You can also use the `--debug` flag, as it shows if the configuration file is correctly parsed, mostly for UID & environment variables.

Regular run:

```
# munin-run df
_dev_hda1.value 83
```

Config run:

```
# munin-run df config
graph_title Filesystem usage (in %)
graph_args --upper-limit 100 -l 0
graph_vlabel %
graph_category disk
graph_info This graph shows disk usage on the machine.
_dev_hda1.label /
_dev_hda1.info / (ext3) -> /dev/hda1
_dev_hda1.warning 92
_dev_hda1.critical 98
```

Check from Munin master

Does the plugin run through *munin-node*, with and without config?

Regular run:

```
# nc foo.example.com 4949
Trying foo.example.com...
Connected to foo.example.com.
Escape character is '^]'.
# munin node at foo.example.com
fetch df
_dev_hda1.value 83
[...]
.
```

With config:

```
# nc foo.example.com 4949
Trying foo.example.com...
Connected to foo.example.com.
Escape character is '^]'.
# munin node at foo.example.com
config df
graph_title Filesystem usage (in %)
graph_args --upper-limit 100 -l 0
graph_vlabel %
graph_category disk
graph_info This graph shows disk usage on the machine.
_dev_hda1.label /boot
_dev_hda1.info /boot (ext3) -> /dev/hda1
_dev_hda1.warning 92
_dev_hda1.critical 98
[...]
.
```

If the plugin works for `munin-run` but not through `netcat`, you might have a `$PATH` problem.

Note: Set `{{env.PATH}}` for the plugin in the plugin's environment file.

Check Munin Master

Do the directories specified by `dbdir`, `htmldir`, `logdir` and `rundir` defined in `munin.conf` have the correct permissions? (If you first run munin as root, maybe they're not readable/writable by the user that runs the cron job)

Is `munin-cron` established as a cron controlled process, run as the Munin user?

Does the output when running `munin-update` as the Munin user on the server node show any errors?

Try running `munin-cron --debug > /tmp/munin-cron.debug` and check the output file `/tmp/munin-cron.debug`.

Check data collection

This step will tell you whether `munin-update` (the master) is able to communicate with `munin-node` (the agent).

Run `munin-update` as user `munin` on the Munin master machine.

```
# su -s /bin/bash munin
$ /usr/share/munin/munin-update --debug --nofork --host foo.example.com --service_
↪df
```

You should get a line like this:

```
Aug 11 22:39:51 - [6846] Updating /var/lib/munin/example.com/foo.example.com-df-_
↪dev_hda1-g.rrd with 57
```

After this, replace `df` with the service you want to check, such as `hddtemp_smartctl`.

If one of these steps does not work, something is probably wrong with the plugin or how `munin-node` talks to the plugin.

1. Does the plugin run when executed directly? If it runs when executed as root and not through `munin-run` (as described above), the plugin has a permission problem. See [this article on environment files](#).
2. Does the plugin output contain too few, too many and/or illegal characters?
3. Does Munin (`munin-cron` and its children) write values into RRD files? Hint: `rrdtool fetch [rrd file] AVERAGE`
4. Does the plugin use legal field names? See [Notes on Field names](#).
5. In case you [loan data](#) from other graphs, check that the `fieldname.type` is set properly. See [Munin file names](#) for a quick reference on what any error messages in the logs might indicate.

Frequent Incidents

SELinux blocks Munin plugins

- See [the documentation start page](#) for links to SELinux rules for Munin.

RRD files are filled with 0

although `munin-node` seems to show sane values.

- The plugin's output shows GAUGE values, but were declared as COUNTER or DERIVE in the plugin's config.

Note: GAUGE is the default data type in Munin! Any other data type for a field must be explicitly declared.

RRD files are filled with NaN

although munin-node seems to show sane values.

- Check that there are no invalid characters in the plugin's output.
- For new plugins let munin gather data for about 20 minutes and things will unwrinkle

munin-node won't give any data

although it is configured properly.

- Check that there is a `.value` directive for every of the plugin's field names (yes, I managed to forget that recently).

munin-node only temporary returns valid data

- Check that no race conditions occur. A typical race condition is updating a file with crontab while the plugin is trying to read the file.

The graphs are empty

- The plugin's output shows GAUGE values, but were declared as COUNTER or DERIVE in the plugin's config. (GAUGE is default data type in Munin)
- The files to be updated by Munin are owned by root or another user account
- The local user browser cache may be corrupt, especially if "most" graphs are displayed correctly and "some" graphs are blank. In Firefox (or your browser of choice) go to tools and clear recent history, then check to see if the graphs are now properly displayed.

Other mumbo-jumbo

- Run the different stages in *munin-cron* manually, using `--debug`, `--nofork`, something like this:

```
# su - munin -c "/usr/lib/munin/munin-update \  
--debug --nofork \  
--host foo.example.com \  
--service df"
```

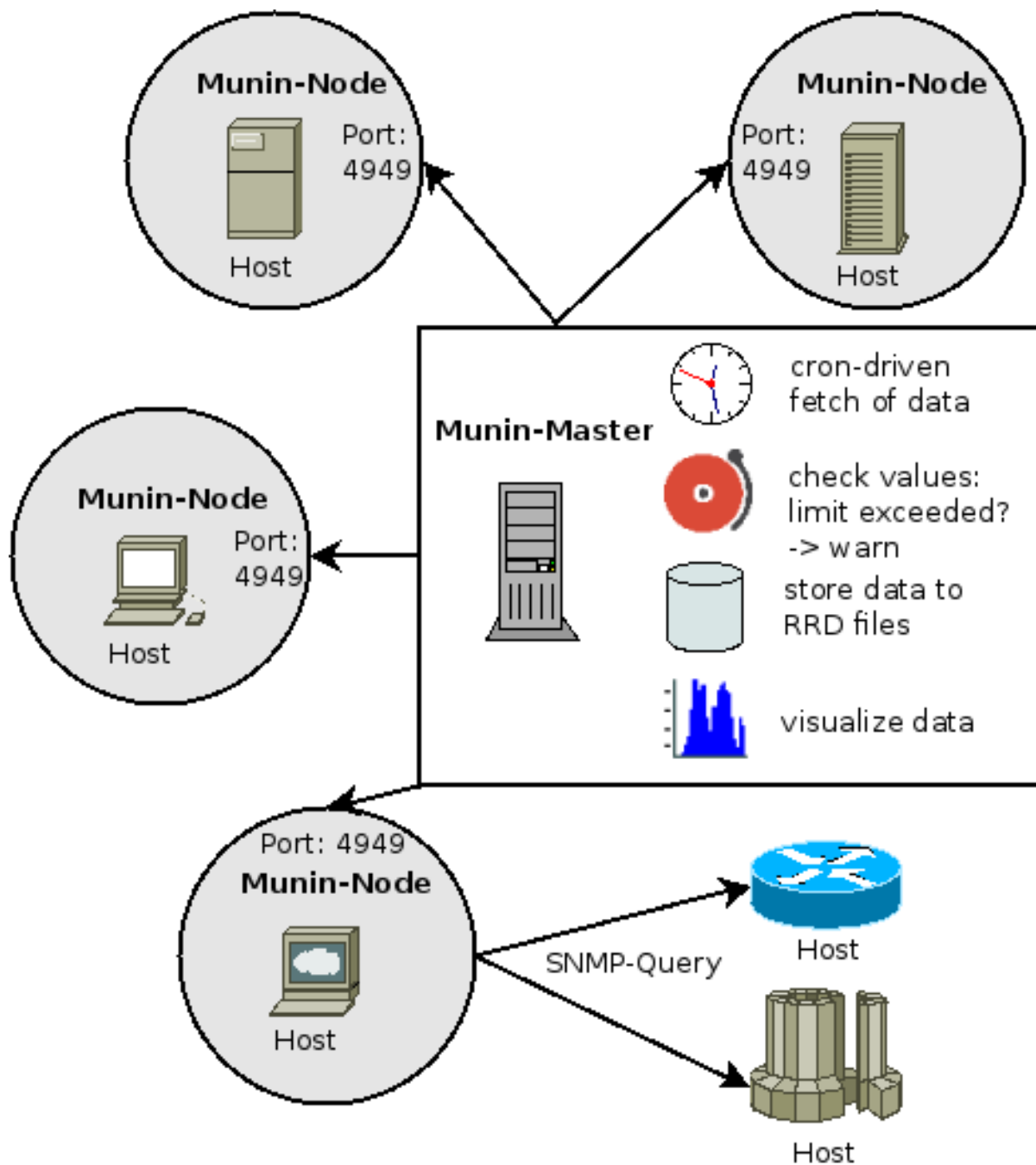
See also

- [No Graph FAQ](#)
- [Upgrade notes](#)

Munin's Architecture

Overview

Munin has a master-nodes architecture.



Components

Here we describe the components of Munin. On page *Protocols* we talk about the rules for interaction between them.

Munin-Master

The master is responsible for all central Munin-related tasks.

It regularly connects to the various nodes, and then *synchronously* asks for the various metrics configuration and values and stores the data in *RRD* <http://oss.oetiker.ch/rrdtool/> files.

On the fly the values are checked against limits (that you may set) and the Munin-Master will croak, if values go above or below the given thresholds.

Here we also generate the graphs, as this is a heavy task that needs some resources. Recent versions of Munin use cgi-graphing to generate graphs only when the user want's to see them.

The Munin master

For an overview see *Architectural Fundamentals*

Role

The munin master is responsible for gathering data from munin nodes. It stores this data in RRD¹, files, and graphs them on request. It also checks whether the fetched values fell below or go over specific thresholds (warning, critical) and will send alerts if this happens and the administrator configured it to do so.

Components

The following components are part of munin-master:

- *munin-cron* runs *munin-limits* and *munin-update*.
- *munin-update* is run by *munin-cron*. It is the munin data collector, and it fetches data from *munin nodes*, which is then stored in RRD files.
- *munin-limits* is run by *munin-cron*. It notifies any configured contacts if a value moves between “ok”, “warn” or “crit”. Munin is commonly used in combination with Nagios, which is then configured as a contact.

Configuration

The munin master has its primary configuration file at */etc/munin/munin.conf*.

Fetching values

Data exchange between master and node

Connect to the node

```
# telnet localhost 4949
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
# munin node at foo.example.com
help
# Unknown command. Try cap, list, nodes, config, fetch, version or quit
.
Connection closed by foreign host.
```

¹ RRDtool (acronym for round-robin database tool) aims to handle time-series data like network bandwidth, temperatures, CPU load, etc. The data are stored in a round-robin database (circular buffer), thus the system storage footprint remains constant over time. Source Wikipedia: <http://en.wikipedia.org/wiki/RRDtool>

Node commands

The *Munin node* daemon will understand and answer to the following inquiries.

cap Lists the capabilities of the node, e.g. `multigraph dirtyconfig`

list [node] Simply lists items available for gathering for this host. E.g. `load, cpu, memory, df, et alia`. If no *host* is given, default to host that runs the munin-node.

nodes Lists hosts available on this node.

config <query-item> Shows the plugins configuration items. See the config protocol for a full description.

fetch <query-item> Fetches values

version Print version string

quit Close the connection. Also possible to use a point ”.”.

capabilities

The master can exchange capabilities with the node using the “cap” keyword, and a list of capabilities. For each capability supported by both the master and node, the node sets an environment variable “MUNIN_CAP_CAPABILITY”, where CAPABILITY is the capability in upper case.

Capabilities used so far by munin node and master:

dirtyconfig

If the node and master support the “dirtyconfig” capability, the `MUNIN_CAP_DIRTYCONFIG` environment variable is set for all plugins.

This allows plugin to send config and data when the master asks for “config” for this plugin, reducing the round trip time.

multigraph

If the node and master support the “multigraph” capability, the `MUNIN_CAP_MULTIGRAPH` environment variable is set for all plugins.

This allows plugins to use the “multigraph” format.

See also *Protocol extension: multiple graphs from one plugin*

spoolfetch

If the node and master support the “spoolfetch” capability, the master can use the “spoolfetch” command to retrieve a spool of all plugin output since a given time.

This is used by *Asynchronous proxy node*.

Example outputs

config

```

> config load
< graph_args --title "Load average"
< load.label Load
< .
> config memory
< graph_args --title "Memory usage" --base 1024
< used.label Used
< used.draw AREA
< shared.label Shared
< shared.draw STACK
< buffers.label Buffers
< buffers.draw STACK
< cache.label Cache
< cache.draw STACK
< free.label Free
< free.draw STACK
< swap.label Swap
< swap.draw STACK

```

fetch

Fetches the current values.

Returned data fields:

```
<field>.value
```

Numeric value, or 'U'.

```

> fetch load
< load.value 0.42
< .
> fetch memory
< used.value 98422784
< shared.value 1058086912
< buffers.value 2912256
< cache.value 8593408
< free.value 235753472
< swap.value 85053440

```

Graphing Charts

Other documentation

Scaling the munin master with rrdcached

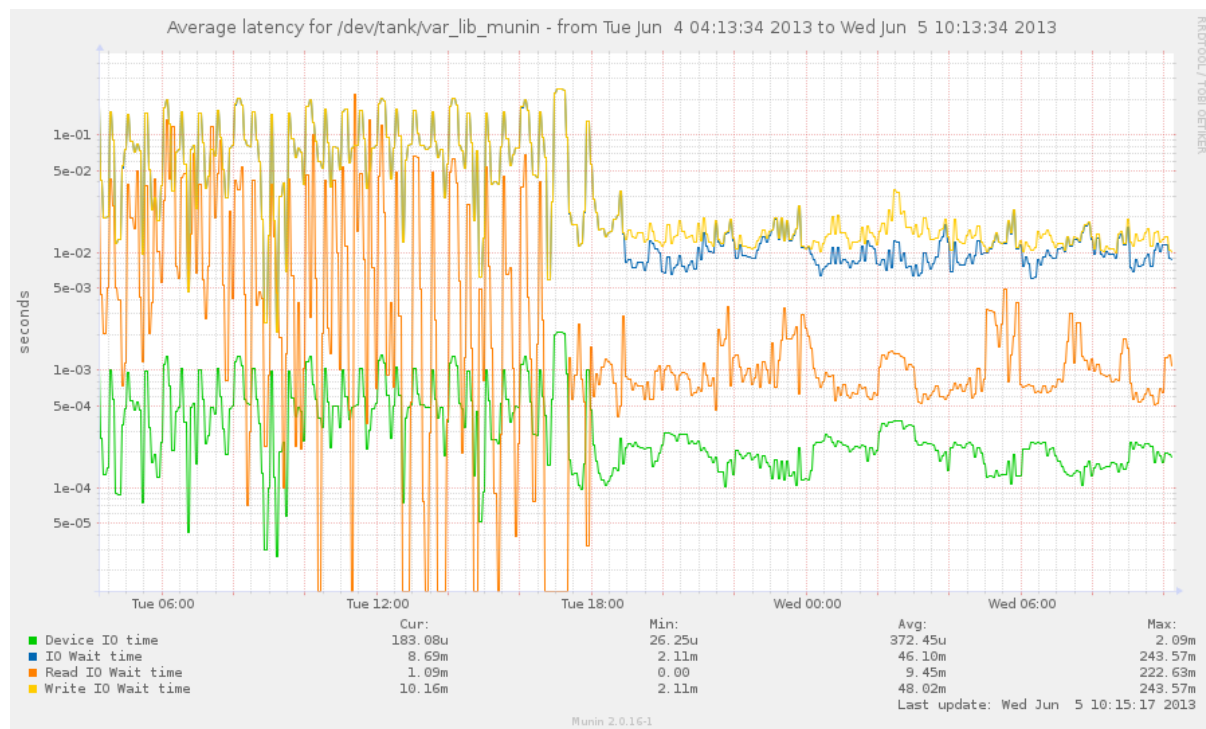
When the master grows big, and has a lot of nodes, there is a risk of disk IO becoming a bottleneck.

To reduce this disk IO, you can use the RRD Cache Daemon.

This will spool RRD changes in a queue, and flush changes on demand, and periodically. This will replace lots of random writes with a much smaller amount of sequential writes.

The effects on disk IO can be quite dramatic.

This example is a graph of a munin master with 400 nodes. Even with storage on mirrored SSDs, the effect of adding rrdcached is an immediate reduction in IO, and especially on wait times.



Configuring rrdcached

Parameters

RRDCached writes the spool data every 5 minutes by default. This is the same as the munin master. To have an effect, change the flushing intervals to allow more data to be spooled. Use the following parameters, and tune to your liking:

-w 1800	Wait 30 minutes before writing data
-z 1800	Delay writes by a random factor of up to 30 minutes (this should be equal to, or lower than, “-w”)
-f 3600	Flush all data every hour

Example

Create a directory for the rrdcached journal, and have the “munin” user own it. (in this example: /var/lib/munin/rrdcached-journal).

Set up a separate RRDCached instance, run by the munin user. The following command starts an RRDCached instance, and can be added to /etc/rc.local.

```
sudo -u munin /usr/bin/rrdcached \
  -p /run/munin/rrdcached.pid \
  -B -b /var/lib/munin/ \
  -F -j /var/lib/munin/rrdcached-journal/ \
  -m 0660 -l unix:/run/munin/rrdcached.sock \
  -w 1800 -z 1800 -f 3600
```

Note: While testing, add “-g” to the command line to prevent rrdcached from forking into the background.

The munin grapher also needs write access to this socket, in order for it to tell the RRDCached to flush data needed for graphing. If you run munin with CGI graphing, you will need to give the web server access. For a common setup, run the following command, as root, after starting rrdcached:


```
chgrp www-data /run/munin/rrdcached.sock
```

Recommended: If you have systemd or upstart installed, use the examples below.

- *Upstart configuration for rrdcached*
- *Systemd configuration for rrdcached*

Configuring munin to use rrdcached

To enable rrdcached on the munin master, you will need to set the “rrdcached_socket” line in /etc/munin/munin.conf

```
rrdcached_socket /run/munin/rrdcached.sock
```

Is it working?

If all goes well, you should see the following:

Munin logging

There should be no messages regarding rrdcached in /var/log/munin/munin-update.log.

On failure to connect, there will be log lines like:

```
2012/06/26 18:56:12 [WARN] RRDCached feature ignored: rrdcached socket not writable
```

... and you should then check for permissions problems.

RRDCached spool

The rrdcached spool file should be in /var/lib/munin/rrdcached-journal/, and it should grow for each run of munin-update until it hits the flush time. The file looks like:

```
/var/lib/munin/rrdcached-journal/rrd.journal.1340869388.141124
```

For a munin master with 200 nodes, this could well grow to 100MiB, depending on the number of plugins, and the spool file time parameters.

Munin-Node

The node is a small agent running on each monitored host. We can have agent-less monitoring but this is a special case that will be addressed later.

Note that an usual setup involves having a node running also on the master host, in order to munin to monitor itself.

The Munin node

Role

The munin node is installed on all monitored servers. It accepts connections from the munin master, and runs plugins on demand.

By default, it is started at boot time, listens on port 4949/TCP, accepts connections from the *munin master*, and runs *munin plugins* on demand.

Configuration

The configuration file is *munin-node.conf*.

Other documentation

Asynchronous proxy node

Context

We already discussed that *munin-update* is the fragile link in the munin architecture. A missed execution means that some data is lost.

The problem : updates are synchronous

In Munin 1.x, updates are synchronous : the epoch and value in each service are the ones *munin-update* retrieves each scheduled run.

The issue is that *munin-update* has to ask every service on every node every run for their values. Since the values are only computed when asked, *munin-update* has to wait quite some time for every value.

This design is very simple, it therefore enables munin to have the simplest plugins since they are completely stateless. While being the greatest strength of munin, it still puts a severe blow on scalability : more plugins and/or nodes means obviously a slower retrieval.

Evolving Solution : Parallel Fetching

1.4 addresses some of these scalability issues by implementing parallel fetching. It takes into account that the most of the execution time of *munin-update* is spent waiting for replies.

Note that there is the `max_processes` configuration parameter that control how many nodes in parallel *munin-update* can ask.

Now, the I/O part is becoming the next limiting factor, since updating many RRD files in parallel means massive and completely random I/O for the underlying *munin-master* OS.

Serializing & grouping the updates is possible with the *rrdcached* daemon from *rrdtool* starting at 1.4 and on-demand graphing. This looks very promising, but doesn't address the root defect in this design : a hard dependence of regular *munin-update* runs. And upon close analysis, we can see that 1.4 isn't ready for *rrdcached* as it asks for flush each run, in *munin-limits*.

2.0 : Stateful plugins (supersampling)

2.0 provides a way for plugins to be stateful. They might schedule their polling themselves, and then when *munin-update* runs, only emit collect already computed values. This way, a missed run isn't as dramatic as it is in the 1.x series, since data isn't lost. The data collection is also much faster because the real computing is done ahead of time. This behavior is called supersampling.

2.0 : Asynchronous proxy node

But changing plugins to be self-pollled is difficult and tedious. It even works against one of the real strengths of munin : having very simple, therefore stateless, plugins.

To address this concern, a proxy node was created. For 2.0 it takes the form of 2 tools : munin-asyncd and munin-async.

The proxy node in detail (munin-async)

Overview

These 2 processes form an asynchronous proxy between munin-update and munin-node. This avoids the need to change the plugins or upgrade munin-node on all nodes.

munin-asyncd should be installed on the same host than the proxied munin-node in order to avoid any network issue. It is the process that will poll regularly munin-node. The I/O issue of munin-update is here non-existent, since munin-async stores all the values by plainly appending them in text files without any processing. The files are defined as one per plugin, rotated per a timeframe.

These files are later read by munin-async client part that is typically accessed via ssh from munin-update. Here again no fancy processing is done, just plainly read back to the calling munin-update to be processed there. This way the overhead on the node is minimal.

The nice part is that the munin-async client does not need to run on the node, it can run on a completely different host. All it takes is to synchronize the spoolfetch dir. Sync can be periodic (think rsync) or real-time (think NFS).

In the same idea, the munin-asyncd can also be hosted elsewhere for disk-less nodes.

Specific update rates

Having one proxy per node enables a polling of all the services there with a plugin specific update rate.

To achieve this, munin-asyncd optionally forks into multiple processes, one for each plugin. This way each plugin is completely isolated from others. It can set its own update_rate, it is isolated from other plugins slowdowns, and it does even completely parallelize the information gathering.

SSH transport munin-async-client uses the new SSH native transport of 2.0. It permits a very simple install of the async proxy.

Notes

In 1.2 a service is the same as plugin, but since 1.4 and the introduction of multigraph, one plugin can provide multiple services. Think it as one service, one graph.

Installation

munin-async is a helper to poll regularly

The munin asynchronous proxy node (or “munin-async”) connects to the local node periodically, and spools the results.

When the munin master connects, all the data is available instantly.

munin-asyncd

The Munin async daemon starts at boot, and connects to the local munin-node periodically, like a *munin master* would. The results are stored the results in a spool, tagged with timestamp.

You can also use munin-asyncd to connect to several munin nodes. You will need to use one spooldir for each node you connect to. This enables you to set up a “fanout” setup, with one privileged node per site, and site-to-site communication being protected by ssh.

munin-async

The Munin async client is invoked by the connecting master, and reads from the munin-async spool using the “spoolfetch” command.

Example configuration

On the munin master

We use ssh encapsulated connections with munin async. In the *the munin master* configuration you need to configure a host with a “ssh://” address.

```
[random.example.org]
address ssh://munin-async@random.example.org
```

You will need to create an SSH key for the “munin” user, and distribute this to all nodes running munin-asyncd.

The ssh command and options can be customized in *munin.conf* with the ssh_command and ssh_options configuration options.

You can also specify the SSH port directly in the address, if a node is not reachable using the default SSH port (22):

```
[random2.example.org]
address ssh://munin-async@random2.example.org:2222
```

On the munin node

Configure your munin node to only listen on “127.0.0.1”.

You will also need to add the public key of the munin user to the authorized_keys file for this user.

- You must add a “command=” parameter to the key to run the command specified instead of whatever command the connecting user tries to use.

```
command="/usr/share/munin/munin-async --spoolfetch" ssh-rsa AAAA[...] munin@master
```

The following options are recommended for security, but are strictly not necessary for the munin-async connection to work

- You should add a “from=” parameter to the key to restrict where it can be used from.
- You should add hardening options. At the time of writing, these are “no-X11-forwarding”, “no-agent-forwarding”, “no-port-forwarding”, “no-pty” and “no-user-rc”.

Some of these may also be set globally in /etc/ssh/sshd_config.

```
no-port-forwarding,no-X11-forwarding,no-agent-forwarding,no-pty,no-user-rc,from=
↪"192.0.2.0/24",command="/usr/share/munin/munin-async --spoolfetch" ssh-rsa AAAA[.
↪..] munin@master
```

See the `sshd_config(5)` and `authorized_keys(5)` man pages for more information.

Munin-Plugin

The munin plugin is a simple executable, which role is to gather one set of facts about the local server (or fetching data from a remote machine via SNMP)

The plugin is called with the argument “config” to get metadata, and with no arguments to get the values. These are mandatory arguments for each plugin. We have some more standard arguments, which play a role in the process of automatic configuration.

The Munin plugin

Introduction

A Munin plugin is a simple executable invoked in a command line environment whose role is to gather a set of facts on a host and present them in a format Munin can use.

A plugin is usually called without any arguments. In this circumstance, the plugin returns the data in a ‘key value’ format. For example, the ‘load’ plugin, which comes standard with Munin, will output the current system load:

```
# munin-run load
load.value 0.03
```

All plugins must also support the argument ‘config’ to get metadata on the plugin:

```
# munin-run load config
graph_title Load average
graph_args --base 1000 -l 0
graph_vlabel load
graph_scale no
graph_category system
load.label load
graph_info The load average of the machine describes how many processes are in the
↳run-queue (scheduled to run "immediately").
load.info 5 minute load average
```

Plugins may support other arguments, but the two cases described above will work for any plugin.

See Also

- *How to use Munin plugins.*
- *How to write your own Munin plugins.*
- *Tips on Writing Munin Plugins.*

Environment variables accessible in plugins

The node automatically defines some environment vars. All the munin-related vars do begin with the `MUNIN_` prefix and are all capitals.

Munin related

MUNIN_DEBUG Defines the debug level the plugin should run in.

Default: 0

MUNIN_MASTER_IP Contains the IP of the connecting master. If using `munin-run`, it is equal to the "-" string.

MUNIN_CAP_MULTIGRAPH Defines if the master is able to understand the `multigraph` keyword.

Table 3.1: Values

Value	Description
0	Master does not understand the <code>multigraph</code> keyword.
1	Master does understand the <code>multigraph</code> keyword.

MUNIN_PLUGSTATE Defines the directory that a plugin must use if it wants to store stateful data that is shared with other plugins.

Default: `/var/lib/munin-node/$USER`

Note: Only the plugins that execute themselves as the same user can exchange data, for obvious security reasons.

MUNIN_STATEFILE Defines a file that the plugin must use if it wants to store stateful data for himself.

It is guaranteed to be unique, per plugin **and** per master. Therefore 2 masters will have 2 different state files for the same plugin.

Config related

Here is a list of other environment vars, that are derived from the `Munin::Common::Defaults` package.

```
MUNIN_PREFIX
MUNIN_CONFDIR
MUNIN_BINDIR
MUNIN_SBINDIR
MUNIN_DOCDIR
MUNIN_LIBDIR
MUNIN_HTMLDIR
MUNIN_CGIDIR
MUNIN_CGITMPDIR
MUNIN_DBDIR
MUNIN_PLUGSTATE
MUNIN_SPOOLDIR
MUNIN_MANDIR
MUNIN_LOGDIR
MUNIN_STATEDIR
MUNIN_USER
MUNIN_GROUP
MUNIN_PLUGINUSER
MUNIN_VERSION
MUNIN_PERL
MUNIN_PERLLIB
MUNIN_GOODSH
MUNIN_BASH
MUNIN_PYTHON
MUNIN_RUBY
MUNIN_OSTYPE
MUNIN_HOSTNAME
MUNIN_HASSETR
```

System related

Munin does redefine some system environment vars :

PATH This is redefined for security. It does provide a safe environment so that shell scripts are able to launch regular commands such as `cat`, `grep` without having to be explicit in their location.

LC_ALL & LANG This is redefined to ease the work of plugin authors. It enables a standard output when parsing common commands output.

Multigraph plugins

As of 1.4.0 Munin supports multigraph plugins.

What are they?

A multigraph plugin supports a “hierarchy of graphs” to provide drill-down from general graphs to more specific graphs. One of the most obvious cases for this is network switch graphing where showing per-port traffic for 48 ports in the main host view would be overwhelming. Therefore the `snmp__if_multi` plugin presents two graphs on the main host view: `if_bytes` and `if_errors`. If you click on the `if_bytes` graph you will arrive at another page showing the throughput on all interfaces. If you click on the `if_errors` graph you will arrive on a page showing errors on all interfaces.

When to use them?

Ordinarily one does not want to use multigraph plugins. This is because they quickly become much more like “ordinary software”, e.g. that the number of lines of code passes around 50-100 lines, and that the data structures become more complex than a very simple hash or array. Most Munin plugins are simple and quick (and fun) to write, and that is by many considered one of the killer features of Munin. A multigraph plugin quickly becomes more unwieldy to write and takes away the quickness and fun from plugin writing.

But, if in your plugins you notice

- duplication of code
- duplication of work
- you have more data than you know how to present in one or a few graphs

and this bothers you or makes things unnecessarily slow you may want to write a multigraph plugin

Features often needed

It turns out that multigraph plugins are written to generate graphs for all network devices or all disk devices on a system. There is a definitive need to provide filtering (include) features, such as device name patterns for disk devices or media types for network boxes so that only e.g. ethernet and ppp devices are included in the graphs, and not the loopback and serial devices (unless the serial device is actually interesting since it’s really a long haul WAN line). Or, on the other hand a exclude feature to drop specifically uninteresting things. How to make one?

It’s quite simple, even if it’s not as simple as without multigraph.

The setup is done in the usual way, with `graph_title` and other configuration items for the two “root” graphs of the multigraph plugin:

```
multigraph if_bytes
graph_title $host interface traffic
graph_order recv send
graph_args --base 1000
graph_vlabel bits in (-) / out (+) per \${graph_period}
graph_category network
graph_info This graph shows the total traffic for $host

send.info Bits sent/received by $host
```

```
recv.label recv
recv.type DERIVE
recv.graph no
recv.cdef recv,8,*
recv.min 0
send.label bps
send.type DERIVE
send.negative recv
send.cdef send,8,*
send.min 0

multigraph if_errors
graph_title $host interface errors
graph_order recv send
graph_args --base 1000
graph_vlabel errors in (-) / out (+) per \${graph_period}
graph_category network
graph_info This graph shows the total errors for $host

send.info Errors in outgoing/incoming traffic on $host
recv.label recv
recv.type DERIVE
recv.graph no
recv.cdef recv,8,*
recv.min 0
send.label bps
send.type DERIVE
send.negative recv
send.cdef send,8,*
send.min 0
```

Then for each of the interfaces the plugin emits these configuration items (interface number is indexed with `$if` in this, and should be replaced with name or number by the plugin itself, likewise for the other settings such as `$alias`, `$speed` and `$warn`. `\${graph_period}` is substituted by Munin.

```
multigraph if_bytes.if_$if

graph_title Interface $alias traffic
graph_order recv send
graph_args --base 1000
graph_vlabel bits in (-) / out (+) per \${graph_period}
graph_category network
graph_info This graph shows traffic for the "$alias" network interface.
send.info Bits sent/received by this interface.
recv.label recv
recv.type DERIVE
recv.graph no
recv.cdef recv,8,*
recv.max $speed
recv.min 0
recv.warning -$warn
send.label bps
send.type DERIVE
send.negative recv
send.cdef send,8,*
send.max $speed
send.min 0
send.warning $warn

multigraph if_errors.if_$if

graph_title Interface $alias errors
```



```

graph_order recv send
graph_args --base 1000
graph_vlabel bits in (-) / out (+) per \${graph_period}
graph_category network
graph_info This graph shows errors for the \"$alias\" network interface.
send.info Errors in outgoing/incoming traffic on this interface.
recv.label recv
recv.type DERIVE
recv.graph no
recv.cdef recv,8,*
recv.max $speed
recv.min 0
recv.warning 1
send.label bps
send.type DERIVE
send.negative recv
send.cdef send,8,*
send.max $speed
send.min 0
send.warning 1

```

As you probably can see the hierarchy is provided by the “multigraph” keyword:

```

multigraph if_bytes
multigraph if_bytes.if_1
multigraph if_bytes.if_2
...
multigraph if_errors
multigraph if_errors.if_1
multigraph if_errors.if_2
...

```

When it comes to getting readings from the plugin this is done with the normal `fieldname.value` protocol, but with the same multigraph “commands” between each value set as between the each “config” set.

Important: The plugin’s name is `snmp__if_multi` but, unlike all other plugins, that name never appears in the munin html pages. The “multigraph” keyword overrides the name of the plugin. If multiple plugins try to claim the same names (the same part of the namespace) this will be logged in `munin-update.log`.

Notes

For 1.4.0 we never tested with deeper levels of graphs than two as shown above. If you try deeper nestings anything could happen! ;-)

Other documentation

Protocol extension: multiple graphs from one plugin

Multigraph plugins are implemented in 1.4.0 and on.

Objective

The object of this extension is to help with one issue:

- Quite a few plugins could after execution with very little additional overhead report on several measurable aspects of whatever it is examining. In these cases it will be cheaper to execute one plugin once to produce multiple graphs instead of executing multiple plugins to generate the same graphs.

This one-plugin one-graph property has resulted in the `if_` and `if_err_` plugins which are basically the same - almost identical code being maintained twice instead of once. The sensors plugins which reports on temperatures, fan speeds and voltages - running one `sensors` command each time (a slow executing program) or caching the results. There are several plugins that cache execution results because of this.

In all we should be able to maintain fewer plugins with less complexity than we are able to now.

Network protocol

A server that is capable of handling “multigraph” output **MUST** announce this to the node - otherwise the node **MUST** only announce and give access to single-graph plugins.

```
> # munin node at lookfar.langfeldt.net
< cap multigraph
> cap multigraph
< list
> if df netstat interrupts ...
< fetch if
> multigraph if_eth0
> out.value 6570
> in.value 430986
> multigraph if_err_eth0
> rcvd.value 0
> trans.value 0
> multigraph if_eth1
> ...
> multigraph if_err_eth1
> ...
> .
< quit
```

If the server had not announced `cap multigraph` the node **MUST NOT** respond with the names of multigraph plugins when the server issues a `list` command. This is to stay compatible with old munin masters that do not understand multigraph.

The value of each consecutive multigraph attribute show above was used to preserve compatibility with present `if_` and `if_err_` wildcard plugins. The field names in the response likewise. When combining separate plugins into one please keep this compatibility issue in mind.

The response to the `config plugin` protocol command **MUST** be similarly interspersed with multigraph attributes.

Node issues

This introduces the need for the node to know which plugins are multigraph. Since the node runs each and every plugin with “config” at startup (or when receiving HUP) it can simply examine the output. If the output contains `/^multigraph\s+/` then the plugin is a multigraph plugin and **MUST** be kept on a separate, additional list of plugins only shown to the masters with multigraph capability.

Plugin issues

In case a multigraph plugin is attempted installed on a node which does not understand multigraph capability it will be able to detect this by the lack of the environment variable `MUNIN_CAP_MULTIGRAPH` that the node uses to communicate that it knows about multigraph plugins. If this environment variable is absent the plugin **SHOULD** not make any kind of response to any kind of request.

In the perl and sh libraries support libraries there are functions to detect if the plugin is run by a capable node and if not simply emit dummy `graph_title` and other graph values to make it obvious that the plugin finds the node/environment lacking.

Future development of Multigraph plugins

The features in the following paragraphs are not implemented, and may never be. They were things and issues that were considered while planning the multigraph feature, but did not make it into 1.4.

Plugin issues

This is not implemented or otherwise addressed

For a multigraph plugin replacing `if_` and `if_err_` we probably want a static behavior, as network interfaces are easily taken up and down (`ppp*`, `tun*`).

To preserve the static behavior of the present wildcard plugins the node can somehow preserve the needed data in `munin-node.conf` or `/etc/munin/plugin-conf.d` and pass the response to the plugin in some environment variable to tell it what to do so the same is done each time. This must be user editable so that it changes when more network interfaces is added, or to enable removing graphing of a specific interface which though present does not actually pass any traffic.

```
[if]
multigraph :eth0 :eth1 err:eth0 err:eth1
```

The separator character may well be something different than `:"`. Any character not normally allowed in a plugin name should suffice.

Sample output

See *Multigraph plugins* for an example.

Protocol extension: dirtyconfig

The `dirtyconfig` capability is implemented in munin 2.0 and on.

Objective

Reduce execution time for plugins.

Description

Munin plugins are usually run twice. Once to provide configuration, and once to provide values.

Plugins which have to fetch data in order to provide meaningful configuration can use the “dirtyconfig” capability to send both configuration and values in the same run.

Using “dirtyconfig”, plugins no longer have to be run twice. There is no longer a need to keep a state file to keep state between “config” and “fetch” invocations for plugins with long execution times.

Network protocol

```
>> command from master to node
<< response from node to master
```

```
<< # munin node at somewhere.example.com
>> cap dirtyconfig
<< cap dirtyconfig
>> list
<< lorem ...
>> config lorem
<< graph_title Lorem ipsum
<< lorem.label Lorem
<< lorem.value 1
```

The master and node exchange capabilities with the `cap` command, with an argument list containing supported capabilities.

The server must send `cap` with `dirtyconfig` as one of the arguments.

The node must respond with `cap`, and include `dirtyconfig` as one of the capabilities.

Effects

The munin node will sets the `MUNIN_CAP_DIRTYCONFIG` variable to 1 in the plugin environment.

The munin master will call the plugin once with `config` plugin, and if the output includes `.value` fields, it will skip the `fetch` plugin step.

Using dirtyconfig

In a plugin, check the environment variable `MUNIN_CAP_DIRTYCONFIG`, ensure it has a value of 1.

If this is correct, you can emit values when the plugin is called with the `config` argument.

sample plugin

```
#!/bin/sh

emit_config() {
    echo "graph_title test with single word"
    echo "graph_category test"
    echo "test.label test"
}

emit_values() {
    echo "test.value 1"
}

case "$1" in
    config)
        emit_config
        if [ "$MUNIN_CAP_DIRTYCONFIG" = "1" ]; then
            emit_values
        fi
        ;;
    *)
        emit_values
        ;;
esac
```

SNMP Plugins

- *Using SNMP Plugins*

Using munin plugins

Default Installation

The default directory for plugin scripts is `/usr/share/munin/plugins/`. A plugin is activated when a symbolic link is created in the `servicedir` (usually `/etc/munin/plugins/` for a package installation of Munin) and `munin-node` is restarted.

The utility `munin-node-configure` is used by the Munin installation procedure to check which plugins are suitable for your node and create the links automatically. It can be called everytime when a system configuration changes (services, hardware, etc) on the node and it will adjust the collection of plugins accordingly.

To have `munin-node-configure` remove plugins for software that may no longer be installed, use the option `'--remove-also'`.

Installing Third Party Plugins

To use a Munin plugin being delivered from a *3rd-Party*, place it in directory `/usr/local/munin/lib/plugins` (or any other directory), make it executable, and create the service link. It is also possible to place the plugin directly into the `servicedir`, but this is not recommended for the following reasons: * it undermines the utility `munin-node-configure` * it is not appropriate for *wildcard plugins* * it interferes with SELinux

It is also possible to put 3rd-Party plugins in the *official* plugin directory (usually `/usr/share/munin/plugins`), but this runs the risk of having said plugins overwritten by distribution updates.

Configuring

`/etc/munin/plugin-conf.d` (sometimes `/etc/opt/munin/plugin-conf.d`) is where plugin configuration files are stored.

To make sure that plugin configurations are updated with software updates admins should not change the file `munin-node` which is delivered with the munin package. Instead place customized configuration in a file called `zzz-myconf`. As the config files are read in alphabetical order, this file is read last and will override configuration data found in the other files.

The file should consist of one or more sections, one section for each (group of) plugin(s) that should run with different privileges and/or environment variables.

Start a plugins configuration section with the plugins name in square brackets:

[<plugin-name>] The following lines are for <plugin-name>. May include one wildcard ('*') at the start or end of the plugin-name, but not both, and not in the middle.

After that each section can set attributes in the following format, where all attributes are optional.

user <username|userid> Run plugin as this user

Default: `munin`

group <groupname|groupid>[, <groupname|groupid>] [...] Run plugin as this group. If group is inside parentheses, the plugin will continue if the group doesn't exist.

What does comma separated groups do? See `$EFFECTIVE_GROUP_ID` in the [manual page for perlvar](#)

Default: `munin`

env.var <variable content> Will cause the environment variable <var> to be set to <contents> when running the plugin. More than one env line may exist. See the individual plugins to find out which variables they care about.

There is no need to quote the variable content.

host_name <host-name> Forces the plugin to be associated with the given host, overriding anything that “plugin config” may say.

timeout <seconds> Maximum number of seconds before the plugin script should be killed when fetching values. The default is 10 seconds, but some plugins may require more time.

command <command> Run <command> instead of plugin. %c will be expanded to what would otherwise have been run. E.g. command sudo -u root %c.

Note: When configuring a munin plugin, add the least amount of extra privileges needed to run the plugin. For instance, do not run a plugin with “user root” to read syslogs, when it may be sufficient to set “group adm” instead.

Examples:

```
[mysql*]
user root
env.mysqlopts --defaults-extra-file=/etc/mysql/debian.cnf

[exim_mailqueue]
group mail, (Debian-exim)

[exim_mailstats]
group mail, adm

[ldap_*]
env.binddn cn=munin,dc=foo,dc=bar
env.bindpw secret

[snmp_*]
env.community SecretSNMPCommunityString

[smart_*]                # The following configuration affects
                          # every plugin called by a service-link starting with_
↪smart_                  # Examples: smart_hda, smart_hdb, smart_sda, smart_sdb

user root
group disk
```

Plugin configuration is optional.

Testing

To test if the plugin works when executed by munin, you can use the *munin-run* command.

```
# munin-run myplugin config
# munin-run myplugin
# munin-run -d myplugin
```

Examples:

```
# munin-run df_abs config
graph_title Filesystem usage (in bytes)
graph_args --base 1024 --lower-limit 0
graph_vlabel bytes
graph_category disk
graph_total Total
_dev_mapper_vg_demo_lv_root__.label /
```

```

_dev_mapper_vg_demo_lv_root__.cdef _dev_mapper_vg_demo_lv_root__,1024,*
tmpfs__dev_shm.label /dev/shm
tmpfs__dev_shm.cdef tmpfs__dev_shm,1024,*
_dev_vda1__boot.label /boot
_dev_vda1__boot.cdef _dev_vda1__boot,1024,*
_dev_mapper_vg_demo_lv_tmp__tmp.label /tmp
_dev_mapper_vg_demo_lv_tmp__tmp.cdef _dev_mapper_vg_demo_lv_tmp__tmp,1024,*
_dev_mapper_vg_demo_lv_var__var.label /var
_dev_mapper_vg_demo_lv_var__var.cdef _dev_mapper_vg_demo_lv_var__var,1024,*

# munin-run -d df_abs
# Processing plugin configuration from /etc/munin/plugin-conf.d/df
# Processing plugin configuration from /etc/munin/plugin-conf.d/fw_
# Processing plugin configuration from /etc/munin/plugin-conf.d/hddtemp_smartctl
# Processing plugin configuration from /etc/munin/plugin-conf.d/munin-node
# Processing plugin configuration from /etc/munin/plugin-conf.d/postfix
# Processing plugin configuration from /etc/munin/plugin-conf.d/sendmail
# Setting /rgid/ruid/ to /99/99/
# Setting /egid/euid/ to /99 99/99/
# Setting up environment
# Environment exclude = none unknown iso9660 squashfs udf romfs ramfs debugfs_
↪binfmt_misc rpc_pipefs fuse.gvfs-fuse-daemon
# About to run '/etc/munin/plugins/df_abs'
_dev_mapper_vg_demo_lv_root__.value 1314076
tmpfs__dev_shm.value 0
_dev_vda1__boot.value 160647
_dev_mapper_vg_demo_lv_tmp__tmp.value 34100
_dev_mapper_vg_demo_lv_var__var.value 897644

```

Tips for writing good munin plugins

When developing plugins for Munin, there's some guidelines that should be observed.

Error Handling

Munin plugins should handle error conditions in a fashion that make them easy to understand and debug. Use these guidelines when developing a plugin:

- Output may always contain comments. Use # blocks within the output to give more information
- If an error occurs in the plugin, two things should happen:
 - A non-zero exit code must be issued
 - A descriptive message should be written to STDERR. On a deployed plugin, this message will appear in munin-node.log. When invoked via munin-run, it'll appear in the console.

Handling temporary files

Munin plugins often run with elevated privileges.

When creating and using temporary files, it is important to ensure that this is done securely.

Example shell plugin

```
#!/bin/sh

# Allow others to override mktemp command with env.mktemp_command in the plugin_
↳config
mktemp_command="${mktemp_command:-/bin/mktemp}"

# make a temporary file, exit if something goes wrong, and ensure it is removed_
↳after exit
my_tempfile=$(mktemp_command) || exit 73
trap 'rm -f "$my_tempfile"' EXIT

# rest of the plugin...
```

Example perl plugin

For perl, you have better tools available to keep data in memory, but if you need a temporary file or directory, you can use `File::Temp`.

```
#!/usr/bin/perl

use strict;
use warnings;

# make a tempfile, it will be removed on plugin exit
use File::Temp qw/ tempfile /;
my ($fh, $filename) = tempfile();
```

Writing a munin plugin

Wiki Pages:

- [How to write plugins](#)
- [The Concise guide to plugin authoring](#)
- [How to write SNMP Munin plugins](#)

A munin plugin is a small executable. Usually, it is written in some interpreted language.

In its simplest form, when the plugin is executed with the argument “config”, it outputs metadata needed for generating the graph. If it is called with no arguments, it outputs the data which is to be collected, and graphed later.

Plugin output

The minimum plugin output when called with “config” it must output the graph title.

It should also output a label for at least one datasource.

```
graph_title Some title for our plugin
something.label Foofoo per second
```

When the plugin is executed with no arguments, it should output a value for the datasource labelled in “config”. It must not output values for which there are no matching labels in the configuration output.

```
something.value 42
```

For a complete description of the available fields, see the *Plugin reference*.

Example shell plugin

The base of a plugin is a small option parser, ensuring the plugin is called with the correct argument, if any.

Two main functions are defined: One for printing the configuration to the standard output, and one for printing the data. In addition, we have defined a function to generate the data itself, just to keep the plugin readable.

The “output_usage” function is there just to be polite, it serves no other function. :)

```
#!/bin/sh

output_config() {
    echo "graph_title Example graph"
    echo "plugins.label Number of plugins"
}

output_values() {
    printf "plugins.value %d\n" $(number_of_plugins)
}

number_of_plugins() {
    find /etc/munin/plugins -type l | wc -l
}

output_usage() {
    printf >&2 "%s - munin plugin to graph an example value\n" ${0##*/}
    printf >&2 "Usage: %s [config]\n" ${0##*/}
}

case $# in
    0)
        output_values
        ;;
    1)
        case $1 in
            config)
                output_config
                ;;
            *)
                output_usage
                exit 1
                ;;
        esac
        ;;
    *)
        output_usage
        exit 1
        ;;
esac
```

Activating the plugin

Place the plugin in the `/etc/munin/plugins/` directory, and make it executable. Note that most distributions place plugins in a different directory, and ‘activate’ them by symlinking them to `/etc/munin/plugins`. New module development should use a similar approach so that in-process development doesn’t get run by mistake.

Any time a new plugin is placed or symlinked into `/etc/munin/plugins`, `munin-node` should be restarted.

Debugging the plugin

Plugins are just small programs or scripts, and just like small programs, are prone to problems or unexpected behaviour. When either developing a new plugin, or debugging an already existing one, use the following information to help track down the problem:

- A plugin may be tested ‘by hand’ by using the command ‘munin-run’. Note the plugin needs to have been activated before this will work (see above).
- If an error occurs, error messages will be written to STDERR, and exit status will be non-zero.
- If a plugin is already activated, any errors that may happen when the ‘munin-node’ cron job is executed will be logged, via stderr, to /var/log/munin/munin-node.log

Supersampling

Every monitoring software has a polling rate. It is usually 5 min, because it’s the sweet spot that enables frequent updates yet still having a low overhead.

Munin is not different in that respect: its data fetching routines have to be launched every 5 min, otherwise you’ll face data loss. And this 5 min period is deeply ingrained in the code. So changing it is possible, but very tedious and error prone.

But sometimes we need a very fine sampling rate. Every 10 seconds enables us to track fast changing metrics that would be averaged out otherwise. Changing the whole polling process to cope with a 10s period is very hard on hardware, since now every update has to finish in these 10 seconds.

This triggered an extension in the plugin protocol, commonly known as “supersampling”.

Overview

The basic idea is that fine precision should only be for selected plugins only. It also cannot be triggered from the master, since the overhead would be way too big.

So, we just let the plugin sample itself the values at a rate it feels adequate. Then each polling round, the master fetches all the samples since last poll.

This enables various constructions, mostly around “streaming” plugins to achieve highly detailed sampling with a very small overhead.

Notes

This protocol is currently completely transparent to *munin-node*, and therefore it means that it can be used even on older (1.x) nodes. Only a 2.0 *master* is required.

Protocol details

The protocol itself is derived from the *OPTIONS* extension.

Config

A new plugin directive is used, *update_rate*. It enables the master to create the rrd with an adequate step.

Omitting it would lead to rrd averaging the supersampled values onto the default 5 min rate. This means **data loss**.

Note: Heartbeat

The heartbeat has always a 2 step size, so failure to send all the samples will result with unknown values, as expected.

Note: Data size

The RRD file size is always the same in the default config, as all the RRA are configured proportionally to the `update_rate`. This means that, since you'll keep as much data as with the default, you keep it for a shorter time.

Fetch

When spoolfetching, the epoch is also sent in front of the value. Supersampling is then just a matter of sending multiple epoch/value lines, with monotonically increasing epoch.

Note: Note that since the epoch is an integer value for `rrdtool`, the smallest granularity is 1 second. For the time being, the protocol itself does also mandates integers. We can easily imagine that with another database as back-end, an extension could be hacked together.

Compatibility with 1.4

On older 1.4 masters, only the last sampled value gets into the RRD.

Sample implementation

The canonical sample implementation is `multicpu1sec`, a contrib plugin on github. It is also a so-called streaming plugin.

Streaming plugins

These plugins fork a background process when called that streams a system tool into a spool file. In `multicpu1sec`, it is the `mpstat` tool with a period of 1 second.

Undersampling

Some plugins are on the opposite side of the spectrum, as they only need a lower precision.

It makes sense when :

- data should be kept for a *very* long time
- data is *very* expensive to generate and it varies only slowly.

Relations

- Each Munin master may monitor one or more Munin nodes (1:n)
- **More than one Munin master may monitor one or more Munin nodes (n:m)**
 - Does this confuse lightly stupid plugins?
 - Is “multi-master” configurations tested, known and/or documented?

- Does the Plugin-writing-howto describe how the plugin should behave if queried more often than in five minutes intervals and/or from different Munin masters?
- Each Munin node controls one or more plugins (1:n)
- **Each plugin returns, when queried:**
 - One or more general directives to control the plugin itself, with corresponding values
 - One or more data sources (fields) described by fieldname (1:n)
 - Each data source has one or more attributes (1:n), with corresponding values

Protocols

The Munin Protocols

Here we describe the rules for collaboration and communication between *Munin's components*.

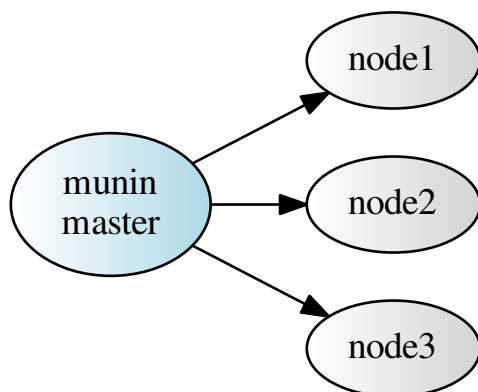
Introduction

Contents on this page will focus on already implemented features. For *proposals* and *ideas* look in the [Wiki](#).

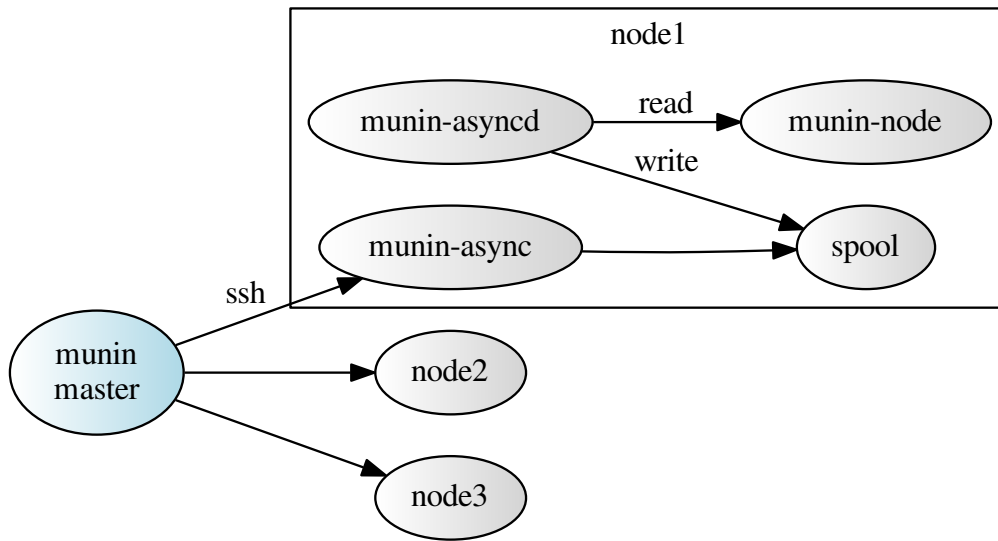
Concepts

Fetching Data

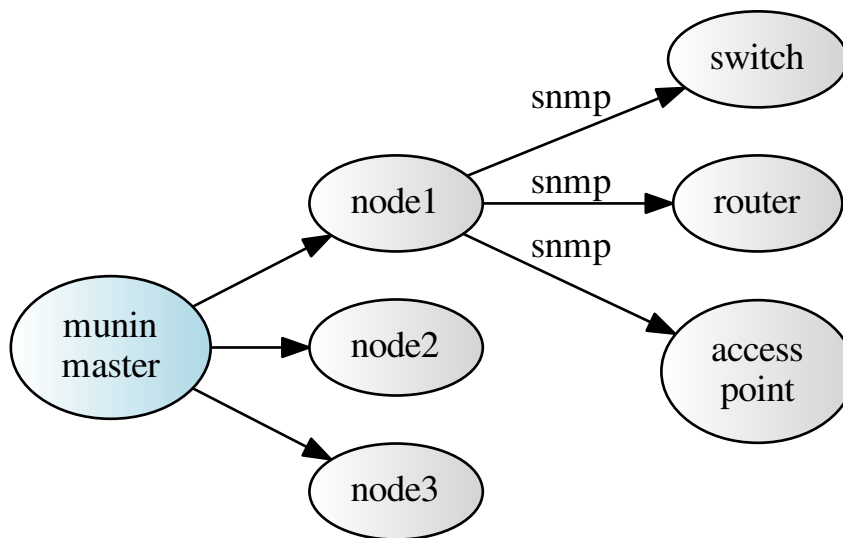
Poller-based monitoring infrastructure



Using the *Asynchronous proxy node*:



Using *SNMP Plugins*.



Network Protocol

Common Plugins

- See *Protocol for data exchange between master and node*

Multigraph Plugins

- See *Protocol for Multigraph Plugins*

Dirtyconfig plugins

- See *Protocol extension: dirtyconfig*

Syntax

Munin's Syntax

POD style documentation

Wiki Pages:

- [munindoc](#)

Configuration

- For the *Munin master* in */etc/munin/munin.conf*.
- For the *Munin node* daemon in */etc/munin/munin-node.conf*.
- For the *Munin plugins* in */etc/munin/plugin-conf.d/...*

Magic Markers

Munin can only autoconfigure plugins that have the corresponding (optional) magic markers. Magic markers are prefixed with `###` and consists of a keyword, an equal sign, and one or more whitespace-separated values.

For a plugin that is part of munin, you should expect to see:

```
### family=auto
### capabilities=autoconf suggest
```

family

For the magic marker `family`, the following values may be used.

auto This is a plugin that can be automatically installed and configured by `munin-node-configure`

snmpauto This is a plugin that can be automatically installed and configured by `munin-node-configure` if called with `--snmp` (and related arguments)

manual This is a plugin that is to be manually configured and installed

contrib This is a plugin which has been contributed to the munin project by others, and has not been checked for conformity to the plugin standard.

test This is a test plugin. It is used when testing munin.

example This is an example plugin. It serves as a starting point for writing new plugins.

capabilities

For the magic marker `capabilities`, the following values may be used.

autoconf The plugin may be automatically configured by “munin-node-configure”.

suggest The plugin is a wildcard plugin, and may suggest a list of link names for the plugin.

Datatypes

GAUGE

“is for things like temperatures or number of people in a room or the value of a RedHat share.” (Source: [rrdcreate man page](#))

If a plugin author does not declare datatype explicitly, GAUGE is the default datatype.

COUNTER

“is for continuous incrementing counters like the `ifInOctets` counter in a router. The COUNTER data source assumes that the counter never decreases, except when a counter overflows. The update function takes the overflow into account. The counter is stored as a per-second rate. When the counter overflows, RRDtool checks if the overflow happened at the 32bit or 64bit border and acts accordingly by adding an appropriate value to the result.” (Source: [rrdcreate man page](#))

Note: on COUNTER vs DERIVE

by Don Baarda <don.baarda@baesystems.com> from <http://oss.oetiker.ch/rrdtool/doc/rrdcreate.en.html>

If you cannot tolerate ever mistaking the occasional counter reset for a legitimate counter wrap, and would prefer “Unknowns” for all legitimate counter wraps and resets, always use DERIVE with `min=0`. Otherwise, using COUNTER with a suitable max will return correct values for all legitimate counter wraps, mark some counter resets as “Unknown”, but can mistake some counter resets for a legitimate counter wrap.

For a 5 minute step and 32-bit counter, the probability of mistaking a counter reset for a legitimate wrap is arguably about 0.8% per 1Mbps of maximum bandwidth. Note that this equates to 80% for 100Mbps interfaces, so for high bandwidth interfaces and a 32bit counter, DERIVE with `min=0` is probably preferable. If you are using a 64bit counter, just about any max setting will eliminate the possibility of mistaking a reset for a counter wrap.

DERIVE

“will store the derivative of the line going from the last to the current value of the data source. This can be useful for gauges, for example, to measure the rate of people entering or leaving a room. Internally, derive works exactly like COUNTER but without overflow checks. So if your counter does not reset at 32 or 64 bit you might want to use DERIVE and combine it with a MIN value of 0.” (Source: [rrdcreate man page](#))

ABSOLUTE

“is for counters which get reset upon reading. This is used for fast counters which tend to overflow. So instead of reading them normally you reset them after every read to make sure you have a maximum time available before the next overflow. Another usage is for things you count like number of messages since the last update.” (Source: [rrdcreate man page](#))

API

Munin's API

This document explains how to get Munin onto your system, where to get help, and how to report bugs.

Prerequisites

In order for you to install Munin you must have the following:

Building munin

In order to build munin, you need:

- A reasonable Perl 5 (Version 5.10 or newer)
- The Module::Build perl module
- The perl modules listed in “requires” in Build.PL

Developers / packagers need, in addition to the above

- The dependencies listed in “test_requires” in Build.PL
- RRDtool perl bindings

In order to build the documentation, you need:

- sphinx

Installing RRDtool bindings

The RRDtool perl bindings needed by munin are normally only installed for the system perl. To install munin using a separate perl installation, add the Alien::RRDtool perl module.

Running munin

Munin master

In order to run the munin master, you need:

- A reasonable perl 5 (Version 5.10 or newer)
- All the perl modules used when building Munin
- A web server (optional)

Munin node

The munin node is lighter on the requirements, and need only the following perl modules:

- Net::Server
- Net::Server::Fork
- Time::HiRes
- Net::SNMP (Optional)

The Munin plugins run by the node have their own needs. Many plugins need libraries or utilities related to what they monitor. Please refer to each plugin.

Installing Munin

Due to *Munin's Architecture* you have to install two different software packages depending on the role, that the machine will play.

You will need to install “munin-master” on the machine that will collect data from all nodes, and graph the results. When starting with Munin, it should be enough to install the Munin master on one server.

The munin master runs *munin-httpd* which is a basic webserver which provides the munin web interface on port 4948/tcp.

Install “munin-node” on the machines that shall be monitored by Munin.

Source or packages?

With open source software, you can choose to install binary packages or install from source-code.

Note: We *strongly* recommend a packaged install, as the source distribution isn't as tested as the packaged one. The current state of the packages is so satisfactory, that even the developers use them instead.

Installing Munin on most relevant operating systems can usually be done with with the systems package manager, typical examples being:

Installing Munin from a package

FreeBSD

From source:

```
cd /usr/ports/sysutils/munin-master && make install clean
cd /usr/ports/sysutils/munin-node && make install clean
```

Binary packages:

```
pkg install munin-master
pkg install munin-node
```

Debian/Ubuntu

Munin is distributed with both Debian and Ubuntu.

In order to get Munin up and running type

```
sudo apt-get install munin-node
```

on all nodes, and

```
sudo apt-get install munin
```

on the master.

Please note that this might not be the latest version of Munin. On Debian you have the option of enabling “backports”, which may give access to later versions of Munin.

RedHat / CentOS / Fedora

Current versions are available at [EPEL](#).

In order to install Munin type

```
sudo yum install munin-node
```

on all nodes, and

```
sudo yum install munin
```

on the master.

You will have to enable the services in systemd to get them up and running.

Likely you will have to fix SELinux issues when using 3rd-Party plugins and SELinux active and set to *enforcing mode* on the Munin node. In case you get competent and friendly support on [SELinux mailinglist](#).

Other systems

On other systems, you are probably best off compiling your own code. See *Installing Munin from source*.

Installing Munin from source

Warning: Usually you don’t want to do that. The following lines are for completeness, and reference for packagers.

The other reason would be because you want to contribute to the development of Munin, and then you should use a development install.

If there are no binary packages available for your system, or if you want to install Munin from source for other reasons, follow these steps:

We recommend downloading a release tarball, which you can find on [sourceforge.net](#).

Alternatively, if you want to hack on Munin, you should clone our git repository by doing.

```
git clone git://github.com/munin-monitoring/munin
```

Please note that a git checkout will need some more build-dependencies than listed below, in particular the Python Docutils and Sphinx.

Build dependencies on Debian / Ubuntu

In order to build Munin from source you need a number of packages installed. On a Debian or Ubuntu system these are:

- perl
- htmldoc
- html2text
- default-jdk

Configuring and installing

Warning for NFS users

If you're using NFS please note that the "make install" process is slightly problematic in that it (Module::Build actually) writes files under \$CWD. Since "make install" is usually run by root and root usually cannot write files on a NFS volume, this will fail. If you use NFS please install munin from /var/tmp, /tmp or some such to work around this.

Running make

There are make targets for node, master, documentation and man files. Generally you want to install everything on the master, and just the node and plugins on the nodes.

- Edit Makefile.config to suit your needs.
- Create the user "munin" with the primary group "munin".

The user needs no shell and no privileges. On most Linux systems the munin user's shell is the nologin shell (it has different paths on different systems - but the user still needs to be able to run cron jobs).

Node

For the node, you need only the common parts, the node and the plugins.

```
make
make install-common-prime install-node-prime install-plugins-prime
```

Master

For the master, this will install everything.

```
make
make install
```

Initial configuration

Node

Plugins

Decide which plugins to use. The munin node runs all plugins present in CONFDIR/plugins/

The quick auto-plug-and-play solution:

```
munin-node-configure --shell --families=contrib,auto | sh -x
```

Access

The munin node listens on all interfaces by default, but has a restrictive access list. You need to add your master's IP address.

The “cidr_allow”, “cidr_deny”, “allow” and “deny” statements are used.

cidr_allow uses the following syntax (the /32 is not implicit, so for a single host, you need to add it):

```
cidr_allow 127.0.0.0/8
cidr_allow 192.0.2.1/32
```

allow uses regular expression matching against the client IP address.

```
allow '^127.'
allow '^192.0.2.1$'
```

For specific information about the syntax, see [Net::Server](#). Please keep in mind that cidr_allow is a recent addition, and may not be available on all systems.

Startup

Start the node agent (as root) SBINDIR/munin-node. Restart it if it was already started. The node only discovers new plugins when it is restarted.

You probably want to use an init-script instead and you might find a good one under build/dists or in the build/resources directory (maybe you need to edit the init script, check the given paths in the script you might use).

Master

Add some nodes

Add some nodes to CONFDIR/munin.conf

```
[node.example.com] address 192.0.2.4
```

```
[node2.example.com] address node2.example.com
```

```
[node3.example.com] address 2001:db8::de:caf:bad
```

Configure web server

On the master, you need to configure a web server.

If you have installed “munin” through distribution packages, a webserver may have been configured for you already.

If you installed from source, there is a minimal configuration example in the “resources” directory in the source tarball.

For a more complex example, see *Apache virtualhost configuration*

Upgrade notes

Upgrading Munin from 2.0.x to 2.1.x

Munin HTTPD

munin-httpd replaces FastCGI. It is a basic webserver capable of serving pages and graphs.

To add transport layer security or authentication, use a webserver with more features as a proxy.

If you choose to use *munin-httpd*, set `graph_strategy` and `html_strategy` to “cgi”.

FastCGI

... is gone. It was hard to set up, hard to debug, and hard to support.

Upgrading Munin from 1.x to 2.x

This is a compilation of items you need to pay attention to when upgrading from Munin 1.x to munin 2.x

FastCGI

Munin graphing is now done with FastCGI.

Munin HTML generation is optionally done with FastCGI.

Logging

The web server needs write access to the `munin-cgi-html` and `munin-cgi-graph` logs.

Advanced Network

- *Monitoring “unreachable” hosts*
- TLS setup
- HOWTO Monitor Windows

Guidelines for developing Munin.

Munin development environment

Getting started

1. Install perl
2. Check out the munin repository
3. Install perl dependencies
4. Install munin in a sandbox
5. Start munin in a sandbox
6. Start hacking

Install perl

You need perl 5.10 or newer for munin development. Check your installed version with `perl --version`. If you have an older perl, look at using `perlbrew` to have perl in a sandbox.

Check out the munin repository

Munin is hosted on github. Clone the git repository, and enter the work directory.

```
git clone https://github.com/munin-monitoring/munin
cd munin
```

Install perl dependencies

Munin needs a lot of perl modules. The dependencies needed to develop, test, build and run munin is listed in the `Build.PL` file.

With the Debian osfamily

This includes Debian, Ubuntu, and many other operating systems.

Dependencies for running Munin from the development environment.

```
apt install libdbd-sqlite3-perl libdbi-perl \  
  libfile-copy-recursive-perl libhtml-template-perl \  
  libhtml-template-pro-perl libhttp-server-simple-perl \  
  libio-socket-inet6-perl liblist-moreutils-perl \  
  liblog-dispatch-perl libmodule-build-perl libnet-server-perl \  
  libnet-server-perl libnet-snmp-perl librrds-perl \  
  libnet-ssleay-perl libparams-validate-perl liburi-perl \  
  libwww-perl libxml-dumper-perl
```

Dependencies for running the Munin development tests:

```
apt install libdbd-pg-perl libfile-readbackwards-perl \  
  libfile-slurp-perl libio-stringy-perl libnet-dns-perl \  
  libnet-ip-perl libtest-deep-perl libtest-differences-perl \  
  libtest-longstring-perl libtest-mockmodule-perl \  
  libtest-mockobject-perl libtest-perl-critic-perl \  
  libxml-libxml-perl libxml-parser-perl
```

With modules from CPAN

```
perl Build.PL  
./Build installdeps
```

Install munin in a sandbox

The `dev_scripts` directory contains scripts to install munin in a sandbox. We also need to disable `taint` in the perl scripts to enable it to run outside the normal perl installation.

```
dev_scripts/install node  
dev_scripts/disable_taint
```

Run munin in a sandbox

Each of these can be done in a separate terminal window, to keep the logs apart.

Start a munin node. This will start the node in the background, and tail the log. If you hit Ctrl-C, the log tailing will stop, and the node will still run in the background.

```
dev_scripts/start_munin-node
```

The `contrib` directory contains a daemon used for simulating a lot of munin nodes. This step is optional. First output a number of node definition to the munin configuration, and then run the daemon in the background.

```
contrib/munin-node-debug -d > sandbox/etc/munin-conf.d/nodes.debug  
contrib/munin-node-debug &
```

Start a munin-update loop. Normally, `munin-update` runs from cron every 5 minutes.

```
while ;; do dev_scripts/run munin-update; sleep 60; done &
```

The munin httpd listens on <http://localhost:4948/> by default.

```
dev_scripts/run munin-httpd
```

Start hacking

Make changes, restart sandboxed services as necessary.

Make a git feature branch, commit changes, publish branch to a public git repository somewhere, submit pull requests, make things happen.

Munin development tests

Scope

The tests we use check for different things.

- Is function x in module y working as expected?
- Can we establish an encrypted network connection between two components?
- Do we follow the perl style guidelines?
- Does this component scale well?

The code tests are broadly separated by scope.

Inspired by <https://pages.18f.gov/automated-testing-playbook/principles-practices-idioms/>

Small

In this category, we place tests for simple classes and functions, preferably with fast execution and without using external resources.

Medium

Enabled with the TEST_MEDIUM variable set.

In this category, we test interaction between components. These may use the file system, fork processes, or access test data sets.

Large

Enabled with the TEST_LARGE variable set.

In this category, we may test the entire system.

A munin master, node, and plugins all running together would be placed in this category.

Performance and bottleneck testing would also be at home in this category.

Guidelines for developing Munin plugins

How to write Munin plugins

Writing a Munin plugin is astoundingly simple. If you know where the data is and know a minimum of scripting in perl/shell/awk/sed or something like that you can do it.

In way of explaining it all we'll write two plugins the simplest way. Since I'm kind of old school I'll use shell and shell tools for these plugins. You can do them completely in perl or any other language you like.

Load average plugin

On Unix you can get a hosts load average from the command `uptime`. However, on Linux the load average is also available from the file called `/proc/loadavg`. One less external command will make the plugin faster.

Getting a field value

```
$ cat /proc/loadavg
0.05 0.07 0.14 1/74 30026
```

The file consists of numbers separated by only one space. The Unix `cut` command should be good for that:

```
$ cut -d' ' -f1 /proc/loadavg
0.05
```

As easy as it gets. One thing wrong here though: Munin calls the plugins every 5 minutes (this may change in the future). The better value to report is not the first one but the second one. In the man page for `uptime` you'll see this called the 5 minute load average.

Also, munin wants the value in a more structured form:

```
# printf "load.value "; cut -d' ' -f2 /proc/loadavg
load.value 0.06
```

Here the `load` is called the field or field name, `value` the attribute, and the number is of course the value. (See our *complete overview of Munin specific terminology*).

That was the hard part of the plugin. The rest is just book-keeping.

Munin plugin config command

For munin to know how to draw a graph of the reported numbers, it calls the plugin script with `config` as the only argument. A minimal output looks like this:

```
graph_title Load average
graph_vlabel load
load.label load
```

The global attribute `graph_title` sets the title of the graph - in large lettering across the top. The other global attribute `graph_vlabel` labels the vertical axis of the graph. In addition `load.label` provides a color coded legend for the graphed line provided by `load.value`.

Everything in a script

The most trivial plugin is then this script:

```
#!/bin/sh

case $1 in
  config)
    cat <<'EOM'
graph_title Load average
graph_vlabel load
load.label load
EOM
    exit 0;;
esac

printf "load.value "
cut -d' ' -f2 /proc/loadavg
```

Testing it:

```
$ ./load
load.value 0.08
$ ./load config
graph_title Load average
graph_vlabel load
load.label load
```

Place the plugin in `/etc/munin/plugins`. To test it for real use `munin-run`. This sets up the environment for the plugin exactly like it would be when run from the `munin-node` network service.

```
# munin-run load
load.value 0.08
# munin-run load config
graph_title Load average
graph_vlabel load
load.label load
```

Finishing touches

There are a couple more things you can add to improve the plugin. For example Munin supports more explanatory legends, and the graph should be tweaked. For instance, one may add these attributes to the `config` output:

```
graph_args --base 1000 -l 0
graph_scale no
graph_category system
load.warning 10
load.critical 120
graph_info The load average of the machine describes how many processes are in the run
↳runqueue (scheduled to run "immediately").
load.info Average load for the five minutes.
```

The values of *graph_args* are passed to the graphing tool (rrd) to instruct it about how to draw the graphs.

`--base` is to make it scale the graph with a 1000 base (1000=1k 1000k=1M and so on. If you give the base as 1024 as you might when measuring bytes then 1024=1k 1024k=1M and so on. Disks are usually measured in units of 1000 due to the industry standard for marketing disks that people have gotten used to).

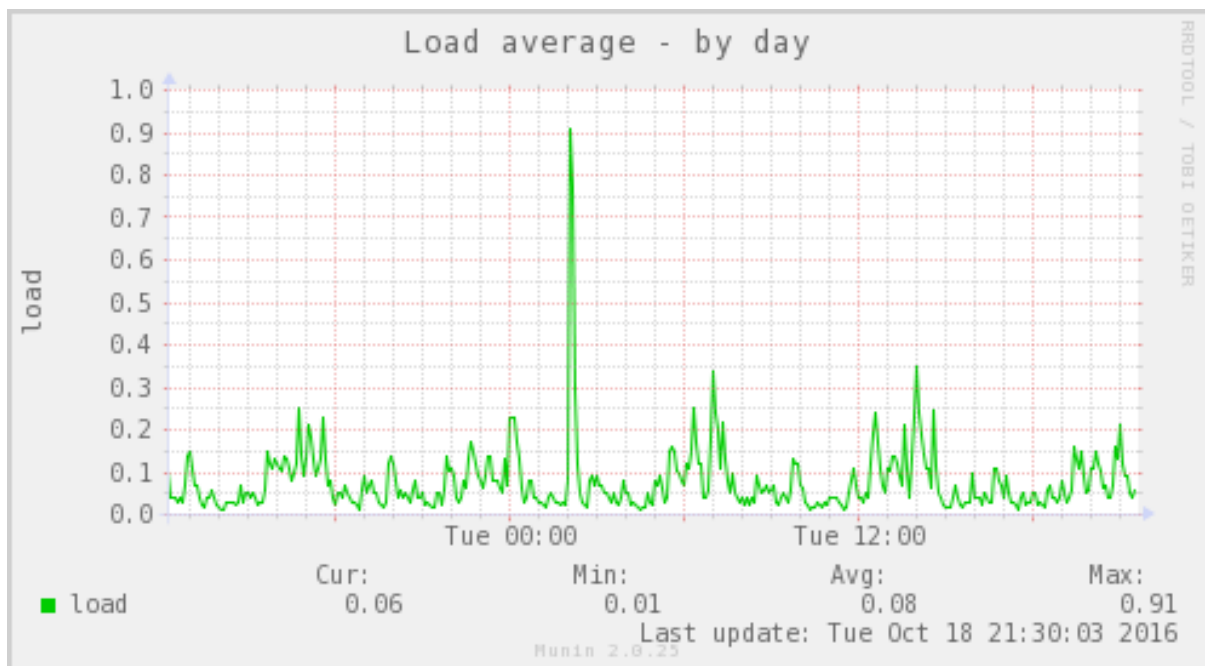
The `-l 0` sets the lowest value to 0. If all readings of a plugin were between 10 and 100 the lowest value on the graph might otherwise be set to 10. On a graph showing readings in percent you might add `--upper-limit 100` (of course some percentage readings goes past 100%).

graph_scale no makes munin (rrd) not scale the number. Normally a reading of 1000 would be scaled to 1k and 1000000 to 1M (according to scales set with `--base` explained above).

The *.warning* and *.critical* attributes are used to issue status messages. In the case of load average they're probably set statically by the plugin author. A plugin may also examine the system on which it runs to determine good values for these. The best way is for the plugin author provide defaults, and then code the plugin to get defaults from environment variables such as `$warning` and `$critical`.

The values *graph_info* attribute and each of the *.info* field-attributes are added as text on the html page under the graphs. They serve as legends for the graphs shown. In the case of this plugin there is not much to say - in other cases, when presenting output from more complex systems (much) more explanation is in order. Imagine writing for a person that knows Unix/networks/operating systems in general but not the specific sub-system the plugin measures in particular.

This results in a graph such as this:



The html load page looks like the [page](#) pointed to here. There you can see the use of the info attributes as well.

Restarting munin-node

The *munin-node* network service will not discover new plugins in the plugin directory until you restart it.

```
# service munin-node reload
Stopping Munin Node agents:          [ OK ]
Starting Munin Node:                 [ OK ]
```

Now you can check the reading by telnet:

```
# telnet localhost 4949
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
# munin node at foo.example.com
fetch load
load.value 0.06
.
quit
```

And now Munin will find the plugin as well - and you should have a readable graph within 15-20 minutes.

Here is the real source for the load average plugin for different architectures:

- [Linux load plugin](#)
- [FreeBSD load plugin](#)
- [Solaris load plugin](#)

You will see that they observe some additional niceties that I've not described here.

Error handling in plugins

The unix way of communicating errors in such small programs is to set the exit code. But just setting the exit code (exit 2, exit 3, exit 4) and not issuing a error message is not very friendly. Therefore, if there is a execution error, such as not being able to open a file that the plugin should be able to access, please both give a error message and exit with a non-zero value. In shell it goes like this:

```
if [ ! -r /proc/loadavg ] ; then
    echo Cannot read /proc/loadavg >&2
    exit -1
fi
```

In perl:

```
open(LOAD, "</proc/loadavg") or die "Could not open /proc/loadavg for reading: $!\n
↪";
```

Network interface plugin

This plugin demonstrates some additional points but it is much the same as the load plugin. Firstly this plugin will make two curves in one. This graph will show how many bytes pass over each network interface on the host it is run. In quite many Unices you'll find these numbers in the output of `/sbin/ifconfig`:

```
$ /sbin/ifconfig
eth1      Link encap:Ethernet  HWaddr 00:13:CE:63:45:B2
          inet addr:10.0.0.2  Bcast:10.0.0.255  Mask:255.255.255.0
          inet6 addr: fe80::213:ceff:fe63:45b2/64  Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:2610 errors:0 dropped:0 overruns:0 frame:0
```



```
TX packets:3162 errors:0 dropped:0 overruns:0 carrier:42
collisions:0 txqueuelen:1000
RX bytes:817182918 (779.3 MiB) TX bytes:2835962961 (2.6 GiB)
Interrupt:18 Base address:0xc000 Memory:b0204000-b0204fff
...
```

As usual in Linux though you can also find these numbers in a file. In this case it's `/proc/net/dev`:

```
Inter-| Receive | Transmit
face | bytes packets errs drop fifo frame compressed multicast| bytes packets
↪errs drop fifo colls carrier compressed
lo:22763978 191841 0 0 0 0 0 0 0 22763978 191841
↪0 0 0 0 0 0 0 0 0 0 0
eth0: 0 0 0 0 0 0 0 0 0 0 0 0
↪0 0 0 0 0 0 0 0 0 0 0
eth1:817283042 3242 0 0 0 0 0 0 0 2836088627
↪4372 0 0 0 0 42 0 0 0 0 0
sit0: 0 0 0 0 0 0 0 0 0 0 0 0
↪0 0 0 0 0 0 0 0 0 0 0
```

This is food for `awk`. For each interface we're interested in, it shows the interface name, received bytes and transmitted bytes. `awk` usually uses whitespace as column separator, but this file uses ":" as well. Fortunately we can adjust `awk`'s column separator.

```
$ awk -v interface="eth1" -F'[: \t]+' \
  '{ sub(/^ */, ""); // Remove leading space
  if ($1 == interface) print "down.value "$2"\nup.value "$10;
  }' /proc/net/dev
down.value 818579628
up.value 2837327179
```

But there is one important difference: The load plugin reports a number that can simply be plotted on the Y axis. These ethernet numbers will just continue to grow into the sky as long as the machine is up. What we actually want to graph is the increase in the numbers between each sampling measured in bits (or bytes) per second. Munin (`rrd`) will take the number and divide by the number of seconds between the samples (currently the sample interval is fixed at 5 minutes, or 300 seconds), so the "per second" part is taken care of. The bytes to bits we'll get into in a second. This is the appropriate `config` output for the plugin thus far:

```
graph_order down up
graph_title eth1 traffic
graph_args --base 1000
graph_vlabel bits in (-) / out (+) per ${graph_period}
down.label received
down.type COUNTER
```

The data type `COUNTER` here says that the value is a counter that keeps increasing rather than a `GAUGE` which the load reading was. If you put this into a script and add a `config` section you'll have a working plugin producing two curves in one graph.

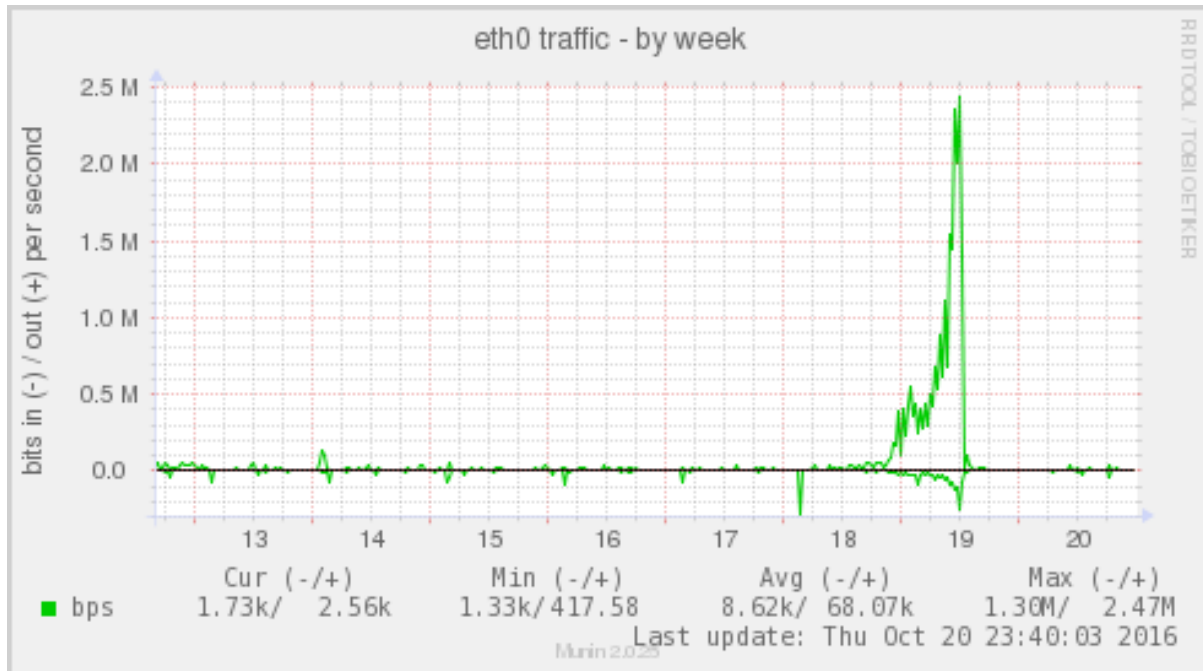
But there is more:

```
down.graph no
down.cdef down,8,*
up.label bps
up.type COUNTER
up.negative down
up.cdef up,8,*
```

Here are two or three points. The `.cdef` thing takes care of multiplying by 8 to get from bytes (or bytes per second) which is what the file shows to bits (i.e., bits per second), which is the unit most humans use when they think about network speeds.

There is a munin policy that input and output to the same device should be graphed in the same graph, this we already do. BUT, we want the output above the X axis and input below. This is done by first disabling graphing of the input (downloaded) value, then using *up.negative down*. Instead of just negating the down value this keeps the sign and gives the down graph the same color as the up graph. The values in the database are kept as normal, the whole thing is magicked by Munin while graphing.

The end result is this kind of graph:



Given a full set of info attributes (I've broken the first line to make it practical to read):

```
graph_info This graph shows the traffic of the eth0 network interface. Please note
that the traffic is shown in bits per second, not bytes. IMPORTANT: Since the
data source for this plugin use 32bit counters, this plugin is really unreliable
and unsuitable for most 100Mb (or faster) interfaces, where bursts are expected
to exceed 50Mbps. This means that this plugin is unsuitable for most production
environments. To avoid this problem, use the ip_ plugin instead.
```

```
up.info Traffic of the eth0 interface. Maximum speed is 1000Mbps
```

Then you end up with a [generated page like this](#).

DERIVE vs. COUNTER

To avoid spikes in the graph when counters are reset (as opposed to wrapping), use *`\${name}.type DERIVE`* and *`\${name}.min 0`*. Note that this will cause lost data points when the counter wraps, and should therefore not be used with plugins that are expected to wrap more often than be reset (or sampled). An example of this is the Linux *if_* plugin on 32bit machines with a busy (100Mbps) network.

The reasons behind this is rooted in the nature of 32 bit two's complement arithmetic and the way such numbers wrap around from huge positive numbers to huge negative numbers when they overflow. Please refer to these two articles in wikipedia to learn more: [Binary Arithmetic](#) and [Two's complement](#).

To summarize:

1. Use DERIVE
2. Use *`\${name}.min`* to avoid negative spikes

Graph category

If the plugin gives the *graph_category* attribute in its *config* output, the graph will be grouped together with other graphs of the same category. Please consult the [list of well-known categories](#).

Validate fieldnames

There are some restrictions on the characters you can use in field names. They are documented in [Notes on field names](#).

Since Munin version 1.3.3 and 1.2.6 we have support modules for shell and perl plugins (see next sections).

Perl and sed

These regular expressions should be applied to all field names to make them safe:

```
s/^[^A-Za-z_]/_/
s/[^A-Za-z0-9_]/_/g
```

Shell plugin

```
...
. $MUNIN_LIBDIR/plugins/plugin.sh
...
fieldname="$(clean_fieldname "$dev")"
...
```

Perl plugin

```
...
use Munin::Plugin;
...
my $fieldname=clean_fieldname($dev);
...
```

Going on

The [plugin documentation](#) should have all the information you need. I suggest the next thing you read about plugins is [Best Practices](#) which should tell you all you need to know to get nice graphs in as few tries as possible. If planning to write a plugin as a shell script, please read [Shell Plugins](#). If your plugin does not work like you think it should, try [Debugging Plugins](#). If you want to get the plugin autoconfigured on install and such take a look at wiki page [PluginConcise](#).

See also

- Concise guide to plugin authoring
- *Debugging Plugins*
- *Global plugin attributes*
- *Datasource-specific plugin attributes*
- *Multi-graph plugins*
- Shell Plugins
- Perl plugins

How To Write SNMP plugins

This HOWTO is not quite done yet.

As writing a Munin plugin is simple writing a SNMP one is even simpler. The SNMP agent on the device has done all the hard work for us. We just need to be able to autodetect if a particular device supports our plugin, and then present the stats the SNMP agent gives us.

If you do not know or understand what SNMP is or what a community string is used for please find (and read) some general SNMP material before you go on.

Supporting library

If you have used SNMP plugins with Munin you may have noticed that there is a uniform way to configure them. This is implemented in the perl library *Munin::Plugin::SNMP* (Perl module) which is supplied in Munin 1.4.

If you run *perldoc Munin::Plugin::SNMP* you'll see the programmers documentation for the module and its public functions - as usual. This also explains how to configure a plugin using this module for different authentications and versions of SNMP.

A load plugin

Returning to the absolute simplest case I would have picked a *snmp_load* plugin for this HOWTO, but not many devices supports that. Uptime on the other hand is very basic to SNMP devices. If you snmpwalk some device the uptime will be there among the first 10 lines:

```
DISMAN-EVENT-MIB::sysUpTimeInstance = Timeticks: (138007908) 15 days, 23:21:19.08
```

The numeric OID for that is

```
$ snmptranslate -On DISMAN-EVENT-MIB::sysUpTimeInstance
.1.3.6.1.2.1.1.3.0
```

A Perl script to get the basics

```
#!/usr/bin/perl

use strict;
use warnings;
use Munin::Plugin::SNMP;

my $session = Munin::Plugin::SNMP->session(-translate =>
                                           [ -timeticks => 0x0 ]);

my $uptime = $session->get_single (".1.3.6.1.2.1.1.3.0") || 'U';
```

```

if ($uptime ne 'U') {
    $uptime /= 8640000; # Convert to days
}

print "uptime.value ", $uptime, "\n";

```

The *-translate* option is to stop the SNMP stack from converting the timetics into a human readable time string, we want a integer to graph.

If you call this *snmp__uptime* and then in */etc/munin/plugins* make a symlink to it: If your device is called “switch” (this should be in DNS (or the hosts file) and possible to look up): *ln -s snmp_switch_uptime ...*

Then the plugin has to be configured, */etc/munin/plugin-conf.d/snmp* for the imagined device called “switch”:

```

[snmp_*]
env.version 2
env.community public

```

Now you can do *munin-run snmp_switch_uptime* as root:

```

# munin-run snmp_switch_uptime
uptime.value 15.979475462963

```

You need a config section too.

```

if (defined $ARGV[0] and $ARGV[0] eq "config") {
    my ($host) = Munin::Plugin::SNMP->config_session();
    print "host_name $host\n" unless $host eq 'localhost';
    print "graph_title System Uptime
graph_args --base 1000 -l 0
graph_vlabel uptime in days
graph_category system
graph_info This graph shows the number of days that the the host is up and running_
↳so far.
uptime.label uptime
uptime.info The system uptime itself in days.
uptime.draw AREA
";
    exit 0;
}

```

That makes the plugin configurable. Please notice that the plugin prints a *host_name* line if it’s not examining the localhost. This is the way Munin knows which device a non-local plugin is examining.

All cool and good. Now we need it to autoconfigure for each new SNMP agent you configure for. In general you would run this command:

```

# munin-node-configure --snmp --snmpversion 2 --snmpcommunity public 192.168.2.0/
↳24 | sh -x

```

This will interrogate all IP addresses in the given CIDR range with the given SNMP version and community string and then the script figures out which SNMP plugins will work on a device it finds and makes goes *ln -s snmp_switch_uptime /usr/share/munin/plugins/snmp__uptime* if the device known as “switch” supports the uptime plugin.

The way it does this is by knowing which plugins to talk to, and then by asking them what OIDs they are interested in:

```

=head1 MAGIC MARKERS

### family=snmpauto
### capabilities=snmpconf

```

```
...
=cut
...

if (defined $ARGV[0] and $ARGV[0] eq "snmpconf") {
    print "require 1.3.6.1.2.1.1.3.0 [0-9]\n"; # Number
    exit 0;
}
```

Given those magic markers `munin-node-configure` will run the plugin with the argument `snmpconf` which makes the plugin tell `munin-node-configure` what OIDs it requires for operation.

In a more complex case, `snmp_if_` more is needed to generate the needed symlinks:

```
# TODO
```

This section contains man pages and other reference material

Nomenclature

Nomenclature

To be able to use Munin, to understand the documentation, and - not to be neglected - to be able to write documentation that is consistent with Munin behavior, we need a common nomenclature.

Common terms

Term Munin Master

Explanation The central host/server where Munin gathers all data to. This machine runs munin-cron

Synonyms master

Term Munin Node

Explanation The daemon/network service running on each host to be contacted by the munin master to gather data. Each node may monitor several hosts. The Munin master will likely run a munin-node itself.

Synonyms In SNMP this might be called an agent.

Term Plugin

Explanation Each Munin node handles one or more plugins to monitor stuff on hosts

Synonym service

Term Host

Explanation A machine monitored by Munin, maybe by proxy on a munin node or via a snmp plugin

Synonym N/A

Term Field

Explanation Each plugin presents data from one or more data sources. Each found, read, calculated value corresponds to a field.attribute tuple

Synonym Data source

Term FQN

Explanation A *Fully Qualified Name* used to address a specific group, node, service or data source.

Synonym Fully Qualified Name

Term Attribute

Explanation Description found in output from plugins, both general (global) to the plugin and also specific to each field

Synonym N/A

Term Directive

Explanation Statements used in configuration files like munin.conf, munin-node.conf and plugin configuration directory (/etc/munin/plugin-conf.d/).

Synonym N/A

Term Environment variable

Explanation Set up by munin-node, used to control plugin behaviour, found in plugin configuration directory (/etc/munin/plugin-conf.d/)

Synonym N/A

Term Global (plugin) attributes

Explanation Used in the global context in a plugin's configuration output. NB: The attribute is considered "global" only to the plugin (and the node) and only when executed.

Synonym

Term Datasource-specific plugin attributes

Explanation Used in the datasource-specific context in a plugin's output.

Synonym N/A

Term Node-level directives

Explanation Used in munin.conf.

Synonym

Term Group-level directives

Explanation Used in munin.conf.

Synonym

Term Field-level directives

Explanation Used in munin.conf.

Synonym

Examples

To shed some light on the nomenclature, consider the examples below:

Global plugin attribute

Global plugin attributes are in the plugins output when run with the config argument. The full list of these attributes is found on the protocol config page. This output does not configure the plugin, it configures the plugins graph.

```
graph_title Load average
-----
|           \----- value
|           \----- attribute
```

Datasource specific plugin attribute

These are found both in the config output of a plugin and in the normal readings of a plugin. A plugin may provide data from one or more data sources. Each data source needs its own set of field.attribute tuples to define how the data source should be presented.

```
load.warning 100
-----
|   |   \- value
|   \----- one of several attributes used in config output
|   \----- field

load.value 54
-----
|   |   \- value
|   \----- only attribute when getting values from a plugin
|   \----- field
```

Configuration files

This one is from the global section of munin.conf:

```
dbdir          /var/lib/munin/
-----
|           \----- value
|           \----- global directive
```

And then one from the node level section:

```
[foo.example.org]
address localhost
-----
|           \----- value
|           \----- node level directive
```

The relation between directives and attributes

Attributes A plugin has a given set of data sources, and the data sources present themselves through a defined set of field.attributes with corresponding values. From a Munin administrator's point of view, these (the names of the fields and attributes) should not be changed as they are part of how the plugins work.

Directives The configuration files, however, are the administrator's domain. Here, the administrator may – through directives – control the plugins' behavior and even override the plugin's attributes if so desired. As such, directives (in configuration files) may override attributes (in plugins).

The distinction between *attributes* and *directives* defines an easily understandable separation between how the (for many people) shrink-wrapped plugins and the editable configuration files.

Fully Qualified Name

The Fully Qualified Name, or “FQN” is the address of a group, node, service, or a data source in munin.

It is most often used when configuring graphs which has no single corresponding plugin, but borrow data sources from other services.

Group FQN

The group FQN consists of one or more components, separated with a semicolon.

Note: If a node is configured without a group, the domain of the hostname becomes the group.

Examples

```
example.com
-----
|
| \--- group "example.com"
```

```
acme;webservers
-----
|   |
|   | \--- group "webservers"
|   \----- group "acme"
```

Node FQN

The fully qualified name to a node consists of a group FQN, a semicolon, and a hostname.

Examples:

```
example.com;foo.example.com
-----
|           |
|           | \--- node "foo.example.com"
|           \----- group "example.com"
```

```
acme;webservers;www1.example.net
-----
|   |           |
|   |           | \----- node "www1.example.net"
|   |           \----- group "webservers"
|   \----- group "acme"
```

Service FQN

The fully qualified name to a service consists of a node FQN, a colon, and a service name.

Note: A simple munin plugin will provide a service with the same name as the plugin. A multigraph plugin will provide one or more services, with arbitrary names.

Example:

Command line scripts

- *munin-node-configure* can automatically configure plugins for the local node.
- *munindoc* outputs plugin documentation.
- *munin-run* runs a plugin with the same environment as if run from *munin-node*. Very useful for debugging.
- *munin-async* is a command line utility, known as an “asynchronous proxy node”. *munin-update* can connect via ssh and run *munin-async* this to retrieve data from the munin async spool without waiting for the node to run plugins.

AUTHORS

Jimmy Olsen, Audun Ytterdal, Brian de Wolf, Nicolai Langfeldt

SEE ALSO

munin-update, *munin-limits*, *munin.conf*,

munin-async

DESCRIPTION

The munin async clients reads from a spool directory written by *munin-asyncd*.

It can optionally request a cleanup of this directory.

OPTIONS

--spooldir | **-s** <spooldir>

Directory for spooled data [/var/lib/munin/spool]

--hostname <hostname>

Overrides the hostname [The local hostname]

This is used to override the hostname used in the greeting banner. This is used when using munin-async from the munin master, and the data fetched is from another node.

--cleanup

Clean up the spooldir after interactive session completes

--cleanupandexit

Clean up the spooldir and exit (non-interactive)

--spoolfetch

Enables the “spool” capability [no]

--vectorfetch

Enables the “vectorized” fetching capability [no]

Note that without this flag, the “fetch” command is disabled.

--verbose | **-v**

Be verbose

--help | **-h**

View this message

EXAMPLES

```
munin-async --spoolfetch
```

This starts an interactive munin node session, enabling the “spoolfetch” command. This does not connect to the local munin node. Everything happens within munin-async, which reads from the spool directory instead of connecting to the node.

SEE ALSO

See *munin* for an overview over munin.

See also *Asynchronous proxy node* for more information and examples of how to configure munin-async.

munin-asyncd

DESCRIPTION

The munin async daemon connects to a *munin node* periodically, and requests plugin configuration and data.

This is stored in a spool directory, which is read by *munin-async*.

OPTIONS

--spool | **-s** <spooldir>
Directory for spooled data [/var/lib/munin/spool]

--host <hostname:port>
Connect a munin node running on this host name and port [localhost:4949]

--interval <seconds>
Set default interval size [86400 (one day)]

--retain <count>
Number of interval files to retain [7]

--nocleanup
Disable automated spool dir cleanup

--fork
Fork one thread per plugin available on the node. [no forking]

--verbose | **-v**
Be verbose

--help | **-h**
View this message

SEE ALSO

See *munin* for an overview over munin.

See also *Asynchronous proxy node* for more information and examples of how to configure munin-asyncd.

munin-check

DESCRIPTION

munin-check is a utility that fixes the permissions of the munin directories and files.

Note: munin-check needs superuser rights.

OPTIONS

- fix-permissions** | -f
Fix the permissions of the munin files and directories.
- help** | -h
Display usage information

SEE ALSO

See *munin* for an overview over munin.

munin-cron

DESCRIPTION

Munin-cron is a part of the package Munin, which is used in combination with *munin-node*.

Munin is a group of programs to gather data from Munin's nodes, graph them, create html-pages, and optionally warn Nagios about any off-limit values.

“munin-cron” runs the following programs, in the given order:

1. *munin-update*
2. *munin-limits*

Unless the munin master is configured otherwise, “munin-cron” should run every 5 minutes.

OPTIONS

- service** <service>
Limit services to <service>. Multiple **--service** options may be supplied. [unset]
- host** <host>
Limit hosts to <host>. Multiple **--host** options may be supplied. [unset]
- config** <file>
Use <file> as configuration file. [/etc/munin/munin.conf]

SEE ALSO

See *munin* for an overview over munin.

munin-update, *munin-limits*, *munin.conf*,

munin-httpd

DESCRIPTION

munin-httpd is a basic webserver. It provides a munin web interface on port 4948/tcp, generating pages and graphs on demand.

If transport layer security and authentication is desired, place a webserver with those features as a reverse proxy in front of munin-httpd.

munin-httpd replaces the FastCGI scripts `munin-cgi-graph` and `munin-cgi-html`.

OPTIONS

munin-httpd takes no options.

CONFIGURATION

munin-httpd reads its configuration from `munin.conf`.

SEE ALSO

See `munin` for an overview over munin.

`munin.conf`

munin-limits

DESCRIPTION

`munin-limits` is one of the processes regularly run from the `munin-cron` script.

It reads the current and the previous collected values for each plugin, and compares them to the plugin's warning and critical values, if it has any.

If the limits are breached, for instance, if a value moves from "ok" to "warning", or from "critical" to "ok", it sends an event to any configured contacts.

A common configured contact is "nagios", which can use events from munin-limits as a source of passive service check results.

OPTIONS

--config <file>

Use <file> as configuration file. [/etc/munin/munin.conf]

--contact <contact>

Limit contacts to those of <contact>. Multiple `--contact` options may be supplied. [unset]

--host <host>

Limit hosts to those of <host>. Multiple `--host` options may be supplied. [unset]

--service <service>

Limit services to those of <service>. Multiple `--service` options may be supplied. [unset]

--always-send <severity list>

Force sending of messages even if you normally wouldn't.

The <severity list> can be a whitespace or comma separated list of the values "ok", "warning", "critical" or "unknown".

This option may be specified several times, to add more values.

Use of "--always-send" overrides the "always_send" value in munin.conf for configured contacts. See also `-force`.

--force

Alias for "--always-send ok,warning,critical,unknown"

--force-run-as-root

munin-limits will normally prevent you from running as root. Use this option to override this.

The use of this option is not recommended. You may have to clean up file permissions in order for munin to run normally afterwards.

--help

View help message.

--debug

Log debug messages.

--screen

If set, log messages to STDERR on the screen.

FILES

/etc/munin/munin.conf

*/var/lib/munin/**

*/var/run/munin/**

SEE ALSO

See *munin* for an overview over munin.

munin.conf

munin-node-configure

SYNOPSIS

munin-node-configure [options]

DESCRIPTION

munin-node-configure reports which plugins are enabled on the current node, and suggest changes to this list.

By default this program shows which plugins are activated on the system.

If you specify "--suggest", it will present a table of plugins that will probably work (according to the plugins' autoconf command).

If you specify "--snmp", followed by a list of hosts, it will present a table of SNMP plugins that they support.

If you additionally specify "--shell", shell commands to install those same plugins will be printed. These can be reviewed or piped directly into a shell to install the plugins.

OPTIONS

General options

--help

Show this help page.

--version

Show version information.

--debug

Print debug information on the operations of “munin-node-configure”. This can be very verbose.

All debugging output is printed to STDOUT, and each line is prefixed with ‘#’. Only errors are printed to STDERR.

--pidebug

Plugin debug. Sets the environment variable MUNIN_DEBUG to 1 so that plugins may enable debugging.

--config <file>

Override configuration file [/etc/munin/munin-node.conf]

--servicedir <dir>

Override plugin directory [/etc/munin/plugins/]

--sconfdir <dir>

Override plugin configuration directory [/etc/munin/plugin-conf.d/]

--libdir <dir>

Override plugin library [/usr/share/munin/plugins/]

--exitnoterror

Do not consider plugins that exit non-zero exit-value as error.

--suggest

Suggest plugins that might be added or removed, instead of those that are currently enabled.

Output options

By default, “munin-node-configure” will print out a table summarising the results.

--shell

Instead of a table, print shell commands to install the new plugin suggestions.

This implies “--suggest”, unless “--snmp” was also enabled. By default, it will not attempt to remove any plugins.

--remove-also

When “--shell” is enabled, also provide commands to remove plugins that are no longer applicable from the service directory.

Plugin selection options

--families <family, ...>

Override the list of families that will be used (auto, manual, contrib, snmpauto). Multiple families can be specified as a comma-separated list, by repeating the “--families” option, or as a combination of the two.

When listing installed plugins, the default families are ‘auto’, ‘manual’ and ‘contrib’. Only ‘auto’ plugins are checked for suggestions. SNMP probing is only performed on ‘snmpauto’ plugins.

--newer <version>

Only consider plugins added to the Munin core since <version>. This option is useful when upgrading,

since it can prevent plugins that have been manually removed from being reinstalled. This only applies to plugins in the 'auto' family.

SNMP options

--snmp <host|cidr,...>

Probe the SNMP agents on the host or CIDR network (e.g. "192.168.1.0/24"), to see what plugins they support. This may take some time, especially if the many hosts are specified.

This option can be specified multiple times, or as a comma-separated list, to include more than one host/CIDR.

--snmpversion <ver>

The SNMP version (1, 2c or 3) to use. ['2c']

--snmpport <port>

The SNMP port to use [161]

--snmpdomain <domain>

The Transport Domain to use for exchanging SNMP messages. The default is UDP/IPv4. Possible values: 'udp', 'udp4', 'udp/ipv4'; 'udp6', 'udp/ipv6'; 'tcp', 'tcp4', 'tcp/ipv4'; 'tcp6', 'tcp/ipv6'.

SNMP 1/2c authentication

SNMP versions 1 and 2c use a "community string" for authentication. This is a shared password, sent in plaintext over the network.

--snmpcommunity <string>

The community string for version 1 and 2c agents. ['public'] (If this works your device is probably very insecure and needs a security checkup).

SNMP 3 authentication

SNMP v3 has three security levels. Lowest is "noAuthNoPriv", which provides neither authentication nor encryption. If a username and "authpassword" are given it goes up to "authNoPriv", and the connection is authenticated. If "privpassword" is also given the security level becomes "authPriv", and the connection is authenticated and encrypted.

Note: Encryption can slow down slow or heavily loaded network devices. For most uses "authNoPriv" will be secure enough – the password is sent over the network encrypted in any case.

ContextEngineIDs are not (yet) supported.

For further reading on SNMP v3 security models please consult RFC3414 and the documentation for Net::SNMP.

--snmpusername <name>

Username. There is no default.

--snmpauthpass <password>

Authentication password. Optional when encryption is also enabled, in which case defaults to the privacy password ("--snmpprivpass").

--snmpauthproto <protocol>

Authentication protocol. One of 'md5' or 'sha' (HMAC-MD5-96, RFC1321 and SHA-1/HMAC-SHA-96, NIST FIPS PIB 180, RFC2264). ['md5']

--snmpprivpass <password>

Privacy password to enable encryption. There is no default. An empty ("") password is considered as no password and will not enable encryption.

Privacy requires a `privprotocol` as well as an `authprotocol` and a `authpassword`, but all of these are defaulted (to `'des'`, `'md5'`, and the `privpassword` value, respectively) and may therefore be left unspecified.

--snmpprivproto <protocol>

If the `privpassword` is set this setting controls what kind of encryption is used to achieve privacy in the session. Only the very weak `'des'` encryption method is supported officially. [`'des'`]

`munin-node-configure` also supports `'3des'` (CBC-3DES-EDE, aka Triple-DES, NIST FIPS 46-3) as specified in IETF draft-reeder-snmpv3-usm-3desede. Whether or not this works with any particular device, we do not know.

FILES

- `/etc/munin/munin-node.conf`
- `/etc/munin/plugin-conf.d/*`
- `/etc/munin/plugins/*`
- `/usr/share/munin/plugins/*`

SEE ALSO

See *munin* for an overview over munin.

munin-node

DESCRIPTION

`munin-node` is a daemon for reporting statistics on system performance.

By default, it is started at boot time, listens on port 4949/TCP, accepts connections from the *munin master*, and runs *munin plugins* on demand.

OPTIONS

--config <configfile>

Use <file> as configuration file. [`/etc/munin/munin-node.conf`]

--paranoia

Only run plugins owned by root. Check permissions as well. Can be negated with `--no-paranoia` [`--no-paranoia`]

--help

View this help message.

--debug

View debug messages.

Note: This can be very verbose.

--pidebug

Plugin debug. Sets the environment variable `MUNIN_DEBUG` to 1 so that plugins may enable debugging.

CONFIGURATION

The configuration file is *munin-node.conf*.

FILES

/etc/munin/munin-node.conf
*/etc/munin/plugins/**
*/etc/munin/plugin-conf.d/**
/var/run/munin/munin-node.pid
/var/log/munin/munin-node.log

SEE ALSO

munin-node.conf

Example configuration

```
# /etc/munin/munin-node.conf - config-file for munin-node
#

host_name random.example.org
log_level 4
log_file /var/log/munin/munin-node.log
pid_file /var/run/munin/munin-node.pid
background 1
setsid 1

# Which port to bind to;

host [::]
port 4949
user root
group root

# Regexp for files to ignore

ignore_file ~$
ignore_file \.bak$
ignore_file %$
ignore_file \.dpkg-(tmp|new|old|dist)$
ignore_file \.rpm(save|new)$
ignore_file \.puppet-bak$

# Hosts to allow

cidr_allow 127.0.0.0/8
cidr_allow 192.0.2.129/32
```

SEE ALSO

See *munin* for an overview over munin.

munin-run

DESCRIPTION

munin-run is a script to run Munin plugins from the command-line.

It is primarily used to debug plugins; `munin-run` runs these plugins in the same conditions as they are under `munin-node`.

OPTIONS

- config** <configfile>
Use <file> as configuration file. [/etc/munin/munin-node.conf]
- servicedir** <dir>
Use <dir> as plugin dir. [/etc/munin/plugins/]
- sconfdir** <dir>
Use <dir> as plugin configuration dir. [/etc/munin/plugin-conf.d/]
- sconfdir** <file>
Use <file> as plugin configuration. Overrides sconfdir. [undefined]
- paranoia**
Only run plugins owned by root and check permissions. [disabled]
- help**
View this help message.
- debug**
Print debug messages.

Debug messages are sent to STDOUT and are prefixed with “#” (this makes it easier for other parts of munin to use munin-run and still have `-debug` on). Only errors go to STDERR.
- pidebug**
Enable debug output from plugins. Sets the environment variable `MUNIN_DEBUG` to 1 so that plugins may enable debugging. [disabled]
- version**
Show version information.

FILES

/etc/munin/munin-node.conf
*/etc/munin/plugins/**
*/etc/munin/plugin-conf.d/**
/var/run/munin/munin-node.pid
/var/log/munin/munin-node.log

SEE ALSO

See *munin* for an overview over munin.

munin-update

DESCRIPTION

`munin-update` is the primary Munin component. It is run from the *munin-cron* script.

This script is responsible for contacting all the agents (munin-nodes) and collecting their data. Upon fetching the data, `munin-update` stores everything in RRD files - one RRD files for each field in each plugin.

Running `munin-update` with the `-debug` flag will often give plenty of hints on what might be wrong.

munin-update is a component in the Munin server.

OPTIONS

- config_file** <file>
Use <file> as the configuration file. [/etc/munin/munin.conf]
- debug**
Log debug messages.
- screen**
If set, log messages to STDERR on the screen.
- fork**
If set, will fork off one process for each host. Can be negated with `--nofork` [`--fork`]
- host** <host>
Limit fetched data to those from <host>. Multiple `--host` options may be supplied. [unset]
- service** <service>
Limit fetched data to those of <service>. Multiple `--service` options may be supplied. [unset]
- timeout** <seconds>
Set the network timeout to <seconds>. [180]
- help**
Print the help message then exit.
- version**
Print version information then exit.

SEE ALSO

See *munin* for an overview over munin.

munin-cron

munin.conf

DESCRIPTION

This is the configuration file for the munin master. It is used by *munin-update* and *munin-limits*

Note: All global directives have to be defined in the first section of the file! It will not work if you place them in later sections of the config file. We recommend to use the delivered munin.conf file and adapt it to your needs.

GLOBAL DIRECTIVES

Global directives affect all munin master components unless specified otherwise.

- dbdir** <path>
The directory where munin stores its database files. Default: /var/lib/munin
- logdir** <path>
The directory where munin stores its logfiles. Default: /var/log/munin
- rundir** <path>
Directory for files tracking munin's current running state. Default: /var/run/munin

tmpldir <path>

Directories for templates used by *munin-htp*d to generate HTML pages. Default /etc/munin/templates

fork <yes|no>

This directive determines whether *munin-update* fork when gathering information from nodes. Default is “yes”.

If you set it to “no” *munin-update* will collect data from the nodes in sequence. This will take more time, but use less resources. Not recommended unless you have only a handful of nodes.

Affects: *munin-update*

timeout <seconds>

This directive determines how long *munin-update* allows a worker to fetch data from a single node. Default is “180”.

Affects: *munin-update*

palette <default|old>

The palette used by *munin-htp*d to color the graphs. The “default” palette has more colors and better contrast than the “old” palette.

custom_palette rrggbb rrggbb ...

The user defined custom palette used by *munin-htp*d to color the graphs. This option overrides the existing palette. The palette must be space-separated 24-bit hex color code.

graph_data_size <normal|huge>

This directive sets the resolution of the RRD files that are created by *munin-update*.

Default is “normal”.

“huge” saves the complete data with 5 minute resolution for 400 days.

Changing this directive has no effect on existing graphs

contact.<contact name>.command <command>

Define which contact command to run.

contact.<contact name>.text <text>

Text to pipe into the command.

contact.<contact name>.max_messages <number>

Close (and reopen) command after given number of messages. E.g. if set to 1 for an email target, Munin sends 1 email for each warning/critical. Useful when relaying messages to external processes that may handle a limited number of simultaneous warnings.

ssh_command <command>

The name of the secure shell command to use. Can be fully qualified or looked up in \$PATH.

Defaults to “ssh”.

ssh_options <options>

The options for the secure shell command.

Defaults are “-o ChallengeResponseAuthentication=no -o StrictHostKeyChecking=no”. Please adjust this according to your desired security level.

With the defaults, the master will accept and store the node ssh host keys with the first connection. If a host ever changes its ssh host keys, you will need to manually remove the old host key from the ssh known hosts file. (with: *ssh-keygen -R <node-hostname>*, as well as *ssh-keygen -R <node-ip-address>*)

You can remove “StrictHostKeyChecking=no” to increase security, but you will have to manually manage the known hosts file. Do so by running “*ssh <node-hostname>*” manually as the munin user, for each node, and accept the ssh host keys.

If you would like the master to accept all node host keys, even when they change, use the options “-o UserKnownHostsFile=/dev/null -o StrictHostKeyChecking=no -o PreferredAuthentications=publickey”.

NODE DEFINITIONS

Node definitions can have several types. In all forms, the definition is used to generate the node name and group for the node, and the following lines define its directives. All following directives apply to that node until another node definition or EOF.

When defining a nodename it is vital that you use a standard DNS name, as in, one that uses only “a-z”, “-”, and “.”. While other characters can be used in a DNS name, it is against the RFC, and Munin uses the other characters as delimiters. If they appear in nodenames, unexpected behavior may occur.

The simplest node definition defines the section for a new node by simply wrapping the DNS name of the node in brackets, e.g. `[machine1.example.com]`. This will add the node `machine1.example.com` to the group `example.com`.

The next form of definition is used to define the node and group explicitly. It follows the form `[example.com; machine1.sub.example.com]`. This adds the node `machine1.sub.example.com` to the group `example.com`. This can be useful if you have machines you want to put together as a group that are under different domains (as in the given example). This can also solve a problem if your machine is `example.com`, where having a group of `com` makes little sense.

A deeper hierarchy can be specified by using a list of groups, separated with “;”. For example: `[site1; customer2; production; mail.customer2.example.org]`.

NODE DIRECTIVES

These are directives that can follow a node definition and will apply only to that node.

address <value>

Specifies the host name or IP address, with an optional scheme.

Permitted schemes are “munin://”, “ssh://” or “cmd://”. If no scheme is specified, the default is “munin://”

The “ssh://” and “cmd://” schemes take arguments after the URL. See [Address schemes](#) for examples.

port <port number>

The port number of the node. Ignored if using alternate transport. Default is “4949”.

local_address <address>

The local address to connect to the node from. This overrides a group or global directive.

use_node_name <yes|no>

Overrides the name supplied by the node. Allowed values: “yes” and “no”. Defaults to “no”.

contacts <no|contact ...>

A list of contacts used by `munin-limits` to report values passing the warning and critical thresholds.

If set to something else than “no”, names a list of contacts which should be notified for this node. Default is “no”.

notify_alias <node name>

Used by `munin-limits`.

If set, changes the name by which the node presents itself when warning through `munin-limits`.

ignore_unknown <yes|no>

If set, ignore any unknown values reported by the node. Allowed values are “yes” and “no”. Defaults to “no”.

Useful when a node is expected to be off-line frequently.

update <yes|no>

Fetch data from this node with `munin-update`? Allowed values are “yes” and “no”. Defaults to “yes”.

If you make a virtual node which borrow data from real nodes for aggregate graphs, set this to “no” for that node.

PLUGIN DIRECTIVES

These directives follow a node definition and are of the form “plugin.directive <value>”.

Using these directives you can override various directives for a plugin, such as its contacts, and can also be used to create graphs containing data from other plugins.

FIELD DIRECTIVES

These directives follow a node definition and are of the form “plugin.field <value>”.

Using these directives you can override values originally set by plugins on the nodes, such as warning and critical levels or graph names.

graph_height <value>

The graph height for a specific service. Default is 175. Affects: *munin-httpd*.

graph_width <value>

The graph width for a specific service. Default is 400. Affects: *munin-httpd*.

warning <value>

The value at which munin-limits will mark the service as being in a warning state. Value can be a single number to specify a limit that must be passed or they can be a comma separated pair of numbers defining a valid range of values. Affects: *munin-limits*.

critical <value>

The value at which munin-limits will mark the service as being in a critical state. Value can be a single number to specify a limit that must be passed or they can be a comma separated pair of numbers defining a valid range of values Affects: *munin-limits*.

EXAMPLES

Three nodes

A minimal configuration file, using default settings for everything, and specifying three nodes.

```
[mail.example.com]
address mail.example.com

[web.example.com]
address web.example.com

[munin.example.com]
address localhost
```

Virtual node

A virtual node definition. Disable update, and make a graph consisting of data from other graphs.

```
[example.com;Totals]
update no
load.graph_title Total load
load.sum_load.label load
load.sum_load.special_stack mail=mail.example.com web=web.example.com munin=munin.
↔example.com
```

Address schemes

The scheme tells munin how to connect to munin nodes.

The `munin://` scheme is default, if no scheme is specified. By default, Munin will connect to the munin node with TCP on port 4949.

The following examples are equivalent:

```
# master: /etc/munin/munin.conf.d/node.example.conf
[mail.site2.example.org]
address munin://mail.site2.example.org

[mail.site2.example.org]
address munin://mail.site2.example.org:4949

[mail.site2.example.org]
address mail.site2.example.org

[mail.site2.example.org]
address mail.site2.example.org
port 4949
```

To connect to a munin node through a shell command, use the `cmd://` prefix.

```
# master: /etc/munin/munin.conf.d/node.example.conf
[mail.site2.example.org]
address cmd:///usr/bin/munin-async [...]
```

To connect through ssh, use the `ssh://` prefix.

```
# master: /etc/munin/munin.conf.d/node.example.conf
[mail.site2.example.org]
address ssh://bastion.site2.example.org/bin/nc mail.site2.example.org 4949

[www.site2.example.org]
address ssh://bastion.site2.example.org/bin/nc www.site2.example.org 4949
```

Note: When using the `ssh://` transport, you can configure how ssh behaves by editing `~munin/.ssh/config`. See the *ssh transport configuration examples*.

SEE ALSO

See *munin* for an overview over munin.

Examples for ssh transport

munin-node.conf

DESCRIPTION

This is the configuration file for *munin-node* and *munin-run*.

The directives “`host_name`”, “`paranoia`” and “`ignore_file`” are munin node specific.

All other directives in `munin-node.conf` are passed through to the Perl module `Net::Server`. Depending on the version installed, you may have different settings available.

DIRECTIVES

Native

host_name

The hostname used by munin-node to present itself to the munin master. Use this if the local node name differs from the name configured in the munin master.

ignore_file

Files to ignore when locating installed plugins. May be repeated.

paranoia

If set to a true value, *munin-node* will only run plugins owned by root.

Inherited

These are the most common Net::Server options used in *munin-node*.

log_level

Ranges from 0-4. Specifies what level of error will be logged. “0” means no logging, while “4” means very verbose. These levels correlate to syslog levels as defined by the following key/value pairs. 0=err, 1=warning, 2=notice, 3=info, 4=debug.

Default: 2

log_file

Where the munin node logs its activity. If the value is Sys::Syslog, logging is sent to syslog

Default: undef (STDERR)

pid_file

The pid file of the process

Default: undef (none)

background

To run munin node in background set this to “1”. If you want munin-node to run as a foreground process, comment this line out and set “setsid” to “0”.

user

The user munin-node runs as

Default: root

group

The group munin-node runs as

Default: root

setsid

If set to “1”, the server forks after binding to release itself from the command line, and runs the POSIX::setsid() command to daemonize.

Default: undef

global_timeout

munin-node holds the connection to Munin master only a limited number of seconds to get the requested operation finished. If the time runs out the node will close the connection.

Timeout for the whole transaction. Units are in sec.

Default: 900 seconds (15 min)

timeout

This is the timeout for each plugin. If plugins take longer to run, this will disconnect the master.

Default: 60 seconds

allow

A regular expression defining which hosts may connect to the munin node.

Note: Use `cidr_allow` if available.

cidr_allow

Allowed hosts given in CIDR notation (192.0.2.1/32). Replaces or complements “allow”. Requires the precense of Net::Server, but is not supported by old versions of this module.

cidr_deny

Like `cidr_allow`, but used for denying host access

host

The IP address the munin node process listens on

Default: * (All interfaces)

port

The TCP port the munin node listens on

Default: 4949

EXAMPLE

A pretty normal configuration file:

```
#
# Example config-file for munin-node
#

log_level 4
log_file /var/log/munin-node/munin-node.log
pid_file /var/run/munin/munin-node.pid

background 1
setsid 1

user root
group root

# This is the timeout for the whole transaction.
# Units are in sec. Default is 15 min
#
# global_timeout 900

# This is the timeout for each plugin.
# Units are in sec. Default is 1 min
#
# timeout 60

# Regexps for files to ignore
ignore_file [#~]$
ignore_file DEADJOE$
ignore_file \.bak$
ignore_file %$
ignore_file \.dpkg-(tmp|new|old|dist)$
ignore_file \.rpm(save|new)$
ignore_file \.pod$

# Set this if the client doesn't report the correct hostname when
# telnetting to localhost, port 4949
#
```

```

host_name localhost.localdomain

# A list of addresses that are allowed to connect. This must be a
# regular expression, since Net::Server does not understand CIDR-style
# network notation unless the perl module Net::CIDR is installed. You
# may repeat the allow line as many times as you'd like

allow ^127\.0\.0\.1$
allow ^::1$

# If you have installed the Net::CIDR perl module, you can use one or more
# cidr_allow and cidr_deny address/mask patterns. A connecting client must
# match any cidr_allow, and not match any cidr_deny. Note that a netmask
# *must* be provided, even if it's /32
#
# Example:
#
# cidr_allow 127.0.0.1/32
# cidr_allow 192.0.2.0/24
# cidr_deny 192.0.2.42/32

# Which address to bind to;
host *
# host 127.0.0.1

# And which port
port 4949

```

SEE ALSO

See [munin](#) for an overview over munin.

[munin-node](#), [munin-run](#)

munindoc

SYNOPSIS

munindoc <plugin>

DESCRIPTION

This program displays Munin plugin documentation.

The documentation for plugins includes include basic usage information, how it can be configured, and how the output can be interpreted.

Additional information typically found is usually the name of the plugin author, licensing and “magic markers” which controls plugin auto configuration (done by [munin-node-configure](#)).

NOTES

Most Munin commands (such as [munin-run](#), and munindoc itself) is only documented through the usual Unix man command.

Note that not all plugins are documented yet. If you want to contribute plugin documentation, take a look at the [munindoc instruction page](#) in our Trac wiki

SEE ALSO

See *munin* for an overview over munin.

Examples and Templates

Examples

Examples of munin and related configuration are gathered here.

Apache virtualhost configuration

This example describes how to set up munin on a separate virtual host using apache httpd. It proxies all requests to *munin-httpd*

Virtualhost configuration

Add a new virtualhost, using the following example:

```
<VirtualHost *:80>
    ServerName munin.example.org
    ServerAlias munin

    ServerAdmin info@example.org

    DocumentRoot /srv/www/munin.example.org

    ErrorLog /var/log/apache2/munin.example.org-error.log
    CustomLog /var/log/apache2/munin.example.org-access.log combined

    RewriteEngine On
    RewriteRule ^/(.*\.html)$ /srv/www/munin.example.org/$1 [L]

    # Proxy everything to munin-httpd
    ProxyPass / http://localhost:4948/
    ProxyPassReverse / http://localhost:4948/
</VirtualHost>
```

lighttpd configuration

This example describes how to set use lighttpd in front of munin-httpd.

Webserver configuration

```
alias.url += ( "/munin-static" => "/etc/munin/static" )
alias.url += ( "/munin" => "/var/cache/munin/www/" )

$http["url"] =~ "^/munin" {
    proxy.server = ( "" => (( "host" => "127.0.0.1", "port" => 4948)) )
}
```

nginx configuration

This example describes how to set up nginx for serving content generated by munin.

This document describes two alternative configurations:

1. serving static graphs and HTML generated by munin-cron
2. proxy traffic to munin-httpd

Serving cron-made graphs and HTML

Nginx is quite good at serving static files, and as such the configuration is mostly in place already.

The paths are as in use on a Debian Linux system. Add the following to `/etc/nginx/sites-enabled/default`:

```
location /munin/static/ {
    alias /etc/munin/static/;
    expires modified +1w;
}

location /munin/ {
    auth_basic "Restricted";
    # Create the htpasswd file with the htpasswd tool.
    auth_basic_user_file /etc/nginx/htpasswd;

    alias /var/cache/munin/www/;
    expires modified +310s;
}
```

For Zoom to work you also need to add the following to the same configuration file:

```
location ^~ /munin-cgi/munin-cgi-graph/ {
    access_log off;
    fastcgi_split_path_info ^(/munin-cgi/munin-cgi-graph) (.*)
    fastcgi_param PATH_INFO $fastcgi_path_info;
    fastcgi_pass unix:/var/run/munin/fcgi-graph.sock;
    include fastcgi_params;
}
```

If this is a dedicated Munin server, you might want to redirect the front page as well:

```
location / {
    rewrite ^/$ munin/ redirect; break;
}
```

Using munin-httpd

You can use `nginx` as a proxy in front of `munin-httpd`.

This enables you to add `transport layer security` and `http authentication` (not included in this example).

```
location /munin/static/ {
    alias /etc/munin/static/;
}

location /munin/ {
    proxy_pass http://localhost:4948/;
}
```

Upstart configuration for rrdcached

This example sets up a dedicated rrdcached instance for munin.

If rrdcached stops, it is restarted.

A pre-start script ensures we have the needed directories

A post-start script adds permissions for the munin fastcgi process. This assumes that your fastcgi graph process is running as the user “www-data”, and that the file system is mounted with “acl”.

```
description "munin instance of rrdcached"
author "Stig Sandbeck Mathisen <ssm@fnord.no>"

start on filesystem
stop on runlevel [!2345]

# respawn
umask 022

pre-start script
  install -d -o munin -g munin -m 0755 /var/lib/munin/rrdcached-journal
  install -d -o munin -g munin -m 0755 /run/munin
end script

script
  start-stop-daemon \
    --start \
    --chuid munin \
    --exec /usr/bin/rrdcached \
    --pidfile /run/munin/rrdcached.pid \
    -- \
    -g \
    -p /run/munin/rrdcached.pid \
    -B -b /var/lib/munin/ \
    -F -j /var/lib/munin/rrdcached-journal/ \
    -m 0660 -l unix:/run/munin/rrdcached.sock \
    -w 1800 -z 1800 -f 3600
end script

post-start script
  sleep 1
  setfacl -m u:www-data:rw /run/munin/rrdcached.sock
end script
```

Systemd configuration for rrdcached

This example sets up a dedicated rrdcached instance for munin.

If rrdcached stops, it is restarted.

A pre-start script ensures we have the needed directories

A post-start script adds permissions for the munin fastcgi process. This assumes that your fastcgi graph process is running as the user “www-data”, and that the file system is mounted with “acl”.

```
[Unit]
Description=Munin rrdcached

[Service]
Restart=always
User=munin
PermissionsStartOnly=yes
```



```

ExecStartPre=/usr/bin/install -d -o munin -g munin -m 0755 \
/var/lib/munin/rrdcached-journal /run/munin
ExecStart=/usr/bin/rrdcached \
-g -B -b /var/lib/munin/ \
-p /run/munin/munin-rrdcached.pid \
-F -j /var/lib/munin/rrdcached-journal/ \
-m 0660 -l unix:/run/munin/rrdcached.sock \
-w 1800 -z 1800 -f 3600
ExecStartPost=/bin/sleep 1 ; /bin/setfacl -m u:www-data:rw /run/munin/rrdcached.
↪sock
[Install]
WantedBy=multi-user.target

```

Recommended graph_args

Set arguments for the rrd grapher with attribute *graph_args*. This is used to control how the generated graph looks, and how values are interpreted or presented.

You can override plugin defaults on Munin master via your own settings on plugin level in *munin.conf*.

See [rrdgraph man page](#) for more details.

Scale

--logarithmic

Plot these values on a logarithmic scale. Should almost never be used, but probably more often than we do now. Logarithmic scale is very useful when the collected values spans more than one to two magnitudes. It makes it possible to see the different small values as well as the different large values - instead of just the large values as usual.

Logarithmic has been tested on netstat (connection count) and some other graphs with good results.

Units

See [Best Current Practices](#)

--base <value>

Set to **1024** for things that are counted in binary units, such as memory (but not network bandwidth)

Set to **1000** for default SI units

--units-exponent <value>

Set to **3** force display unit to K, **-6** would force display in u/micro.

Axis

--lower-limit <value>

Start the Y-axis at value e.g. `--lower-limit 0` (also seen as: `-l 0`)

--upper-limit <value>

Set value to **100** for percentage graphs, ends the Y-axis at 100 (also seen as: `-u 100`)

--rigid

Force rrdgraph y-axis scale to the set upper and lower limit. Usually, the graph scale can *overflow*. (also seen as: `-r`)

Graph aggregation by example

This example covers creating aggregate graphs. The configuration reads the current and power from two UPSes (i.e. two hosts with two plugins each) and then creates one virtual host with two virtual plugins; one for current and one for power.

Plugins involved

The example uses a plugin for monitoring UPSes through SNMP, where the UPS address and the different aspects are defined through symlinks. The two UPSes, called “ups-5a” and “ups-5b”, are monitored with respect to “current” and “power”. Thus, the affected plugins are called as:

```
snmp_ups_ups-5a_current
snmp_ups_ups-5b_current
snmp_ups_ups-5a_power
snmp_ups_ups-5b_power
```

The original plugin name is actually “snmp_ups__” - note the “two” underscores at the end. The plugin is then symlinked to the given host name(s) (e.g. ups-5a) and what we want to monitor (e.g. power). Let’s just take one closer look at one of them:

```
snmp_ups_ups-5a_power
-----
|           |           |
|           |           |--- The function we want to monitor
|           |----- The node name of the UPS
|----- The plugin
```

Extract from munin.conf

The following extract from /etc/munin/munin.conf is explained in detail, step by step, below the configuration.

```
1 [UPS;ups-5a]
2     address 127.0.0.1 # localhost fetches data
3
4 [UPS;ups-5b]
5     address 127.0.0.1 # localhost fetches data
6
7 [UPS;Aggregated]
8     update no
9     contacts no
10
11     snmp_ups_current.update no
12     snmp_ups_current.graph_args --base 1000 -l 0
13     snmp_ups_current.graph_category UPS
14     snmp_ups_current.graph_title Aggregated input/output current
15     snmp_ups_current.graph_vlabel Ampere
16     snmp_ups_current.inputtotal.label Input current
17     snmp_ups_current.outputtotal.label Output current
18     snmp_ups_current.graph_order inputtotal outputtotal
19     snmp_ups_current.inputtotal.sum \
20         ups-5a:snmp_ups_ups-5a_current.inputcurrent \
21         ups-5b:snmp_ups_ups-5b_current.inputcurrent
22     snmp_ups_current.outputtotal.sum \
23         ups-5a:snmp_ups_ups-5a_current.outputcurrent \
24         ups-5b:snmp_ups_ups-5b_current.outputcurrent
25
26     snmp_ups_power.update no
27     snmp_ups_power.graph_args --base 1000 -l 0
```

```

28     snmp_ups_power.graph_category UPS
29     snmp_ups_power.graph_title Aggregated output power
30     snmp_ups_power.graph_vlabel Watts
31     snmp_ups_power.output.label Output power
32     snmp_ups_power.graph_order output
33     snmp_ups_power.output.sum \
34         ups-5a:snmp_ups_ups-5a_power.outputpower \
35         ups-5b:snmp_ups_ups-5b_power.outputpower

```

Explanations, per line

- 1 - 2: The SNMP-based plugin for the UPS known as “ups-5a” is defined. The group name is “UPS” and the node name is “ups-5a”. The plugin is run from localhost.
- 4 - 5: The SNMP-based plugin for the UPS known as “ups-5b” is defined. The group name is “UPS” and the node name is “ups-5b”. The plugin is run from localhost.
- 7: The group and “virtual node name” for the aggregated graphs are defined. The group name is “UPS” and the virtual node name is “Aggregated”.
- 8: Make sure that Munin (specifically, “munin-update”) does not try to actively gather information for this node.
- 9: Tell “munin-limits” not to send alerts if any limit is breached.

The above lines (1 - 9) have now established the fundament for three different graph pages; one for each of the two UPSes and one for the aggregate graphs.

- 11 - 15: Define the basic information for the virtual plugin for aggregated current. Note that “snmp_ups_current” is the virtual plugin’s name.
- 16 - 17: Simultaneously define and label “two” values to be graphed in the virtual plugin: “inputtotal” and “outputtotal”.
- 18: Order the values.
- 19 - 21: Calculate the value for “inputtotal” by reading the “inputcurrent” values from each of the two UPSes.

Let’s take a closer look at the components

```

snmp_ups_current.inputtotal.sum \
-----
|           |           |
|           |           |-- The sum mechanism
|           |           `----- One of this virtual plugin's values
|           |           `----- The name of the virtual plugin

```

```

ups-5a:snmp_ups_ups-5a_current.inputcurrent \
ups-5b:snmp_ups_ups-5b_current.inputcurrent
-----
|           |           |
|           |           |           `----- The "inputcurrent" value from the_
↔real plugin           |           `----- The real plugin's name (symlink)
|           |           |           `----- The host name from which to seek_
↔information           |

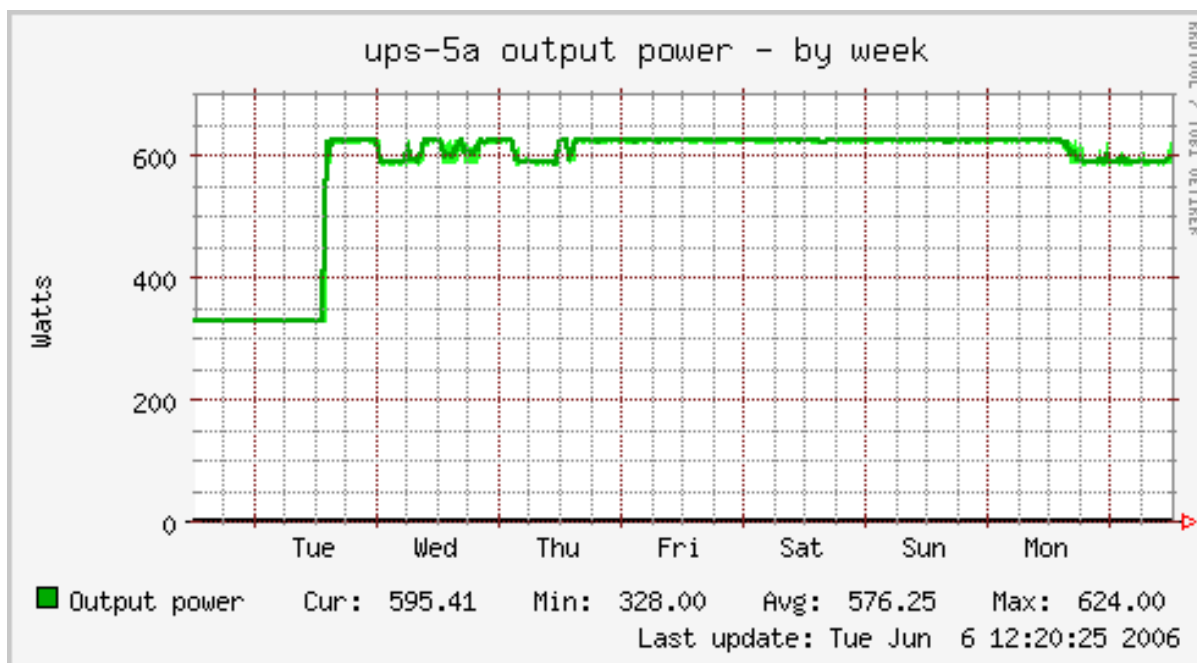
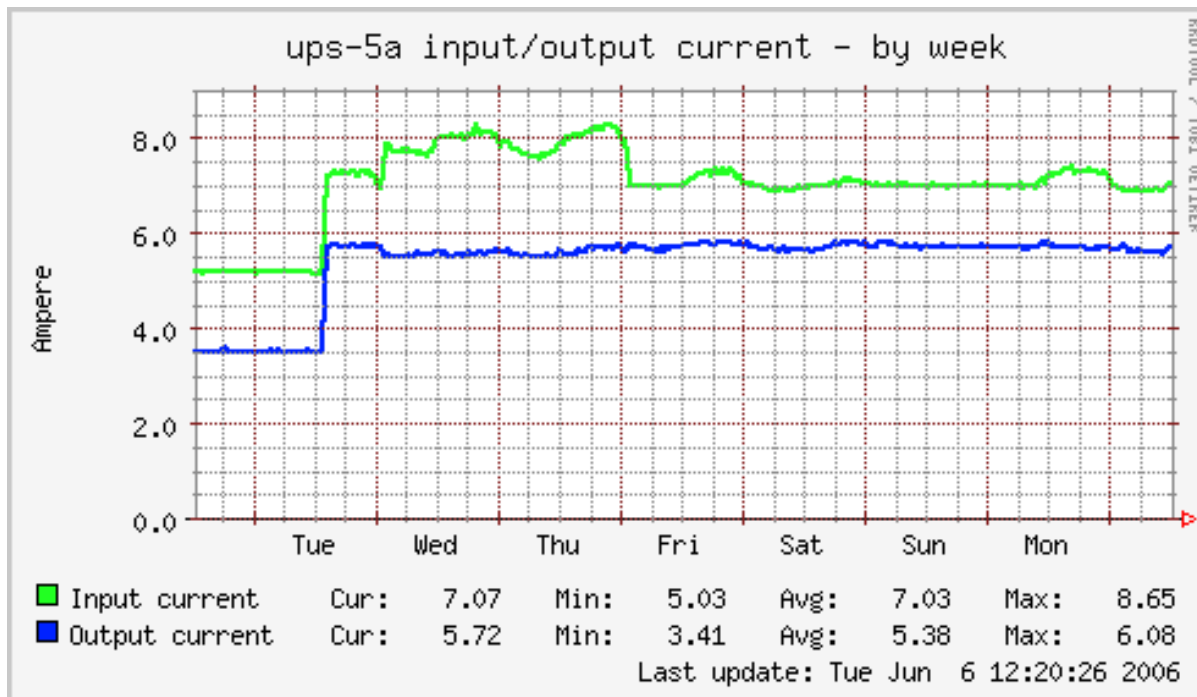
```

- 22 - 24: Similarly for “outputtotal”.
- 26 - 35: Like the above, but for power instead. Note that this virtual plugin graphs only “one” value, and as such, only “one” “sum” mechanism is used.

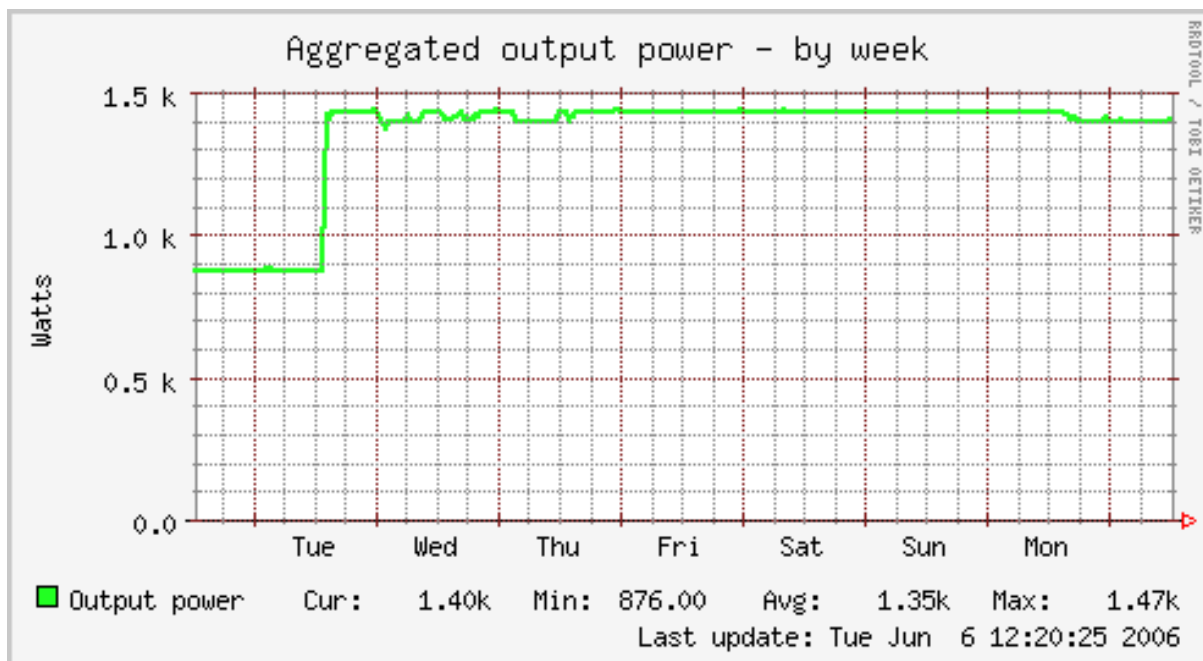
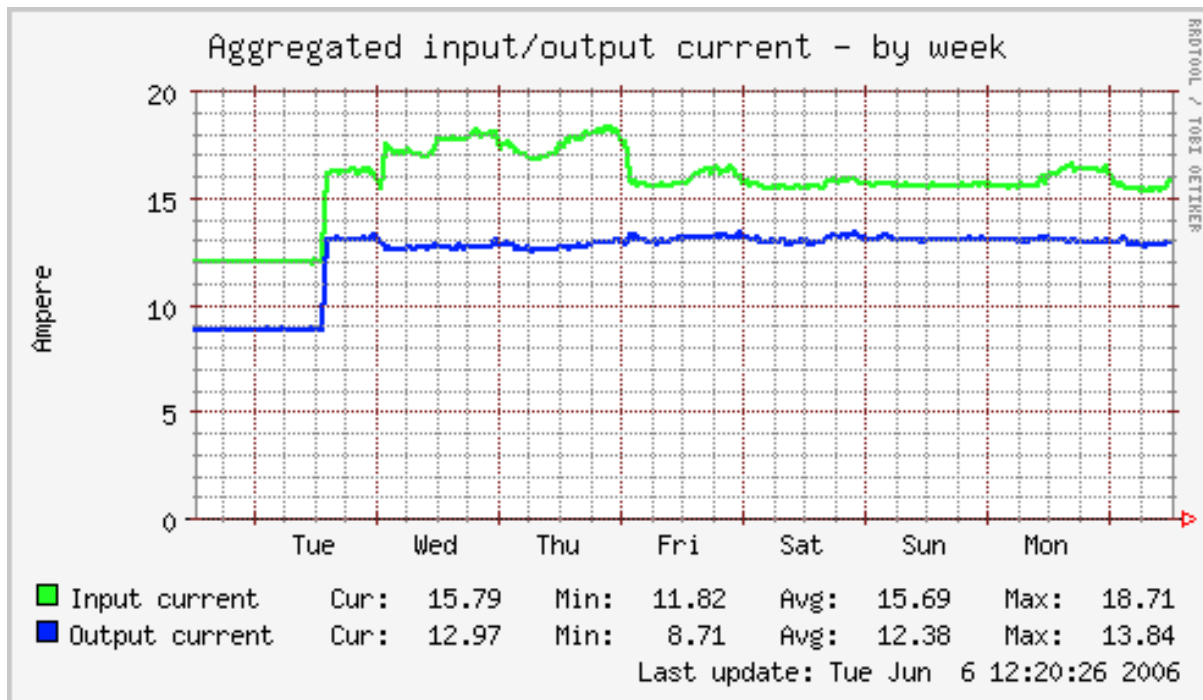
Result graphs

The graphs below show one of the UPSes, and the aggregated values. The graphs used are by week, because they had a nice dip in the beginning of the graphing period :-)

Source graphs for one of the UPSes:



Aggregate graphs:



Summary

We have now, in addition to the two real UPS nodes “ups-5a” and “ups-5b” (lines 1 - 5), created one virtual host named “Aggregated” (line 7) with two virtual plugins: “snmp_ups_current” (lines 11 - 24) and “snmp_ups_power” (lines 26 - 35).

The “snmp_ups_current” virtual plugin outputs two field names: “inputtotal” (lines 16 and 19 - 21) and “outputtotal” (lines 17 and 22 - 24), while the “snmp_ups_power” virtual plugin outputs only one field name, namely “output” (lines 31 - 35).

Further reading

- *Graph aggregation stacking example*
- *Using SNMP plugins*
- *munin.conf* directives explained

Graph aggregation stacking example

Here we show how to create a stacked graph with data sources from multiple nodes.

Plugin involved

The example uses a plugin that monitors Postfix message throughput.

Let's first look at its config output:

```
# munin-run postfix_mailstats config
graph_title Postfix message throughput
graph_args --base 1000 -l 0
graph_vlabel mails / ${graph_period}
graph_scale no
graph_total Total
graph_category postfix
graph_period minute
delivered.label delivered
delivered.type DERIVE
delivered.draw AREA
delivered.min 0
r450.label reject 450
r450.type DERIVE
r450.draw STACK
r450.min 0
r454.label reject 454
r454.type DERIVE
r454.draw STACK
r454.min 0
r550.label reject 550
r550.type DERIVE
r550.draw STACK
r550.min 0
r554.label reject 554
r554.type DERIVE
r554.draw STACK
r554.min 0
```

Extract from munin.conf

The following extract from *munin.conf* is explained in detail, step by step, below the configuration.

```
1 [foo.example.com]
2     address ..
3
4 [bar.example.com]
5     address ..
6
7 [baz.example.com]
8     address ..
```

```

9
10 [aggregates.example.com]
11     update no
12     contacts no
13
14 # This graph stacks the number of postfix delivered mails / minute
15 # from the nodes foo.example.com, bar.example.com and baz.example.com
16
17     total_mailstats.update no
18     total_mailstats.graph_args --base 1000 -l 0
19     total_mailstats.graph_category postfix
20     total_mailstats.graph_period minute
21     total_mailstats.graph_title Postfix delivered messages
22     total_mailstats.graph_vlabel mails / ${graph_period}
23     total_mailstats.graph_scale no
24     total_mailstats.graph_total Total
25     total_mailstats.total_delivered.label not_used
26     total_mailstats.total_delivered.type DERIVE
27     total_mailstats.total_delivered.draw AREA
28     total_mailstats.total_delivered.min 0
29     total_mailstats.total_delivered.stack \
30         foo=Infrastruktur;foo.example.com:postfix_mailstats.delivered \
31         bar=Infrastruktur;bar.example.com:postfix_mailstats.delivered \
32         baz=Infrastruktur;baz.example.com:postfix_mailstats.delivered

```

Explanations, per line

- 1 - 2: Configuration of Node `foo.example.com`.
- 4 - 5: Configuration of Node `bar.example.com`.
- 7 - 8: Configuration of Node `bar.example.com`.
- 10: Define the name of the virtual node. The name of a node group can optionally be added here, e.g. “[Virtual; aggregates.example.com]”.
- 11: Make sure that `munin-update` does not try to actively gather information for this node.
- 12: Tell `munin-limits` not to send alerts if any limit is breached.

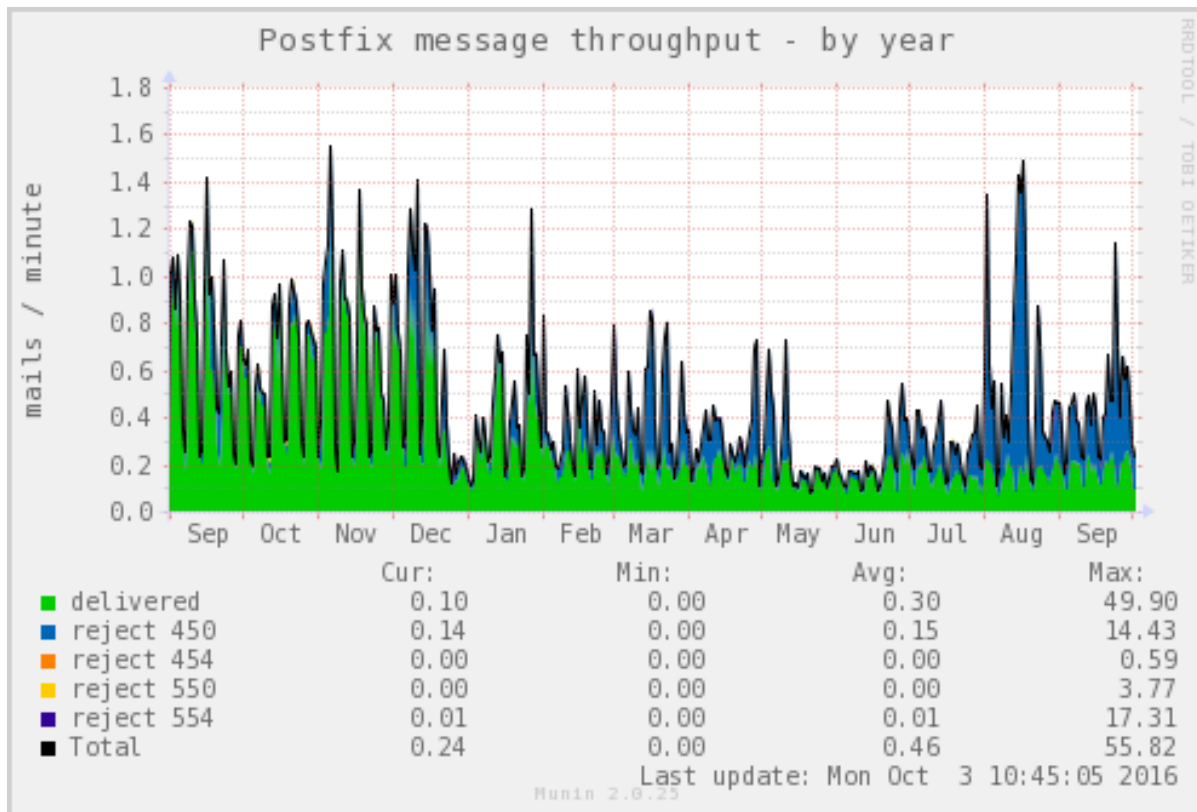
The above lines (1 - 12) have now established the fundament for four nodes in the Munin tree; three *real* nodes delivering data on connect by `munin-update` and one *virtual* node for the aggregate graphs.

- 17: No fetch from node should be done for this virtual plugin (which is named implicitly herewith to “total_mailstats”).
- 18 - 24: Define the virtual plugin’s config in the same way as set for its *ancestor* plugin `postfix_mailstats` (see output in section *Plugin involved* above). We set a different graph title though, as we graph only field `delivered` here.
- 25: The virtual plugin’s field itself will not show up in the graph but only the fields loaned from the real nodes. Therefore this hint that will be shown in the graphs legend.
- 26: Default type is `GAUGE` but we have type `DERIVE` here. So the field’s type has to be declared explicitly.
- 27: Values should be drawn as `AREA` and not as `LINE` for better recognition of small values.
- 28: Cut off negative values which can arise at arithmetic overflow for data type `DERIVE`. See [manual of rrdgraph](#)
- 29: This directive is the key that opens the door for loaning data sources from other nodes and plugins. As we choose option `stack` here, the values of the hereafter referenced data sources will show up stacked on each other in the graph.

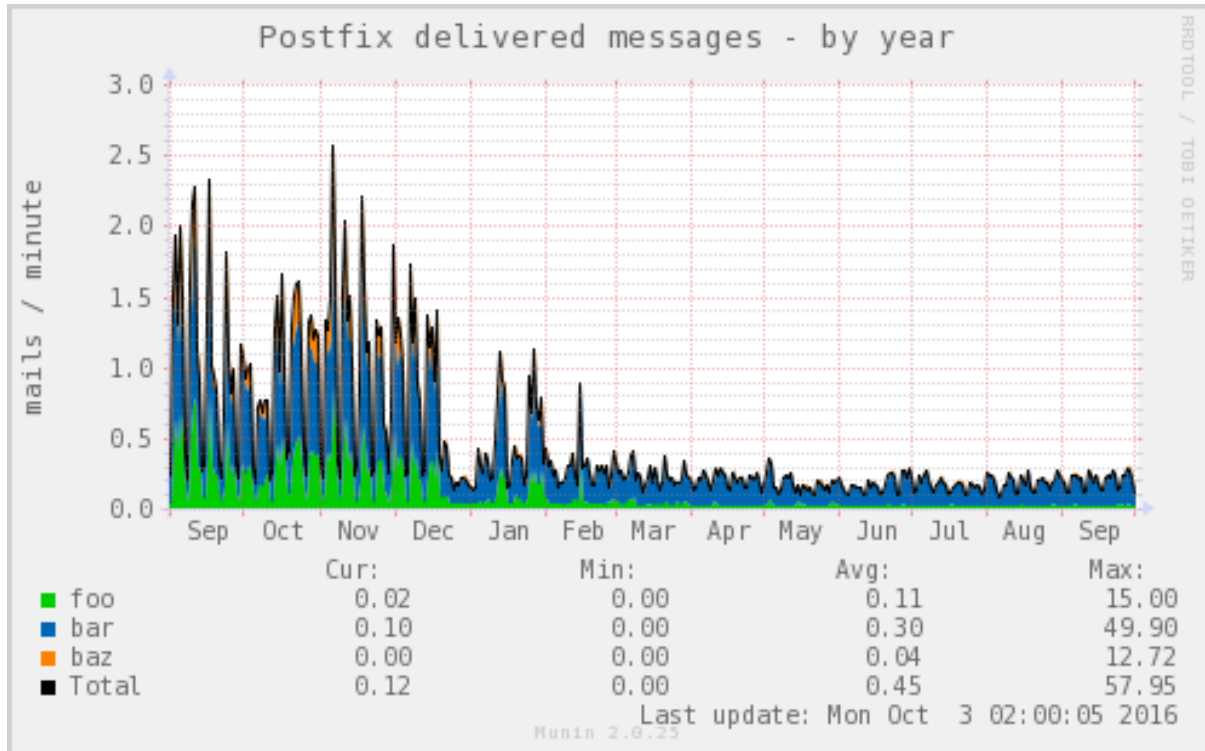
- 30 - 33: Declare the virtual data fields as reference to the original data source in node foo, bar and baz. The string on the left side of the expression will be the name of the data field shown in the graph of this virtual plugin. Hint: If you need to reference a node that is member of a node group, then enter its name with the leading group name like "Group;node.name".

Result graph

Source graph for bar.example.com:



Aggregate graph:



Further reading

- [Graph aggregation by example](#)
- [munin.conf directives explained](#)

Loaning data combined with sum and cdef

Different arithmetics may be combined to create the graphs you want. Insert appropriate lines in the Munin Master configuration file *munin.conf*.

The first example shows how to create a graph by loaning data from other sources, then adding an average value from five other data sources using *sum* and *cdef*. The data in the example uses the following types of temperature measurements

1. Disk temperatures from S.M.A.R.T. readouts from servers called **donald** and **ferdinand** (lines 06, 07)
2. Motherboard temperatures from `sensors` readouts from the same servers (lines 08, 09)
3. weather.com reading from Oslo airport, Gardermoen (line 10)
4. A regular temperature sensor (line 11)

Line 06 is only a placeholder, which will be populated later by using *sum* and *cdef*.

```
01 temperatures.update no
02 temperatures.graph_args --base 1000 -l 0
03 temperatures.graph_category Sensors
04 temperatures.graph_title Different temperatures
05 temperatures.graph_order \
06     donald_disk \
07     ferdinand_disk=ferdinand.example.com:hddtemp_smartctl.sda \
08     donald_mb=donald.example.com:sensors_temp.temp1 \
09     ferdinand_mb=ferdinand.example.com:sensors_temp.temp1 \
10     gardermoen=ferdinand.example.com:temperatures.ENG1 \
11     VG=donald.example.com:munintemp.vg
```

```

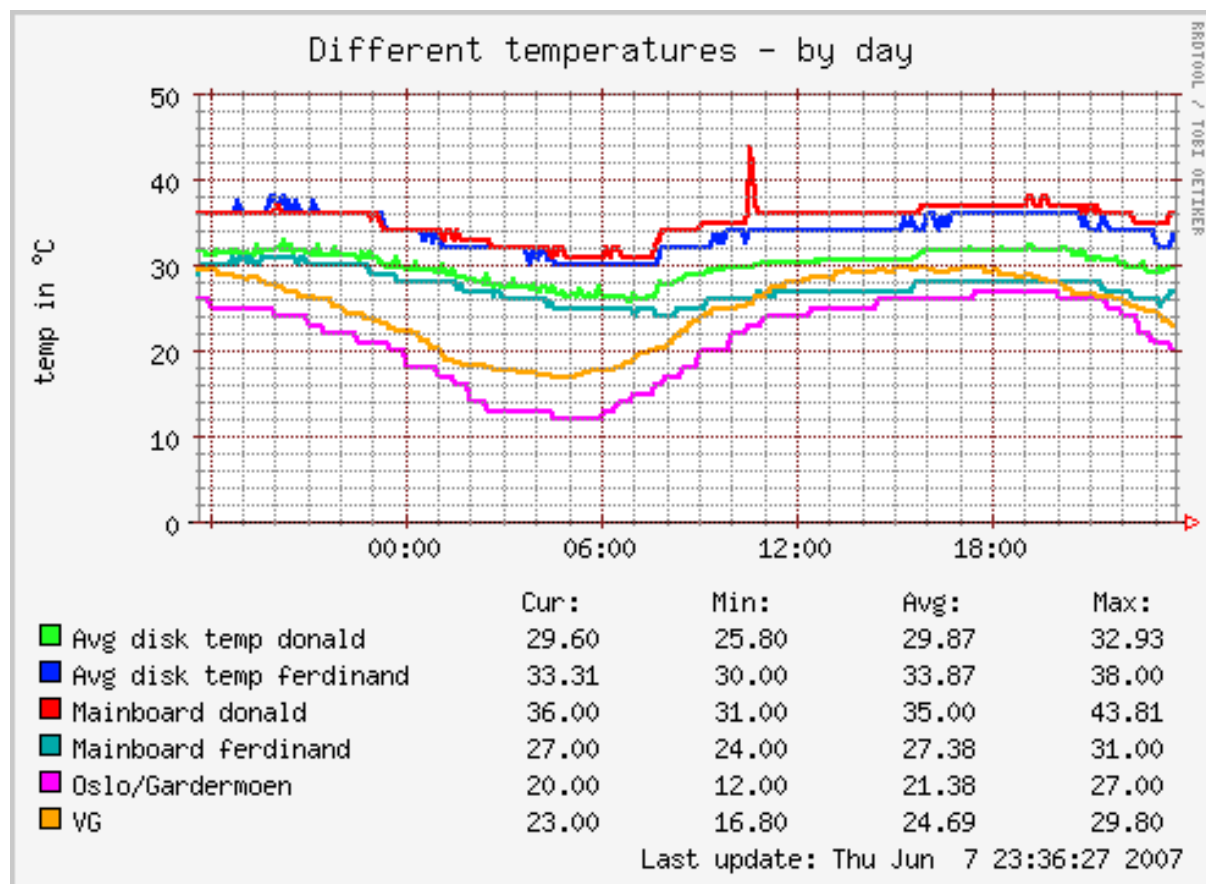
12  temperatures.donald_disk.sum \
13      donald.example.com:hddtemp_smartctl.sda \
14      donald.example.com:hddtemp_smartctl.sdb \
15      donald.example.com:hddtemp_smartctl.sdc \
16      donald.example.com:hddtemp_smartctl.sdd \
17      donald.example.com:hddtemp_smartctl.sde
18  temperatures.donald_disk.cdef donald_disk,5,/
19  temperatures.VG.label VG
20  temperatures.donald_mb.label Mainboard donald
21  temperatures.ferdinand_mb.label Mainboard ferdinand
22  temperatures.gardermoen.label Oslo/Gardermoen
23  temperatures.ferdinand_disk.label Avg disk temp ferdinand
24  temperatures.donald_disk.label Avg disk temp donald

```

Explanations, per line

- 01 - 04: The usual headers
- 05 - 11: Defines the graph order, where 5 out of 6 data sources are borrowed elsewhere Note: Line 11 defines a not yet “populated” data source.
- 12 - 17: Sums 5 other data sources (temperatures from 5 disks), into the “donald_disk” data source
- 18: Divides the “donald_disk” data source by the number of sources (5) to create an average
- 19 - 24: Labels to make it all look neat

This produces a pretty graph like this, to show the relation between outside temperatures and disk/mainboard temperatures:



Further reading

- *Graph aggregation by example*
- *Graph aggregation stacking example*
- *munin.conf* directives explained

multiple master data aggregation

This example describes a way to have multiple master collecting different information, and show all the data in a single presentation.

When you reach some size (probably several hundreds of nodes, several thousands plugins), 5 minutes is not enough for your single master to connect and gather data from all hosts, and you end up having holes in your graph.

Requirements

This example requires a shared nfs space for the munin data between the nodes.

Before going that road, you should make sure to check other options first, like changing the number of update threads, and having rrdcached.

An other option you might consider, is using munin-async. It requires modifications on all nodes, so it might not be an option, but I felt compelled to mention it. If you can't easily have shared nfs, or if you might have connectivity issues between master and some node, async would probably be a better approach.

Because there is some rrd path merge required, it is highly recommended to have **all** nodes in groups.

Overview

Munin-Master runs different scripts via the cron script (munin-cron).

munin-update is the only part actually connecting to the nodes. It gathers information and updates the rrd (you'll probably need rrdcached, especially via nfs).

munin-limits checks what was collected, compared to the limits and places warning and criticals.

The trick about having multiple master running to update is :

- run `munin-update` on different masters (called update-masters there after), having `dbdir` on nfs
- run `munin-limits` on either each of the update-masters, or the `html-master` (see next line)

Of course, all hosts must have access to the shared nfs directory.

Exemples will consider the shared folder `/nfs/munin`.

Running munin-update

Change the `munin-cron` to only run `munin-update` (and `munin-limits`, if you have alerts you want to be managed directly on those masters).

Change your `munin.conf` to use a `dbdir` within the shared nfs, (ie: `/nfs/munin/db/<hostname>`).

To make it easier to see the configuration, you can also update the configuration with an `includedir` on nfs, and declare all your nodes there (ie: `/nfs/munin/etc/<hostname>.d/`).

If you configured at least one node, you should have `/nfs/munin/db/<hostname>` that starts getting populated with subdirectories (groups), and a few files, including `datafile`, and `datafile.storable` (and `limits` if you also have `munin-limits` running here).

Merging data

All our update-masters generate update their dbdir including:

- `datafile` and `datafile.storable` which contain information about the collected plugins, and graphs to generate.
- directory tree with the rrd files

Merging files

`datafile` is just plain text with lines of `key value`, so concatenating all the files is enough.

`datafile.storable` is a binary representation of the data as loaded by munin. It requires some munin internal structures knowledge to merge them.

If you have `munin-limits` also running on update-masters, it generate a `limits` files, those are also plain text.

In order to make that part easier, a `munin-mergedb.pl` is provided in contrib.

Merging rrd tree

The main trick is about rrd. As we are using a shared nfs, we can use symlinks to get them to point to one an other, and not have to duplicate them. (Would be hell to keep in sync, that's why we really need shared nfs storage.)

As we deal with groups, we could just link top level groups to a common rrd tree.

Example, if you have two updaters (`update1` and `update2`), and 4 groups (`customer1`, `customer2`, `customer3`, `customer4`), you could make something like that:

```
/nfs/munin/db/shared-rrd/customer1/
/nfs/munin/db/shared-rrd/customer2/
/nfs/munin/db/shared-rrd/customer3/
/nfs/munin/db/shared-rrd/customer4/
/nfs/munin/db/update1/customer1 -> ../shared-rrd/customer1
/nfs/munin/db/update1/customer2 -> ../shared-rrd/customer2
/nfs/munin/db/update1/customer3 -> ../shared-rrd/customer3
/nfs/munin/db/update1/customer4 -> ../shared-rrd/customer4
/nfs/munin/db/update2/customer1 -> ../shared-rrd/customer1
/nfs/munin/db/update2/customer2 -> ../shared-rrd/customer2
/nfs/munin/db/update2/customer3 -> ../shared-rrd/customer3
/nfs/munin/db/update2/customer4 -> ../shared-rrd/customer4
/nfs/munin/db/html/customer1 -> ../shared-rrd/customer1
/nfs/munin/db/html/customer2 -> ../shared-rrd/customer2
/nfs/munin/db/html/customer3 -> ../shared-rrd/customer3
/nfs/munin/db/html/customer4 -> ../shared-rrd/customer4
```

At some point, an option to get the rrd tree separated from the dbdir, and should avoid the need of such links.

Examples for ssh transport

SSH options

Options for the `ssh://` transport can be added to `.ssh/config` in the home directory of the munin user.

The available options are available with `man ssh_config`. Here are some examples.

Compression

SSH has the option of compressing the data transport. To add compression to all SSH connections:

```
Host *
  Compression yes
```

If you have a lot of nodes, you will reduce data traffic by spending more CPU time. See also the `CompressionLevel` setting from the `ssh_config` man page.

Connecting through a Proxy

By using the `ProxyCommand` SSH option, you can connect with ssh via a jump host, and reach *munin-node* instances which are not available directly from the munin master:

```
Host *.customer.example.com !proxy.customer.example.com
ProxyCommand ssh -W %h:%p proxy.customer.example.com
```

This will make all connections to host ending with `.customer.example.com`, connect through `proxy.customer.example.com`, with an exemption for the proxy host itself.

Note: If you use *Compression*, try not to compress data twice. Disable compression for the proxied connections with *Compression no*.

Re-using SSH connections

If you connect to a host often, you can re-use the SSH connection instead. This is a good example to combine with the *Connecting through a proxy* and the *Compression* examples:

```
Host proxy.customer.example.com
ControlMaster      auto
ControlPath        /run/munin/ssh.%h_%p_%r
ControlPersist     360
TCPKeepAlive       yes
ServerAliveInterval 60
```

This will keep a long-lived SSH connection open to `proxy.customer.example.com`, it will be re-used for all connections. The SSH options `TCPKeepAlive` and `ServerAliveInterval` are added to detect and restart a dropped connection on demand.

service examples

Generic

- `munin crontab` entry

HP-UX

- `munin-node` init script `/etc/init.d/munin-node`
- `munin-node` rc.conf `/etc/rc.config.d/munin-node`

Linux

- `munin cron.d /etc/cron.d/munin`
- `munin-node cron.d /etc/cron.d/munin-node`
- `munin-node systemd service /etc/systemd/system/munin-node.service`
- `munin-node upstart job /etc/init/munin-node.conf`

Solaris

- `munin-node SMF manifest /var/svc/manifest/site/munin-node.xml`
- `munin-node SMF method /lib/svc/method/munin-node`

Examples from Munin Wiki

These should be checked against recent Munin versions and be transferred to the Munin-Guide.

- [Loaning data from other graphs](#)

Other reference material

Directories

dbdir

This directory is used to store the munin master database.

It contains one subdirectory with RRD files per group of hosts, as well as other variable state the munin master would need.

plugindir

This directory contains all the plugins the *munin node* should run.

pluginconfdir

This directory contains plugin configuration.

rundir

This directory contains files needed to track the munin run state. PID files, lock files, and possibly sockets.

logdir

Contains the log files for each munin program.

Plugin reference

When a plugin is invoked with “config” as (the only) argument it is expected to output configuration information for the graph it supports. This output consists of a number of attributes. They are divided into one set of global attributes and then one or more set(s) of datasource-specific attributes. (Things are more complex in the case of *Multigraph plugins* due to their nested hierarchy.)

Global attributes

Attribute graph**Value** yesno**Type** optional**Description** Decides whether to draw the graph.**See also****Default** yes

Attribute graph_args**Value** string**Type** optional**Description** Arguments for the rrd grapher. This is used to control how the generated graph looks, and how values are interpreted or presented.**See also** [rrdgraph](#)**Default**

Attribute graph_category**Value** string (Allowed characters: [a-z0-9-.])**Type** optional**Description**

Name of the category used to sort the graphs on the generated index web page.

Lower case string as we like a consistent view and want to avoid duplicates.

No whitespace as this makes the build of Munin Gallery a lot easier.

See also [Well known categories](#), [Plugin Gallery](#)**Default** ‘other’

Attribute graph_height**Value** integer (pixel)**Type** optional**Description** The height of the graph. Note that this is only the graph’s height and not the height of the whole PNG image.**See also****Default** 200

Attribute graph_info

Value html text

Type optional

Description Provides general information on what the graph shows.

See also

Default

Attribute graph_order

Value space separated list of data sources (fieldnames)

Type optional

Description

Ensures that the listed fields are displayed in specified order. Any additional fields are added in the order of appearance after fields appearing on this list. This attribute is useful when STACKing data sources with *fieldname.draw*.

It's also used for *loaning data* from other data sources (other plugins), which enables Munin to *create aggregate or other kinds of combined graphs*.

See also [Loaning Data](#), [Aggregate Graphs](#)

Default

Attribute graph_period

Value second|minute|hour

Type optional

Description

Controls the time unit munin (actually rrd) uses to calculate the average rates of change. Changing the default “second” to “minute” or “hour” is useful in cases of a low frequency of whatever the plugin is measuring.

Changing the graph_period makes sense only when the data type is COUNTER or DERIVE.

This does not change the sample interval - it remains per default at 5 minutes.

See also

Default second

Attribute graph_printf

Value Default format string for data source values.

Type optional

Description

Controls the format munin (actually rrd) uses to display data source values in the graph legend.

See also

Default “%7.2lf” if `-base` is 1024, otherwise “%6.2lf”

Attribute graph_scale

Value yesno

Type optional

Description Per default the unit written on the graph will be scaled. So instead of 1000 you will see 1k or 1M for 1000000. You may disable autoscale by setting this to 'no'.

See also

Default 'yes'

Attribute graph_title

Value string

Type required

Description Sets the title of the graph

See also

Default

Attribute graph_total

Value string

Type optional

Description

If set, summarizes all the data sources' values and reports the results in an extra row in the legend beneath the graph. The value you set here is used as label for that line.

Note that, since Munin version 2.1, using the special `undef` keyword disables it (to override in `munin.conf`).

See also

Default

Attribute graph_vlabel

Value string

Type optional

Description Label for the vertical axis of the graph. Don't forget to also mention the unit ;)

See also

Default

Attribute graph_width

Value integer (pixel)

Type optional

Description The width of the graph. Note that this is only the graph's width and not the width of the whole PNG image.

See also

Default 400

Attribute `host_name`

Value string

Type optional

Description Override the host name for which the plugin is run.

See also

Default Host name as declared in `munin.conf`.

Attribute `multigraph`

Value string

Type optional

Description

Herewith the plugin tells that it delivers a hierarchy of graphs. The attribute will show up multiple times in the config section, once for each graph that it contains. It announces the name of the graph for which the further configuration attributes then follow.

This feature is available since Munin version 1.4.0.

See also *Multigraph plugins*

Default

Attribute `update`

Value yes | no

Type optional

Description

Decides whether `munin-update` should fetch data for the graph.

Note that the graph will be shown even if updates are disabled and then be blank.

See also Set to `no` when dealing with *Graph aggregation* and/or *loaning data*.

Default 'yes'

Attribute `update_rate`

Value integer (seconds)

Type optional

Description

Sets the `update_rate` used by the Munin master when it creates the RRD file.

The update rate is the interval at which the RRD file expects to have data.

This attribute requires a Munin master version of at least 2.0.0

See also

Default

Data source attributes

Notes on field names

Each data source in a plugin must be identified by a field name.

The characters must be `[a-zA-Z0-9_]`, while the first character must be `[a-zA-Z_]`.

Reserved keyword(s): A field must not be named `root`. If it's done [Graph generation would be stopped](#).

In earlier versions of Munin the fieldname may not exceed 19 characters in length. Since munin 1.2 this limit has been circumvented.

Field name attributes

Attribute `{fieldname}.cdef`

Value CDEF statement

Type optional

Description

A CDEF statement is a Reverse Polish Notation statement. It can be used here to modify the value(s) before graphing.

This is commonly used to calculate percentages. See the [FAQ](#) for examples.

See also [cdeftutorial](#)

Default

Attribute `{fieldname}.colour`

Value Hexadecimal colour code

Type optional

Description Custom specification of colour for drawing curve. Available since 1.2.5 and 1.3.3.

See also

Default Selected by order sequence from Munin standard colour set

Attribute `{fieldname}.critical`

Value integer or decimal numbers (both may be signed)

Type optional

Description Can be a max value or a range separated by colon. E.g. "min:", ":max", "min:max", "max". Used by munin-limits to submit an error code indicating critical state if the value fetched is outside the given range.

See also [Let Munin croak alarm](#)

Default

Attribute `{fieldname}.draw`

Value AREA, LINE, LINE[n], STACK, AREASTACK, LINESTACK, LINE[n]STACK

Type optional

Description

Determines how the data points are displayed in the graph. The “LINE” takes an optional width suffix, commonly “LINE1”, “LINE2”, etc. . .

The *STACK values are specific to munin and makes the first a LINE, LINE[n] or AREA datasource, and the rest as STACK.

See also [rrdgraph](#)

Default ‘LINE1’ since Munin version 2.0.

Attribute {fieldname}.extinfo

Value html text

Type optional

Description Extended information that is included in alert messages (see *warning* and *critical*). Since 1.4.0 it is also included in the HTML pages.

See also

Default

Attribute {fieldname}.graph

Value yes/no

Type optional

Description Determines if the data source should be visible in the generated graph.

See also

Default yes

Attribute {fieldname}.info

Value html text

Type optional

Description Explanation on the data source in this field. The Info is displayed in the field description table on the detail web page of the graph.

See also

Default

Attribute {fieldname}.label

Value anything except # and \

Type required

Description The label used in the legend for the graph on the HTML page.

See also

Default

Attribute {fieldname}.line

Value value [:color[:label]]

Type optional

Description Adds a horizontal line with the fieldname's colour (HRULE) at the value defined. Will not show if outside the graph's scale.

See also [rrdgraph](#)

Default

Note: Didn't work here (munin-2.0.25-2.el6.noarch, rrdtool-1.3.8-7.el6.x86_64). Please investigate on your platforms and report the versions of Munin and rrdtool to Munin mailinglist if it worked for you.

Attribute {fieldname}.max

Value numerical of same data type as the field it belongs to.

Type optional

Description Sets a maximum value. If the fetched value is above "max", it will be discarded.

See also

Default

Attribute {fieldname}.min

Value numerical of same data type as the field it belongs to.

Type optional

Description Sets a minimum value. If the fetched value is below "min", it will be discarded.

See also

Default

Attribute {fieldname}.negative

Value {fieldname} of related field.

Type optional

Description You need this for a "mirrored" graph. Values of the named field will be drawn below the X-axis then (e.g. plugin `if_` that shows traffic going in and out as mirrored graph).

See also See the [Best Current Practices for good plugin graphs](#) for examples

Default

Attribute {fieldname}.stack

Value List of field declarations referencing the data sources from other plugins by their virtual path. (FIXME: Explanation on topic "virtual path" should be added elsewhere to set a link to it here)

Type optional

Description Function for creating stacked graphs.

See also [How do I use filename.stack?](#) and [Stacking example](#)

Default

Attribute {fieldname}.sum

Value List of fields to summarize. If the fields are loaned from other plugins they have to be referenced by their virtual path. (FIXME: Explanation on topic “virtual path” should be added elsewhere to set a link to it here)

Type optional

Description Function for creating summary graphs.

See also [How do I use fieldname.sum?](#) and [Graph aggregation by example](#)

Default

Attribute {fieldname}.type

Value GAUGE|COUNTER|DERIVE|ABSOLUTE

Type optional

Description Sets the RRD Data Source Type for this field. The values **must** be written in capitals. The type used may introduce restrictions for {fieldname.value}.

See also [Datatypes](#), [rrdcreate](#)

Default GAUGE

Note: COUNTER is now considered **harmful** because you can't specify the wraparound value. The same effect can be achieved with a DERIVE type, coupled with a `min 0`.

Attribute {fieldname}.warning

Value integer or decimal numbers (both may be signed)

Type optional

Description Can be a max value or a range separated by colon. E.g. “min:”, “:max”, “min:max”, “max”. Used by munin-limits to submit an error code indicating warning state if the value fetched is outside the given range.

See also [Let Munin croak alarm](#)

Default

On a data fetch run, the plugin is called with no arguments. the following fields are used.

Attribute {fieldname}.value

Value integer, decimal numbers, or “U” (may be signed). For DERIVE and COUNTER values this must be an integer. See [rrdcreate](#) for restrictions.

Type required

Description The value to be graphed.

See also

Default No default

Example

This is an example of the plugin fields used with the “df” plugin. The “munin-run” command is used to run the plugin from the command line.

Configuration run

```
# munin-run df config
graph_title Filesystem usage (in %)
graph_args --upper-limit 100 -l 0
graph_vlabel %
graph_category disk
graph_info This graph shows disk usage on the machine.
_dev_hda1.label /
_dev_hda1.info / (ext3) -> /dev/hda1
_dev_hda1.warning 92
_dev_hda1.critical 98
```

Data fetch run

```
# munin-run df
_dev_hda1.value 83
```

Plugin graph categories

The graph categories should create a general grouping of plugins.

A plugin that outputs a “graph_category” attribute will get the graph grouped with other plugin graphs using the same category, across all nodes on the same Munin master.

If a plugin doesn’t declare a graph_category in its config output, the graph is moved to default plugin category *other*.

A graph may only belong to one category.

Note: A *multigraph plugin* may create multiple graphs, and may place those in different categories.

To get a clear and concise overview in the Munin web interface **the list of categories should be small and meaningful**.

Therefore we compiled a list of *well-known categories* (see below).

Customizing category names

If you have lots of different types of databases in use, it makes sense to be more specific, and add a graph_category for each e.g. “oracle”, “mysql”.

Also graphs in several categories could be moved to a “security” category, but that may not make sense for everyone.

Or in our example below we add a new category *mem* to collect graphs that show memory aspects of the machine.

In short: Categories should reflect **your** monitoring perspective and you can move graphs to other categories or create new category names by overwriting the graph_category directives in the concerning host tree section of the Munin Master configuration in *munin.conf*.

Example configuration

```
[munin.example.com]
address localhost

# Node specific changes of plugin directives
memory.graph_category mem
buddyinfo.graph_category mem
swap.graph_category mem
```

Well known categories

Below we name our well-known graph categories (as already implemented in the contrib repository) and describe which data sources are suitable for the different categories.

The list is meant as a proposal to discuss and comment. You can do so on our munin-users mailing list or by creating a bug report (issue) on github.

Info for plugin contributors

You should refer to the “well known categories” when uploading your plugins to the repository.

The graph categories set for plugins in the repositories are also used to browse the [Munin Plugin Gallery](#). They are shown on each index page on the left side with a link to the concerning category page which lists all plugins with graphs in this category.

Therefore it makes sense to **use generic terms only for the categories**. This way we make sure that users get significant search results when looking for a special software product using a search engine. Specific **product names should be used to name the directories in the repository**, where you place the plugin. Their names are shown in the Plugin Gallery as title of the section where the plugins are listed. This way the search for product names brings only those Gallery pages as hits, where significant plugins are listed.

Please do not contribute plugins with product specific category terms as the search will then bring **all index pages** as hits, which is not helpful for the users of the Gallery. It should operate in an effective way as *Plugin Shop*, so significant retrieval is an important and critical demand here.

Note: Important! Please write the config line for plugins category in a concrete string (e.g. `graph_category memory`). The gallery build script scans for such a line in the plugins source code and needs it. Otherwise (e.g. use of variables) your plugin will only be shown under category “other”.

graph_category 1sec

Description

Examples

graph_category antivirus

Description Anti virus tools

Examples

graph_category appserver

Description Application servers

Examples

graph_category auth

Description Authentication servers and services

Examples

graph_category backup

Description All measurements around backup creation

Examples

graph_category chat

Description Messaging servers

Examples

graph_category cloud

Description Cloud providers and cloud components

Examples

graph_category cms

Description Content Management Systems

Examples

graph_category cpu

Description CPU measurements

Examples

graph_category db

Description Database servers

Examples MySQL, PostgreSQL, MongoDB, Oracle

graph_category devel

Description (Software) Development Tools

Examples

graph_category disk

Description Disk and other storage measurements

Examples : used space, free inodes, activity, latency, throughput

graph_category dns

Description Domain Name Server

Examples

graph_category filetransfer

Description Filetransfer tools and servers

Examples

graph_category forum

Description Forum applications

Examples

graph_category fs

Description (Network) Filesystem activities, includes also monitoring of distributed storage appliances

Examples

graph_category fw

Description All measurements around network filtering

Examples

graph_category games

Description Game-Server

Examples

graph_category htc

Description High-throughput computing

Examples

graph_category loadbalancer

Description Load balancing and proxy servers..

Examples

graph_category mail

Description Mail throughput, mail queues, etc.

Examples Postfix, Exim, Sendmail

Comment For monitoring a large mail system, it makes sense to override this with configuration on the Munin master, and make graph categories for the mail roles you provide. Mail Transfer Agent (postfix and exim), Mail Delivery Agent (filtering, sorting and storage), Mail Retrieval Agent (imap server).

graph_category mailinglist

Description Listserver

Examples

graph_category memory

Description All kind of memory measurements. Note that info about memory caching servers is also placed here

Examples

graph_category munin

Description Monitoring the monitoring.. (includes other monitoring servers also)

Examples

graph_category network

Description General networking metrics.

Examples interface activity, latency, number of open network connections

graph_category other

Description Plugins that address seldom used products. Category /other/ is the default, so if the plugin doesn't declare a category, it is also shown here.

Examples

graph_category printing

Description Monitor printers and print jobs

Examples

graph_category processes

Description Process and kernel related measurements

Examples

graph_category radio

Description Receivers, signal quality, recording, ..

Examples

graph_category san

Description Storage Area Network

Examples

graph_category search

Description All kinds of measurement around search engines

Examples

graph_category security

Description Security information

Examples login failures, number of pending update packages for OS, number of CVEs in the running kernel fixed by the latest installed kernel, firewall counters.

graph_category sensors

Description Sensor measurements of device and environment

Examples temperature, power, devices health state, humidity, noise, vibration

graph_category spamfilter

Description Spam fighters at work

Examples

graph_category streaming

Description

Examples

graph_category system

Description General operating system metrics.

Examples CPU speed and load, interrupts, uptime, logged in users

graph_category time

Description Time synchronization

Examples

graph_category tv

Description Video devices and servers

Examples

graph_category virtualization

Description All kind of measurements about server virtualization. Includes also Operating-system-level virtualization

Examples

graph_category voip**Description** Voice over IP servers**Examples**

graph_category webserver**Description** All kinds of webserver measurements and also for related components**Examples** requests, bytes, errors, cache hit rate for Apache httpd, nginx, lighttpd, varnish, hitch, and other web servers, caches or TLS wrappers.

graph_category wiki**Description** wiki applications**Examples**

graph_category wireless**Description****Examples**

A Brief History of Munin

2002 Born as LRRD**2004** Renamed as Munin**2007** Hacked zooming for 1.2**2009** 1.4 came out**2011** EOL of *Munin Exchange website*, content moved to GitHub branch **contrib****2012** Released 2.0, for its 10 years !**2013** Released 2.1**July 2014** target for 2.2

Glossary

Munin Exchange

Was a web platform in the beginning setup and hosted by Linpro (Bjorn Ruberg?). Later (when?) a Munin supporter re-invented the *Munin Exchange* website to improve its usability. When he left the project (when?) it was not possible to maintain the website any longer, because it was coded in Python with Django and Steve Schnepf said “we clearly lack skills on that”. So it was decided to move all the plugins over to github branch “contrib”.

GitHub is now the official way of contributing 3rd-party plugins.

These are tagged with **family contrib** (see: `--families` in *munin-node-configure*).

Only if they meet the requirements for **vetted plugins** they can be included in the core plugins collection (distributed as *official* Munin release by the Munin developer team). They get tagged with **family auto** then as all core collection plugins should have the command `autoconf` implemented.

See also: *Munin Gallery*

Contributing

We need help with completing the [Munin Gallery](#). A lot of plugins don't have their documentation in POD style format. And the Gallery needs more pics!

See our wiki page with [instructions for gallery contributors](#).

Developer Talk

- [Proposed extensions](#)

Specifications

- [Munin Relay](#) : a multiplexing relay (not yet implemented)
- [REST API \(RFC\)](#)
- [Stream plugin \(RFC\)](#)

Ideas

- [Some thoughts about Tags](#)
- [Some thoughts about basic module API for plugins written in perl, python and ruby](#)

Snippets

Dirty config

A *dirty* fetch is not desirable because some plugins (rightly) assume that `config` is done first and then `fetch` and updates the state file accordingly. This should be accommodated since we can. Therefore the feature will be called *dirty config* instead.

Today, for each plugin, on every run, `munin-update` first does `config $plugin` and then a `fetch $plugin`.

Some plugins do a lot of work to gather their numbers. Quite a few of these need to do the same amount of work for the value fetching and printing as for the `config` output.

We could halve the execution time if `munin-update` detects that `config $plugin` produces `.value` and from that deduces that the `fetch $plugin` is superfluous. If so the `fetch` will not be executed thus halving the total execution time. A `config` with `fetch-time` output in would be called a `dirty config` since it not only contains `config-time` output. If the `config` was not dirty a old fashioned `fetch` must be executed.

So: Versions of `munin-update` that understands a dirty config emits `cap dirtyconfig` to the node, if the node understands the capability it will set the environment variable `MUNIN_CAP_DIRTYCONFIG` before executing plugins.

A plugin that understands `MUNIN_CAP_DIRTYCONFIG` can simply do something like (very pseudo code) `if (is_dirtyconfig()) then do_fetch(); endif; at the end of the do_config() procedure. Plugin.sh and Plugin.pm needs to implement a is_dirtyconfig - or similarly named - procedure to let plugins test this easily.`

Plugins that do dirty config to a master that does not understand it should still work OK: old `munin-updates` would do a good deal of error logging during the `config`, but since they do not understand the point of all this `config` in the `fetch` they would also do `config` and everything would work as before.

Plugins that want to be polite could check the masters capabilities before executing a dirty `config` as explained above.

After 2.0 has been published with this change in plugins would be simpler and combined with some other extensions it would greatly reduce the need for many plugin state files and such complexity. Should make plugin author lives much easier all together.

CHAPTER 10

Indices and tables

- `genindex`

Symbols

- `-always-send <severity list>`
 - `munin-limits` command line option, 91
- `-base <value>`
 - command line option, 109
- `-cleanup`
 - `munin-async` command line option, 88
- `-cleanupandexit`
 - `munin-async` command line option, 88
- `-config <configfile>`
 - `munin-node` command line option, 95
 - `munin-run` command line option, 97
- `-config <file>`
 - command line option, 93
 - `munin-cron` command line option, 90
 - `munin-limits` command line option, 91
- `-config_file <file>`
 - `munin-update` command line option, 98
- `-contact <contact>`
 - `munin-limits` command line option, 91
- `-debug`
 - command line option, 93
 - `munin-limits` command line option, 92
 - `munin-node` command line option, 95
 - `munin-run` command line option, 97
 - `munin-update` command line option, 98
- `-exitnoterror`
 - command line option, 93
- `-families <family,...>`
 - command line option, 93
- `-fix-permissions | -f`
 - `munin-check` command line option, 90
- `-force`
 - `munin-limits` command line option, 92
- `-force-run-as-root`
 - `munin-limits` command line option, 92
- `-fork`
 - `munin-asyncd` command line option, 89
 - `munin-update` command line option, 98
- `-help`
 - command line option, 93
 - `munin-limits` command line option, 92
 - `munin-node` command line option, 95
 - `munin-run` command line option, 97
 - `munin-update` command line option, 98
- `-help | -h`
 - `munin-async` command line option, 88
 - `munin-asyncd` command line option, 89
 - `munin-check` command line option, 90
- `-host <host>`
 - `munin-cron` command line option, 90
 - `munin-limits` command line option, 91
 - `munin-update` command line option, 98
- `-host <hostname:port>`
 - `munin-asyncd` command line option, 89
- `-hostname <hostname>`
 - `munin-async` command line option, 88
- `-interval <seconds>`
 - `munin-asyncd` command line option, 89
- `-libdir <dir>`
 - command line option, 93
- `-logarithmic`
 - command line option, 109
- `-lower-limit <value>`
 - command line option, 109
- `-newer <version>`
 - command line option, 93
- `-nocleanup`
 - `munin-asyncd` command line option, 89
- `-paranoia`
 - `munin-node` command line option, 95
 - `munin-run` command line option, 97
- `-pidebug`
 - command line option, 93
 - `munin-node` command line option, 95
 - `munin-run` command line option, 97
- `-remove-also`
 - command line option, 93
- `-retain <count>`
 - `munin-asyncd` command line option, 89
- `-rigid`
 - command line option, 109
- `-sconfdir <dir>`
 - command line option, 93
 - `munin-run` command line option, 97
- `-sconfdir <file>`
 - `munin-run` command line option, 97

- screen
 - munin-limits command line option, 92
 - munin-update command line option, 98
- service <service>
 - munin-cron command line option, 90
 - munin-limits command line option, 91
 - munin-update command line option, 98
- servicedir <dir>
 - command line option, 93
 - munin-run command line option, 97
- shell
 - command line option, 93
- snmp <hostcidr,...>
 - command line option, 94
- snmpauthpass <password>
 - command line option, 94
- snmpauthproto <protocol>
 - command line option, 94
- snmpcommunity <string>
 - command line option, 94
- snmpdomain <domain>
 - command line option, 94
- snmpport <port>
 - command line option, 94
- snmpprivpass <password>
 - command line option, 94
- snmpprivproto <protocol>
 - command line option, 95
- snmpusername <name>
 - command line option, 94
- snmpversion <ver>
 - command line option, 94
- spool | -s <spooldir>
 - munin-asyncd command line option, 89
- spooldir | -s <spooldir>
 - munin-async command line option, 88
- spoolfetch
 - munin-async command line option, 88
- suggest
 - command line option, 93
- timeout <seconds>
 - munin-update command line option, 98
- units-exponent <value>
 - command line option, 109
- upper-limit <value>
 - command line option, 109
- vectorfetch
 - munin-async command line option, 88
- verbose | -v
 - munin-async command line option, 88
 - munin-asyncd command line option, 89
- version
 - command line option, 93
 - munin-run command line option, 97
 - munin-update command line option, 98

A

address <value>

- munin.conf command line option, 100
- aggregate
 - plugin, 109, 114, 117
- Aggregating munin plugins, 109, 114, 117
- allow
 - command line option, 103
- apache httpd configuration
 - example, munin-httpd, 106
- auto
 - magic marker family, 58
- autoconf
 - magic marker capability, 59

B

background

- command line option, 103

C

cap

- protocol command, 34

capabilities

- magic marker, 58

capability

- autoconf, magic marker, 59
- dirtyconfig, 47
- multigraph, 45
- suggest, magic marker, 59

cdef

- field, 117

cidr_allow

- command line option, 104

cidr_deny

- command line option, 104

command

- cap, protocol, 34
- config, protocol, 34
- fetch, protocol, 34
- list, protocol, 34
- nodes, protocol, 34
- quit, protocol, 34
- version, protocol, 34

command line option

- base <value>, 109
- config <file>, 93
- debug, 93
- exitnoterror, 93
- families <family,...>, 93
- help, 93
- libdir <dir>, 93
- logarithmic, 109
- lower-limit <value>, 109
- newer <version>, 93
- pidebug, 93
- remove-also, 93
- rigid, 109
- sconfdir <dir>, 93
- servicedir <dir>, 93
- shell, 93

- snmp <host|cidr,...>, 94
- snmpauthpass <password>, 94
- snmpauthproto <protocol>, 94
- snmpcommunity <string>, 94
- snmpdomain <domain>, 94
- snmpport <port>, 94
- snmpprivpass <password>, 94
- snmpprivproto <protocol>, 95
- snmpusername <name>, 94
- snmpversion <ver>, 94
- suggest, 93
- units-exponent <value>, 109
- upper-limit <value>, 109
- version, 93
- allow, 103
- background, 103
- cidr_allow, 104
- cidr_deny, 104
- global_timeout, 103
- group, 103
- host, 104
- host_name, 103
- ignore_file, 103
- log_file, 103
- log_level, 103
- paranoia, 103
- pid_file, 103
- port, 104
- setsid, 103
- timeout, 103
- user, 103
- config
 - protocol command, 34
- configuration
 - example plugin, 50
 - plugin, 49
- contact.<contact name>.command <command>
 - munin.conf command line option, 99
- contact.<contact name>.max_messages <number>
 - munin.conf command line option, 99
- contact.<contact name>.text <text>
 - munin.conf command line option, 99
- contacts <nolcontact ...>
 - munin.conf command line option, 100
- contrib
 - magic marker family, 58
 - munin, 71
- contributing
 - munin documentation, 7
 - munin; documentation, 73
- critical <value>
 - munin.conf command line option, 101
- custom_palette rrggbb rrggbb ...
 - munin.conf command line option, 99
- D**
- data source
 - fqn, 87
 - loan, 109, 114, 117
- dbdir <path>
 - munin.conf command line option, 98
- development
 - munin, 67
 - plugin, 71
- dirtyconfig
 - capability, 47
 - protocol extension, 47
- documentation
 - contributing munin, 7
- ds
 - fqn, 87
- E**
- environment
 - plugin, 41
- environment variable
 - MUNIN_DEBUG, 95, 97
- example
 - graph_args, 109
 - lighttpd configuration munin-httpd, 106
 - magic marker family, 58
 - munin-httpd apache httpd configuration, 106
 - munin-httpd nginx configuration, 107
 - munin-node.conf, 95
 - munin.conf, 99
 - plugin configuration, 50
 - ssh transport munin.conf, 120
- executing
 - plugin, 130
- extension
 - dirtyconfig, protocol, 47
 - multigraph, protocol, 45
- F**
- family
 - auto, magic marker, 58
 - contrib, magic marker, 58
 - example, magic marker, 58
 - magic marker, 58
 - manual, magic marker, 58
 - snmpauto, magic marker, 58
 - test, magic marker, 58
- fetch
 - protocol command, 34
- field
 - cdef, 117
 - sum, 117
- fields
 - plugin, 123
- fork <yes|no>
 - munin.conf command line option, 99
- fqn, 85
 - data source, 87
 - ds, 87
 - group, 86
 - node, 86

- plugin, 86
- service, 86

fully qualified name, 85

G

global_timeout
 command line option, 103

graph
 stacked, 114

graph_args, 109
 example, 109

graph_category, 131

graph_data_size <normallhuge>
 munin.conf command line option, 99

graph_height <value>
 munin.conf command line option, 101

graph_width <value>
 munin.conf command line option, 101

group
 command line option, 103

 fqn, 86

H

host
 command line option, 104

host_name
 command line option, 103

I

ignore_file
 command line option, 103

ignore_unknown <yes/no>
 munin.conf command line option, 100

installing
 plugin, 49

L

lighttpd configuration
 munin-httpd, example, 106

list
 protocol command, 34

loan
 data source, 109, 114, 117

local_address <address>
 munin.conf command line option, 100

log_file
 command line option, 103

log_level
 command line option, 103

logdir <path>
 munin.conf command line option, 98

M

magic marker
 capabilities, 58

- capability autoconf, 59
- capability suggest, 59

- family, 58
- family auto, 58
- family contrib, 58
- family example, 58
- family manual, 58
- family snmpauto, 58
- family test, 58
- plugin, 58

man pages, 87

manual
 magic marker family, 58

multigraph
 capability, 45

- protocol extension, 45

munin
 contrib, 71

- development, 67
- documentation, contributing, 7
- project, 67

Munin Exchange, 137

munin-async command line option
 -cleanup, 88

- cleanupandexit, 88
- help | -h, 88
- hostname <hostname>, 88
- spooldir | -s <spooldir>, 88
- spoolfetch, 88
- vectorfetch, 88
- verbose | -v, 88

munin-asynd command line option
 -fork, 89

- help | -h, 89
- host <hostname:port>, 89
- interval <seconds>, 89
- nocleanup, 89
- retain <count>, 89
- spool | -s <spooldir>, 89
- verbose | -v, 89

munin-check command line option
 -fix-permissions | -f, 90

- help | -h, 90

munin-cron command line option
 -config <file>, 90

- host <host>, 90
- service <service>, 90

munin-httpd
 apache httpd configuration example, 106

- example lighttpd configuration, 106
- nginx configuration, example, 107

munin-limits command line option
 -always-send <severity list>, 91

- config <file>, 91
- contact <contact>, 91
- debug, 92
- force, 92
- force-run-as-root, 92
- help, 92
- host <host>, 91

- screen, 92
 - service <service>, 91
 - munin-node command line option
 - config <configfile>, 95
 - debug, 95
 - help, 95
 - paranoia, 95
 - pidebug, 95
 - munin-node.conf
 - example, 95
 - munin-run command line option
 - config <configfile>, 97
 - debug, 97
 - help, 97
 - paranoia, 97
 - pidebug, 97
 - sconfdir <dir>, 97
 - sconfdir <file>, 97
 - servicedir <dir>, 97
 - version, 97
 - munin-update command line option
 - config_file <file>, 98
 - debug, 98
 - fork, 98
 - help, 98
 - host <host>, 98
 - screen, 98
 - service <service>, 98
 - timeout <seconds>, 98
 - version, 98
 - munin.conf
 - example, 99
 - example, ssh transport, 120
 - munin.conf command line option
 - address <value>, 100
 - contact.<contact name>.command <command>, 99
 - contact.<contact name>.max_messages <number>, 99
 - contact.<contact name>.text <text>, 99
 - contacts <nolcontact ...>, 100
 - critical <value>, 101
 - custom_palette rrggbb rrggbb ..., 99
 - dbdir <path>, 98
 - fork <yes/no>, 99
 - graph_data_size <normallhuge>, 99
 - graph_height <value>, 101
 - graph_width <value>, 101
 - ignore_unknown <yes/no>, 100
 - local_address <address>, 100
 - logdir <path>, 98
 - notify_alias <node name>, 100
 - palette <default/old>, 99
 - port <port number>, 100
 - rundir <path>, 98
 - ssh_command <command>, 99
 - ssh_options <options>, 99
 - timeout <seconds>, 99
 - tmpldir <path>, 98
 - update <yes/no>, 100
 - use_node_name <yes/no>, 100
 - warning <value>, 101
 - munin; documentation
 - contributing, 73
 - MUNIN_DEBUG, 95, 97
- ## N
- nginx configuration
 - example munin-httpd, 107
 - node
 - fqn, 86
 - virtual, 98, 109, 114, 117
 - nodes
 - protocol command, 34
 - notify_alias <node name>
 - munin.conf command line option, 100
- ## P
- palette <default/old>
 - munin.conf command line option, 99
 - paranoia
 - command line option, 103
 - pid_file
 - command line option, 103
 - plugin
 - aggregate, 109, 114, 117
 - configuration, 49
 - configuration, example, 50
 - development, 71
 - environment, 41
 - executing, 130
 - fields, 123
 - fqn, 86
 - installing, 49
 - magic marker, 58
 - testing, 50
 - virtual, 109, 114, 117
 - port
 - command line option, 104
 - port <port number>
 - munin.conf command line option, 100
 - project
 - munin, 67
 - protocol
 - command cap, 34
 - command config, 34
 - command fetch, 34
 - command list, 34
 - command nodes, 34
 - command quit, 34
 - command version, 34
 - extension dirtyconfig, 47
 - extension multigraph, 45
- ## Q
- quit

protocol command, 34

R

rundir <path>
munin.conf command line option, 98

S

service
fqn, 86
setsid
command line option, 103
snmpauto
magic marker family, 58
ssh transport
munin.conf example, 120
ssh_command <command>
munin.conf command line option, 99
ssh_options <options>
munin.conf command line option, 99
stacked
graph, 114
suggest
magic marker capability, 59
sum
field, 117

T

test
magic marker family, 58
testing
plugin, 50
timeout
command line option, 103
timeout <seconds>
munin.conf command line option, 99
tmpldir <path>
munin.conf command line option, 98
tuple: graph
category, 131
tuple: munin-node.conf
example, 104
tuple: plugin
graph_category, 131

U

update <yeslno>
munin.conf command line option, 100
use_node_name <yeslno>
munin.conf command line option, 100
user
command line option, 103

V

version
protocol command, 34
virtual
node, 98, 109, 114, 117

plugin, 109, 114, 117

W

warning <value>
munin.conf command line option, 101