Munin Documentation

Release 2.999.10-detached-2018-12-16-c13-g47debb5

The Munin project and its contributors

Dec 16, 2018

Contents

1 Preface **3**
- 1.1 What is Munin ? . . . 3
- 1.2 Conventions . . . 3
- 1.3 Further Information . . . 3
- 1.4 Bug Reporting Guidelines . . . 6
- 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 . . . 25
- 2.8 Monitoring the “unreachable” hosts . . . 25
- 2.9 Troubleshooting . . . 28

3 Munin’s Architecture **35**
- 3.1 Overview . . . 35
- 3.2 Components . . . 36
- 3.3 Protocols . . . 60
- 3.4 Syntax . . . 62
- 3.5 API . . . 64

4 Munin Installation **65**
- 4.1 Prerequisites . . . 65
- 4.2 Installing Munin . . . 66
- 4.3 Initial Configuration . . . 68
- 4.4 Webserver Configuration . . . 69
- 4.5 Upgrade Notes . . . 70

5 Munin Advanced **71**
- 5.1 TLS Setup . . . 71
- 5.2 Advanced Network . . . 74
- 5.3 Per plugin custom rrd sizing . . . 74

6 How-to Collection **77**
- 6.1 How to remove spikes . . . 77

7 Developing Munin **79**

7.1 Munin development environment 79
7.2 Munin development tests 81
7.3 Data Structures 82

8 Developing Plugins **87**
8.1 How to write Munin Plugins 87
8.2 How to write SNMP Plugins 94
8.3 Best Current Practices for good plugin graphs 96
8.4 The Concise guide to plugin authoring 99
8.5 Plugin Gallery 103
8.6 Advanced Topics for Plugin Development 105
8.7 Requirements for 'Vetted Plugins' 107

9 Reference **109**
9.1 Nomenclature 109
9.2 Man pages 113
9.3 Examples and Templates 139
9.4 Other reference material 160

10 Other docs **177**
10.1 Contributing 177
10.2 Developer Talk 177

11 Indices and Tables **179**

Contents:

CHAPTER 1

Preface

1.1 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.

1.2 Conventions

1.2.1 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.

1.3 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.

1.3.1 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://guide.munin-monitoring.org/ thanks to ReadTheDoc.

It is specially designed for easy contribution and distribution thanks to Sphinx. That aspect will be handled in *Contributing*.

1.3.2 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.

1.3.3 GitHub

The Munin GitHub has slowly become the center of all the community-driven development. It is a very solid platform, and despites 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 a 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.

1.3.4 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.

1.3.5 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.

1.3.6 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.

1.3.7 Planet Munin

In order to provide some central place to reference munin-related blogs out there, Planet Munin was created.

It aggregates many blogs via RSS, and presents them as just one feed.

To add your blog, just visit us on our *IRC Channel*, and ask there.

Note that providing a tagged or a category-filtered feed is the best way to remain on-topic.

1.4 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.

1.4.1 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.

1.4.2 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.

1.4.3 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.

1.5 Documenting Munin

1.5.1 Munin Guide

The pages you are just viewing ;)

The guide contains documentation about the Munin software.

It is written using reStructuredText[1].

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.

1.5.2 Plugin Documentation

Plugin documentation is included in each plugin, written using the POD[2] 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.

1.5.3 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.

1.5.4 Unix Manual Pages

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

1.5.5 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

[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.

[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 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

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 install[1], 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 license[2].

---

[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.

CHAPTER 2

Tutorial

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*.

2.1 Getting Started

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

2.1.1 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.

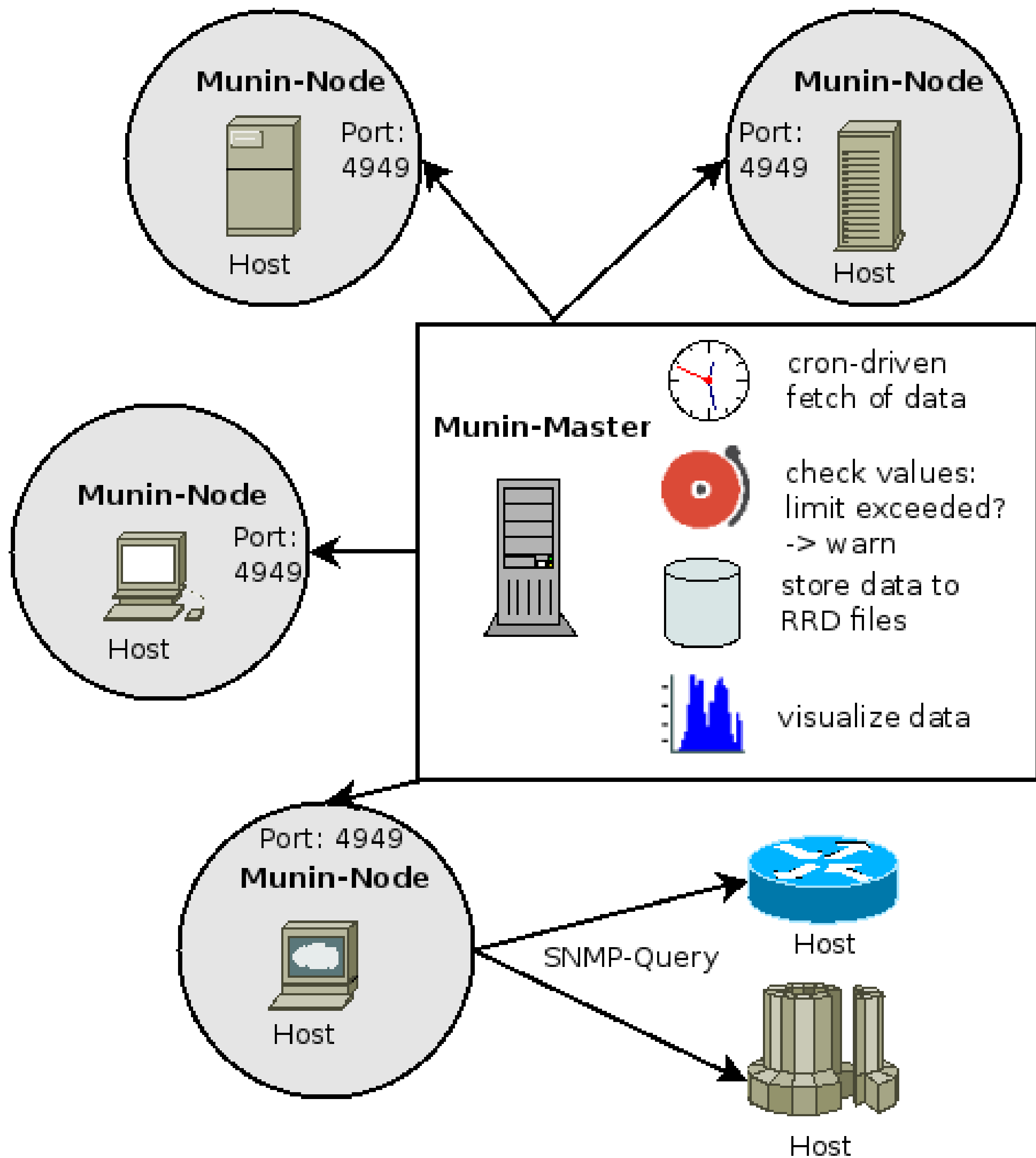
All the tutorial will assume a Debian installation, so all the commands are suited to the Debian package management system. As the one in Ubuntu is mostly the same, examples should work unchanged. For RPM-based systems, the equivalent yum command is left as an exercise to the reader, but should not be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

2.1.2 Architectural Fundamentals

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



2.1.3 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 :-)

2.2 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.

2.2.1 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 (_).

2.2.2 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.

2.2.3 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`).

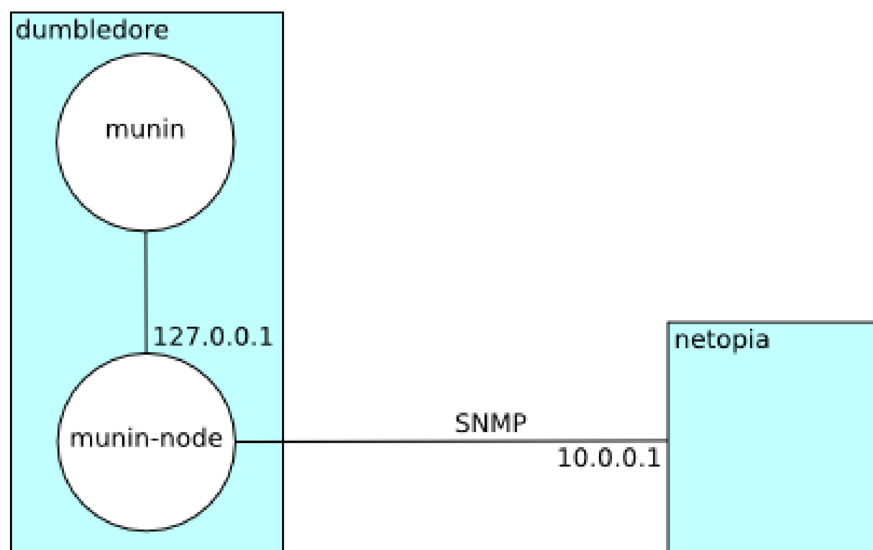
See *Using SNMP plugins* for information about configuring SNMP plugins.

2.3 Using SNMP plugins

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

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
```

(continues on next page)

(continued from previous page)

```
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 treat 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.

2.4 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.

2.4.1 Sending alerts through Nagios

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

2.4.2 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 '|' 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]).
```

2.4.3 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.

2.4.4 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.

2.4.5 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*.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.

2.5 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.

2.5.1 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_nsca` 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_nsca -H localhost -c /etc/
↪send_nsca.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!
```

2.5.2 Sending messages from Munin

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

Configuring munin.conf

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

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

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

2.5.3 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.
```

2.5.4 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.

2.6 Advanced Features

2.6.1 Introduction

2.6.2 Alias

2.6.3 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.

2.6.4 Supersampling

2.6.5 Conclusion

2.7 Extraordinary Usage

2.7.1 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 ;-)

2.7.2 Examples

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

2.8 Monitoring the "unreachable" hosts

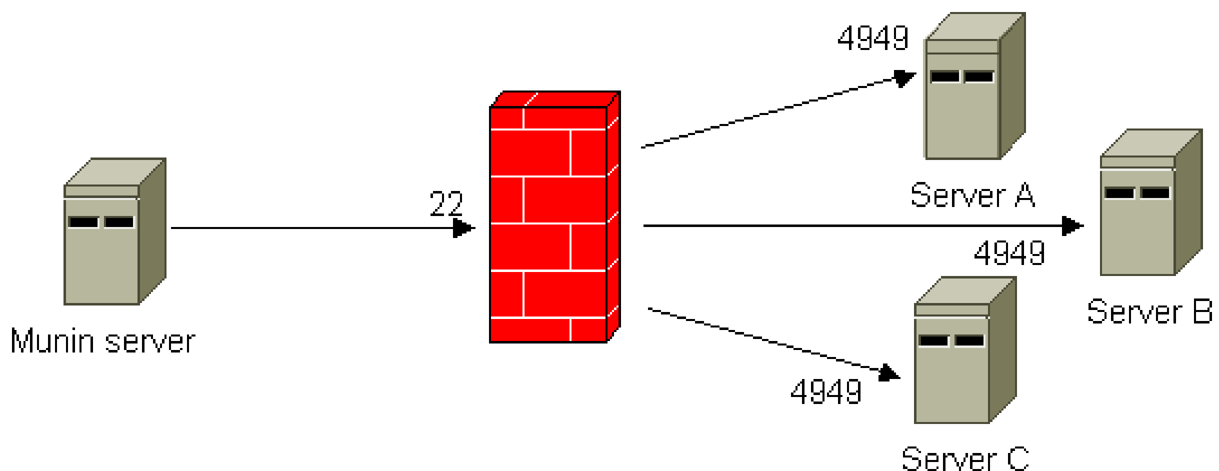
There are a number of situations where you'd like to run munin-node on hosts not directly available to the Munin server. This article describes a few scenarios and different alternatives to set up monitoring. Monitoring hosts behind a non-routing server.

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.

2.8.1 SSH tunneling

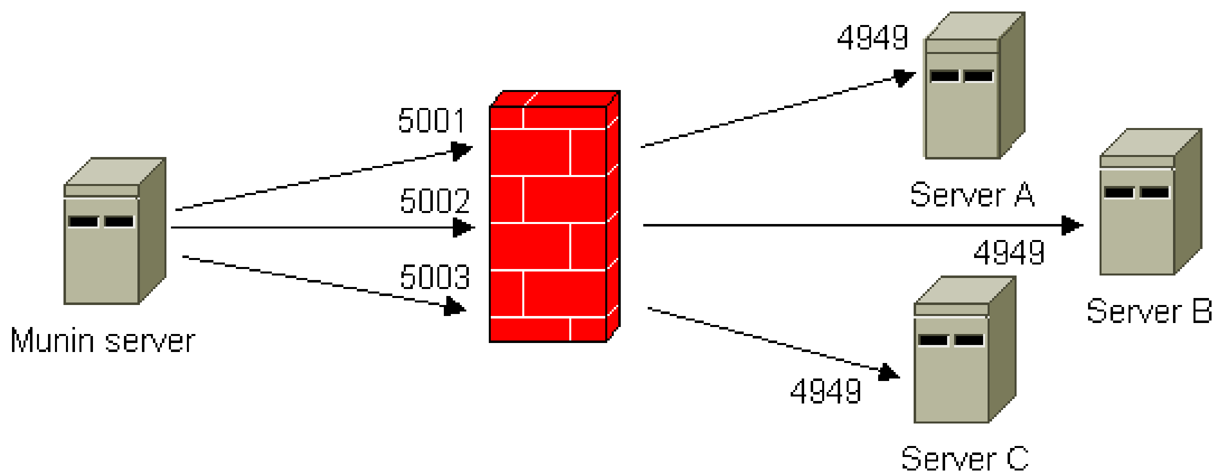
The illustration below shows the principle. By using SSH tunneling only one SSH connection is required, even if you need to reach several hosts on "the other side". The Munin server listens to different ports on the localhost interface. A configuration example is included. Note that there is also a FAQ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



2.8.2 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.

The in-between server ("bouncer") assigns and opens different TCP ports pointing to each of the Munin nodes you need to reach. This is really quite identical to regular port forwarding (see *next section below*).



Comparing this method to *SSH tunneling*, using (x)inetd does not require a login account on the bouncer, while the downside is that the access control may be weaker and you might need to open lots of TCP ports.

From `/etc/services` on the bouncer:

```
munin            4949/tcp
munin-server-a   5001/tcp
munin-server-b   5002/tcp
munin-server-c   5003/tcp
```

If you use `inetd`, entries like these must exist in `/etc/inetd.conf` on the bouncer:

```
munin-server-a   stream  tcp     nowait  root   /usr/bin/nc /usr/bin/nc -w 30␣
↪server-a munin
munin-server-b   stream  tcp     nowait  root   /usr/bin/nc /usr/bin/nc -w 30␣
↪server-b munin
munin-server-c   stream  tcp     nowait  root   /usr/bin/nc /usr/bin/nc -w 30␣
↪server-c munin
```

If you use `xinetd`, the `/etc/xinetd.d/` directory on the bouncer needs one file each for the different servers (Server-A, Server-B and Server-C). For easier recognition, it's a good idea to prefix the files with for example "munin-". A sample *munin-server-a* file looks like this (note that the file name equals the "service" directive, and that the destination server and port are given as server_args):

```
service munin-server-a
{
        disable = no
        socket_type      = stream
        wait             = no
        user             = root
        protocol         = tcp
        server           = /usr/bin/nc
        server_args      = -w 30 server-a munin
}
```

The node definitions in *munin.conf* on Munin Master must be configured accordingly:

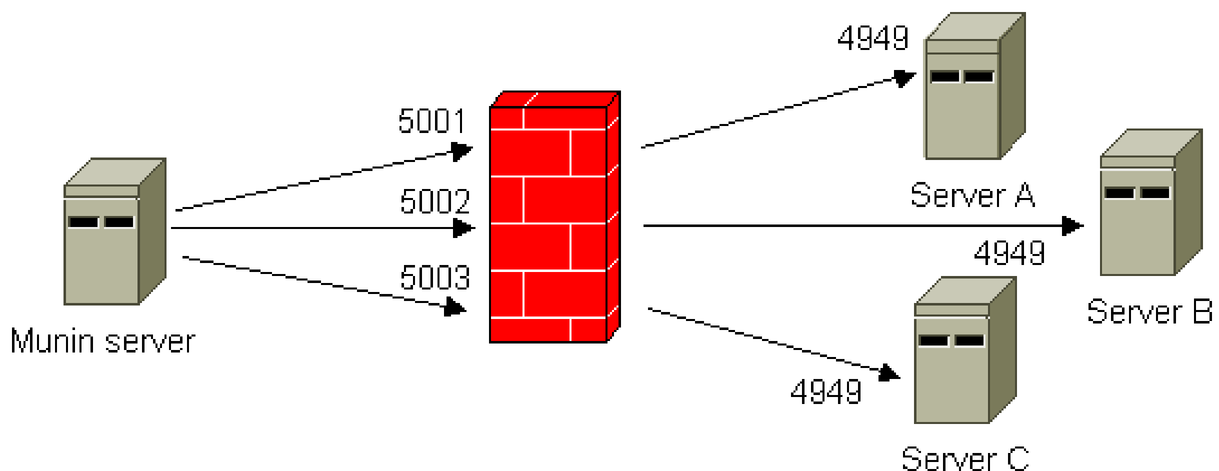
```
[Server-A]
  address bouncer
  port 5001
  use_node_name yes

[Server-B]
  address bouncer
  port 5002
  use_node_name yes

[Server-C]
  address bouncer
  port 5003
  use_node_name yes
```

Note: In this scenario your Munin Nodes have to allow connections (in *munin-node.conf*) **from the IP address of the bouncer** and not from the Munin Master as usual.

2.8.3 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.



The port mapping links TCP port 5001 to server A, port 5002 to server B, and port 5003 to server C.

After you have successfully configured this, the node definitions in *munin.conf* on Munin Master must be configured accordingly:

```
[Server-A]
  address bouncer
  port 5001
  use_node_name yes

[Server-B]
  address bouncer
  port 5002
  use_node_name yes

[Server-C]
```

(continues on next page)

(continued from previous page)

```
  address bouncer
  port 5003
  use_node_name yes
```

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

2.9 Troubleshooting

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

2.9.1 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?

2.9.2 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 *socat* 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.

2.9.3 Check the Logs

Munin's log files (typically below `/var/log/munin/`) are a good source of information while debugging problems.

Log files of a *munin-node*:

- `munin-node.log` and `munin-node-configure.log`: configuration issues and connection messages

Log files of a *munin master*:

- `munin-cgi-graph.log` and `munin-graph.log`: issues with generating graphs
- `munin-cgi-html.log` and `munin-html.log`: issues with generating html content
- `munin-update.log`: fetch configuration and values from a remote *munin-node*
- `munin-limits.log`: generated alarms due to specified *warning*/*critical* thresholds

2.9.4 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 [...]
```

Note: Some plugins require specific capabilities (most notably: *multigraph*). These plugins do not show up in the list, unless the client announces this capability. For example type `cap multigraph` before `list` in order to also find multigraph plugins in the list.

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.

2.9.5 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/writeable 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.

2.9.6 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.

A plugin's graph is missing

Check the following conditions if there is no graph produced for plugin:

- the plugin file (or a symlink to it) is placed in the plugin directory (typically: `/etc/munin/plugins`)
- the executable permission of the plugin file is set
- *munin-node* was restarted after the plugin was added
- user/group is configured for the plugin (if necessary)
- the plugin works as expected locally via *munin-run*
- the *munin master* supports all capabilities required by the plugin (e.g. type `cap multigraph` before `list` in an interactive `nc/telnet` session)
- no related error messages for this plugin appear in `/var/log/munin/munin-update.log` (on the *munin master*)
- an rrd file is created on the *munin master* (e.g. below `/var/lib/munin`)

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"
```

2.9.7 See also

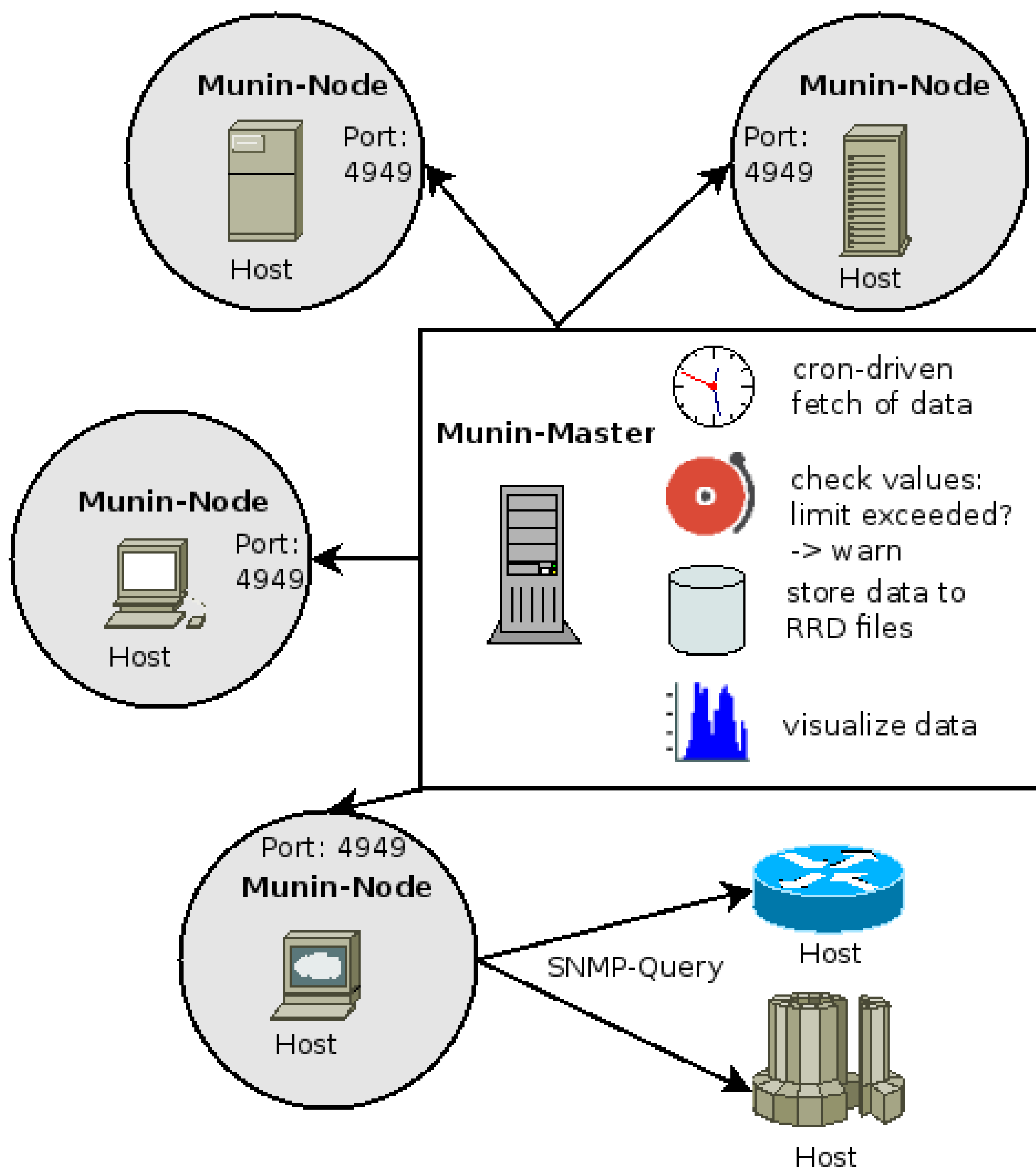
- No Graph FAQ
- *Upgrade notes*

CHAPTER 3

Munin's Architecture

3.1 Overview

Munin has a master-nodes architecture.



3.2 Components

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

3.2.1 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 <https://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 wants 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 RRD[1], 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.

Additionally munin 2.0 contains two more components:

- *munin-graph* is run by *munin-cron*. It generates static graphs in PNG format. It is not needed if *munin-httpd* is used.
- *munin-cgi-graph* is run by a web server, and generates graphs on request.
- *munin-html* is run by *munin-cron*. It generates static HTML pages. It is not needed if *munin-httpd* is used.
- *munin-cgi-html* is run by a web server, and generates HTML pages on request.

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
```

(continues on next page)

[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

(continued from previous page)

```
.
Connection closed by foreign host.
```

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 setes 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

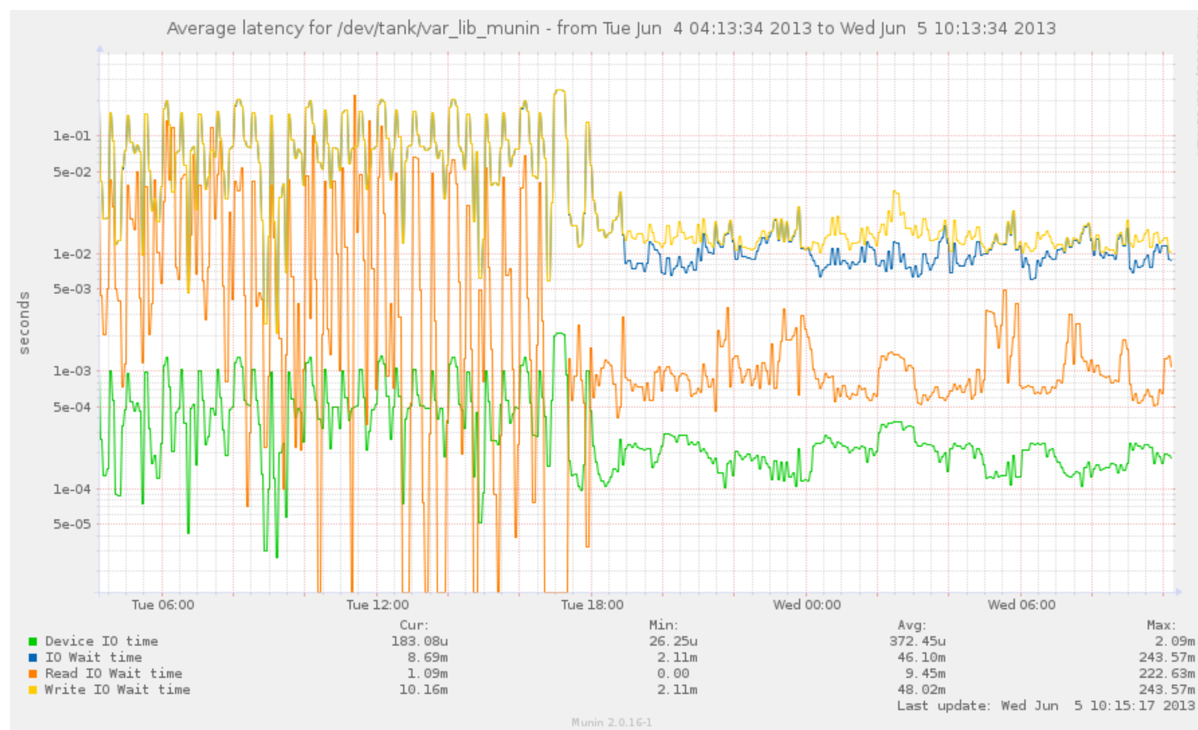
When the master grows big, and has a lot of nodes, there is a risk of disk IO becoming a bottleneck.

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.

Monitoring rrdcached

Munin plugins to monitor `rrdcached` are in the distribution. You can download them also from our repository plugin for stable 2.0 and plugin from master branch You need to *configure your munin-node* to run this plugin as group rrdcached.

3.2.2 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. On machines without native support for Perl scripting you can use munin-c, which is a C rewrite of munin node components. (Look for the details in the munin-c chapter.)

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-polled is difficult and tedious. It even works against of one of the real strength 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.

Theses 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.

3.2.3 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*.

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_DIRTYCONFIG Indicates whether the master is able to understand the *dirtyconfig protocol*.

Table 1: Values

Value	Description
0	Master does not understand `value` lines that are returned within a `config` response.
1	Master is able to consume `value` lines right after reading the configuration from a plugin.

MUNIN_CAP_MULTIGRAPH Indicates whether the master is able to understand the *multigraph* keyword.

Table 2: Values

Value	Description
0	Master does not understand the `multigraph` keyword.
1	Master does understand the `multigraph` 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
```

(continues on next page)

(continued from previous page)

```
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.

See also

- *Environment variables in plugin configuration*

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
```

(continues on next page)

(continued from previous page)

```
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"
```

(continues on next page)

(continued from previous page)

```
  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.

```
# activating a simple plugin
ln -s /usr/share/munin/plugins/cpu /etc/munin/plugins/
# activating a wildcard plugin
ln -s /usr/share/munin/plugins/if_ /etc/munin/plugins/if_eth0
# restart munin-node with your distribution's tools (e.g. systemctl or service)
service munin-node restart
```

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 every time 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 it 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 overitten 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.

disable_autoconf <boolean> If set to True, ignore plugin when running munin-node-configure. This prevents the plugin even when possibly be supported on the system to be installed.

Default: False

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
```

(continues on next page)

(continued from previous page)

```
[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.

Inheritance

In the plugin configuration file(s), values are inherited. Values assigned in sections with more specific expressions have higher priority.

This means that values from `[foo_bar_*]` have precedence over values from `[foo_*]`, regardless of order in the plugin config file.

Non-conflicting values

Consider the following example for a plugin called `dummy_foo_gazonk`:

```
[dummy_*]
env.test1 foo

[dummy_foo_*]
env.test2 baz
```

In this case, the resulting environment values are:

```
test1 = foo
test2 = baz
```

Conflicting values

Another example for the plugin called `dummy_foo_gazonk`:

```
[dummy_*]
env.test1 foo

[dummy_foo_*]
env.test1 bar
env.test2 baz
```

As the more specific `env.test1` has priority, these are the result values:

```
test1 = bar
test2 = baz
```

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
```

Writing a munin plugin

Tutorials:

- *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 Foobar 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
                ;;
```

(continues on next page)

(continued from previous page)

```
        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 htem 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.

3.2.4 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

3.3 Protocols

3.3.1 The Munin Protocols

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

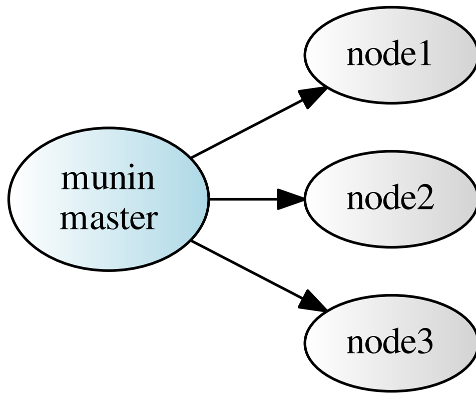
Introduction

Contents on this page will focus on already implemented features. For *proposals* and *ideas* look in the Wiki.

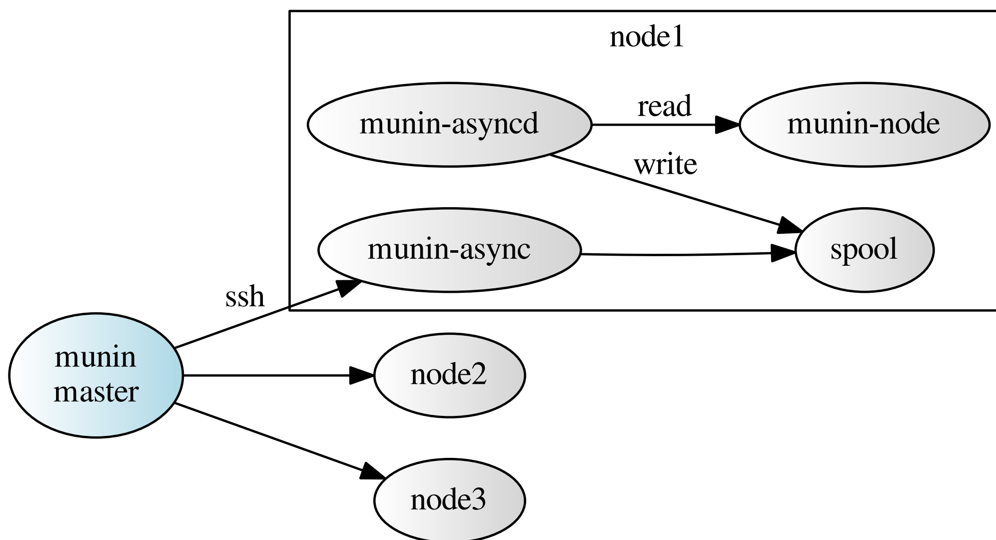
Concepts

Fetching Data

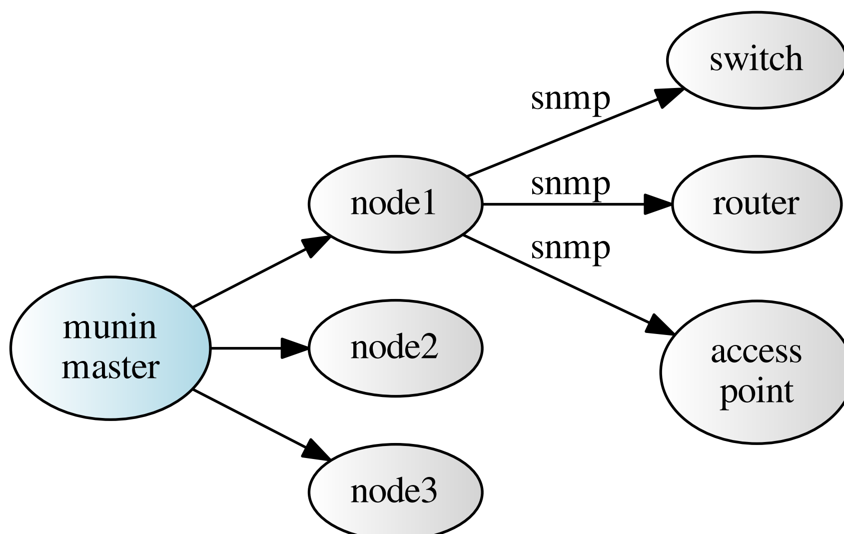
Poller-based monitoring infrastructure



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*

3.4 Syntax

3.4.1 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 https://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)

Note: When loaning data from other graphs, the `{fieldname}.type` must be set to the same data type as the original data. If not, Munin default to searching for gauge files, i.e. files ending with `-g.rdd`. See *dbdir* for the details on RRD filenames.

3.5 API

3.5.1 Munin's API

CHAPTER 4

Munin Installation

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

4.1 Prerequisites

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

4.1.1 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.

4.1.2 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.

4.2 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.

4.2.1 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 the systems package manager, typical examples being:

4.2.2 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*.

4.2.3 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
```

4.3 Initial Configuration

4.3.1 Node

Plugins

Decide which plugins to use. The munin node runs all plugins present in CONFDIR/plugins/ (usually /etc/munin/plugins).

The quick auto-plug-and-play solution:

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

See *Using munin plugins* for more details.

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).

4.3.2 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

4.4 Webserver Configuration

4.4.1 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, you may want to take a look at the following examples:

Mode of Operation	Example Configurations
Generate graphs and HTML pages on demand (recommended)	*apache*
Periodically generate graphs and HTML pages	*apache* / *nginx*
Proxy connections to separate *munin-httpd* process (Munin 2.999 or later)	*apache* / *nginx* / *lighttpd*

4.5 Upgrade Notes

4.5.1 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.

4.5.2 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.

CHAPTER 5

Munin Advanced

5.1 TLS Setup

If your Munin installations reside in a hostile network environment, or if you just don't want anyone passing by with a network sniffer to know the CPU load of your Munin nodes, a quick solution is to enable Munin's built-in Transport Layer Security (TLS) support. Other tricks involve using *SSH tunnels* and key logins, methods "outside of" Munin.

5.1.1 Requirements

In order for this to work you need the Perl package `Net::SSLEay` available. If you are running Debian or Ubuntu this is available in the package `libnet-ssleay-perl`.

5.1.2 Scenarios

The test setups described below consist of two servers, *Aquarium* and *Laudanum* (the Norwegian names of two of the Roman fortifications outside the village of Asterix the Gaul). *Laudanum* is a Munin master and *Aquarium* is a Munin slave.

Non-paranoid TLS setup

This first setup will only provide TLS encrypted communication, not a complete verification of certificate chains. This means that the communication will be encrypted, but that anyone with a certificate, even an invalid one, will be able to talk to the node.

First of all, you must create an X.509 key and certificate pair. That is somewhat outside the scope of this howto (it should be inside the scope, if anyone is willing to include the needed openssl commands and commentary that would be good -janl), but this link explains it in detail. On a Debian system, you can install the ssl-cert package, which automatically creates a dummy key/certificate pair, stored as `/etc/ssl/private/ssl-cert-snakeoil.key` and `/etc/ssl/certs/ssl-cert-snakeoil.pem`. Please note that the permissions on the key file must be restrictive, e.g. `chmod 700`.

Instead of creating your own CA and certificates, you may want to use *Let's Encrypt* instead.

On the Munin master

To make *munin-update* on *Laudanum* do TLS enabled requests, add the following to *munin.conf*:

```
tls enabled
tls_verify_certificate no
tls_private_key /etc/ssl/private/ssl-cert-snakeoil.key
tls_certificate /etc/ssl/certs/ssl-cert-snakeoil.pem
```

And, believe it or not, that's actually it on the Munin-master side. If we run *munin-update* now, we see that the traffic is now encrypted after the command STARTTLS has been issued.

On the Munin node

Even though we've now asked the Munin master to perform all its requests in TLS mode, but anyone can still request data from the Munin node in plain text. So how should we restrict that? Well, that's quite simple as well! Add the following lines to *munin-node.conf*, replacing the paths to the certificate and key as appropriate for your server. Don't forget to restart the *munin-node* process afterwards.

```
tls enabled
tls_verify_certificate no
tls_private_key /etc/ssl/private/ssl-cert-snakeoil.key
tls_certificate /etc/ssl/certs/ssl-cert-snakeoil.pem
```

This will let munin-node accept TLS encrypted communication, but will not check the validity of the certificate presented.

Now, when we try to pump the Munin node for information without starting TLS, this is what happens:

```
laudanum:~# telnet 192.168.1.163 4949
Trying 192.168.1.163...
Connected to 192.168.1.163.
Escape character is '^]'.
# munin node at aquarium
list
# I require TLS. Closing.
Connection closed by foreign host.
laudanum:~#
```

TLS configuration with complete certificate chain

If we switch to a stricter mode, munin-node will only accept update requests from a master presenting a certificate signed by the same CA as its own certificate.

For this setup, the tools provided with OpenSSL can be used to create a CA (Certificate Authority) and one certificate per server signed by the same CA. Creating your own CA should be more that sufficient, unless you really want to spend money on certificates from a real CA. Remember that the "common name" of the server certificate must be the host's fully qualified domain name as it is known in DNS.

The TLS directives are the same on both master and node. This setup requires that both key/cert pairs are signed by the same CA, and the CA certificate must be distributed to each Munin node. Also note that the passphrase protection must be removed from the keys so that the *munin-update* and *munin-node* processes won't require manual intervention every time they start.

On the Munin master

This extract is from *munin.conf* on the master, *Laudanum*:

```
tls paranoid
tls_verify_certificate yes
tls_private_key /etc/opt/munin/laudanum.key.pem
tls_certificate /etc/opt/munin/laudanum.crt.pem
tls_ca_certificate /etc/opt/munin/cacert.pem
tls_verify_depth 5
```

On the Munin node

This extract is from *munin-node.conf* on the node, *Aquarium*:

```
tls paranoid
tls_verify_certificate yes
tls_private_key /etc/opt/munin/aquarium.key.pem
tls_certificate /etc/opt/munin/aquarium.crt.pem
tls_ca_certificate /etc/opt/munin/cacert.pem
tls_verify_depth 5
```

What to expect in the logs

Note that log contents have been formatted for readability.

In munin-update.log (in versions above 1.4.4, the TLS lines only show up in debug mode):

```
Starting munin-update
Processing domain: aquarium
Processing node: aquarium
Processed node: aquarium (0.05 sec)
Processed domain: aquarium (0.05 sec)
[TLS] TLS enabled.
[TLS] Cipher `AES256-SHA'.
[TLS] client cert:
    Subject Name: /C=NO/ST=Oslo/O=Example/CN=aquarium.example.com/
↪emailAddress=bjorn@example.com\n
    Issuer  Name: /C=NO/ST=Oslo/O=Example/CN=CA master/emailAddress=bjorn@example.
↪com
Configured node: aquarium (0.07 sec)
Fetched node: aquarium (0.00 sec)
connection from aquarium -> aquarium (31405)
connection from aquarium -> aquarium (31405) closed
Munin-update finished (0.14 sec)
```

In munin-node.log, something like will show up (in versions above 1.4.4, the TLS lines only show up in debug mode):

```
CONNECT TCP Peer: "192.168.1.161:2104" Local: "192.168.1.163:4949"
TLS Notice: TLS enabled.
TLS Notice: Cipher `AES256-SHA'.
TLS Notice: client cert:
    Subject Name: /C=NO/ST=Oslo/O=Example/CN=laudanum.example.com/
↪emailAddress=bjorn@example.com\n
    Issuer  Name: /C=NO/ST=Oslo/O=Example/CN=CA master/emailAddress=bjorn@example.
↪com
```

5.1.3 Miscellaneous

Intermediate Certificates / Certificate Chains

It is common that external Certificate Authorities use a multi-layer certification process, e.g. the root certificate signs an intermediate certificate, which is used for signing the client or server certificates.

In this case you should assemble the TLS related files in the following way:

- **tls_certificate:**
    1. the *leaf* certificate (for the client or server)
- **tls_ca_certificate:**
    1. intermediate certificate
    2. root certificate

Selective TLS

If you want to run munin-node on the Munin master server, you shouldn't need to enable TLS for that connection as one can usually trust localhost connections. Likewise, if some of the nodes are on a trusted network they probably won't need TLS. In Munin, TLS is enabled on a per node basis.

The node definitions in *munin.conf* on *Laudanum* looks like this (`tls disabled` for localhost communication):

```
[Group;laudanum]
address 127.0.0.1
use_node_name yes
tls disabled

[Group;aquarium]
address 192.168.1.163
use_node_name yes
```

From the source code, it seems you can even use different certificates for different hosts. This, however, has not been tested for the purpose of this article.

Let's Encrypt

You may want to use certificates from the Let's Encrypt CA. Technically they work fine. But please note, that you will not be able to restrict access to specific peers. Instead all users of *Let's Encrypt* will be able to connect to your nodes and your nodes will be unable to distinguish between *your* master and *any other* master connecting with a certificate from the CA.

Thus by using certificates from *Let's Encrypt* you are following a *non-paranoid* approach.

5.2 Advanced Network

- *Monitoring "unreachable" hosts*
- HOWTO Monitor Windows

5.3 Per plugin custom rrd sizing

Choosing the RRD sizing is possible via the config option *graph_data_size* since 1.4.0, but for 2.0 there is also a custom format.

Note: The **configuration should be done on the Munin Master** (*munin.conf*) as the plugins usually do not integrate this config option in their setup!

5.3.1 Objective

The object of this extension is to help with two issues:

- Per default there are only two values `normal` and `huge`. For some users this isn't enough, specially coupled with the *update_rate* option. A new special value `custom`, followed by its definition should offer as much flexibility as anyone needs.
- The RRD sizing is global. Since *update_rate* is per plugin, this should also be per plugin.

5.3.2 The custom format

There are 2 custom formats: the *computer* one, and the *human-readable* one. The *human-readable* one will be converted on-the-fly to the *computer* one.

computer-readable

The format is comma-separated:

```
full_rra_nb, multiple_1 multiple_rra_nb_1, multiple_2 multiple_rra_nb_2, ...␣
↪multiple_N multiple_rra_nb_N
```

- `multiple_N` is the step of the RRA, in multiple of the *update_rate* of the plugin.
- `multiple_rra_nb_N` is the number of RRA frames to keep. The total time amount is function of `multiple_N`.

In this format the original values `normal`/`huge` can be translated:

- `normal` meaning `custom 576, 6 432, 24 540, 288 450`
- `huge` meaning `custom 115200`

Note: The first multiple isn't asked, since it's always 1 (full resolution).

human-readable

Editing the computer format is powerful, but not very friendly. So, another format is available that specifies time duration instead of multiples. It is therefore independent of the *update_rate* of the plugin.

The format is still comma-separated, only the elements are translated:

```
time_res_1 for time_duration_1, time_res_2 for time_duration_2, ... time_res_N for␣
↪time_duration_N
```

- `time_res_N` represents the step of the RRA.
- `time_duration_N` represents the time of RRA frames to keep. The actual number of frames is function of `time_res_N`.

The format for both fields is the same : a number followed by a unit like **134d** or **67w**.

The units are case sensitive and mean:

- `s`: second

- `m`: minute (60s)
- `h`: hour (60m)
- `d`: day (24h)
- `w`: week (7d)
- `t`: month (31d)
- `y`: year (365d)

In this format the original values `normal`/`huge` can be translated:

- `normal` meaning `custom 2d, 30m for 9d, 2h for 45d, 1d for 450d`
- `huge` meaning `custom 400d`

Note: The 2 last units (`t` and `y`) are a fixed number of days. They do **not** take the real number of days in the current month.

5.3.3 Notes

- The RRA is always created with a 10% increase so you can really compare on 10% of the period instead of just the last value.

5.3.4 Issues

- If a human readable config value isn't a multiple of *update_rate*, no graph should be emitted, so the user is immediately alerted of his misconfiguration.
- The first number always represent the full resolution: *update_rate*.

CHAPTER 6

How-to Collection

Here we collect how-to articles which do not fit into the other chapters of the Munin Guide.

6.1 How to remove spikes

If a plugins field is initially created with datatype `COUNTER` and no declaration of a `.max` value, there may occur spikes in the back-end RRD database when the counter goes back to zero. Here we show a simple way to fix this.

6.1.1 Set a MAX value

First, the plugins config **must** be enhanced to present a `.max` value.

Once this has been accomplished and the Munin master has polled the plugin at least once, the masters datafile will have the `.max` value stored for the plugin.

Then the following perl 1-liner can be used to update all matching RRDs. Replace *DOMAIN* with the name of the Munin domain being polled, and replace *PLUGIN* with the name of the plugins RRD database files that need updating.

```
cd /var/lib/munin/DOMAIN
perl -ne 'next unless /:PLUGIN/; if (/.*;(\S+):(\S+)\.(\S+)\.max\s+(\d+)/)
↪{foreach (glob "$1-$2-$3-?.rrd") {print qq{File: $_\tMax: $4\n};qx{rrdtool tune
↪$_ -a 42:$4};qx{rrdtool dump $_ > /tmp/rrdtool-xml};qx{mv $_ $_.bak};qx{rrdtool␣
↪restore -r /tmp/rrdtool-xml $_};qx{chown munin:munin $_}}}' ../datafile
```

An example, updating the *MAX* value in all the `xvm_` plugins that have been updated to report a `.max` value:

```
cd /var/lib/munin/example.com
perl -ne 'next unless /:xvm_/; if (/.*;(\S+):(\S+)\.(\S+)\.max\s+(\d+)/) {foreach␣
↪(glob "$1-$2-$3-?.rrd") {print qq{File: $_\tMax: $4\n};qx{rrdtool tune $_ -a 42:
↪$4};qx{rrdtool dump $_ > /tmp/rrdtool-xml};qx{mv $_ $_.bak};qx{rrdtool restore -
↪r /tmp/rrdtool-xml $_};qx{chown munin:munin $_}}}' ../datafile
File: ic5.example.com-xvm_arch4-xvm_arch4_read_bytes-c.rrd    Max: 34359738352
File: ic6.example.com-xvm_arch3-xvm_arch3_read_bytes-c.rrd    Max: 34359738352
File: ic3.example.com-xvm_dwork-xvm_dwork_write_bytes-c.rrd   Max: 25769803764
```

6.1.2 Modify the RRD files concerned

Note: The following is a distillation of the process, outlined[1] by Greg Connor in 2004-05-06.

Here is a quick recipe for getting rid of spikes.

In this example *dnscache* is the name of the plugin and *dns1.example.com* is one of the affected hosts.

The desired MAX value is 1000.

Verify that the plugin now sets the right `.max` parameter. Fix the plugin if needed.

```
# cd /var/lib/munin
# cat datafile | grep dnscache | grep max
```

View the rrd file with "rrdtool info"

```
# rrdtool info DNS/dns1.example.com-dnscache-cache-c.rrd
```

to look for

```
ds[42].max = NaN
```

This indicates that the plugin didn't have a `.max` defined at the time the rrd file was started. If you don't mind losing the data, you can delete the RRD files at this point and the new ones will be created with the right max.

If you want to save the data, first modify each .rrd file so that it has a max for datasource 42 (not sure why it is always 42, probably a tribute to d.adams) (Best to do this right after an update like at :06 :11 or so)

```
# bash
# for j in `find /var/lib/munin -name "*dnscache*-c.rrd"`; do \
rrdtool tune $j -a 42:1000;
done
```

Note: Replace 1000 with the desired MAX value.

Finally, dump each rrd file to xml and restore with -r flag. There will be some output to let you know which data points were dropped and replaced with NaN.

```
# bash
# for j in `find /var/lib/munin -name "*dnscache*-c.rrd"`; do
rrdtool dump $j > /tmp/xml ;
mv $j $j~ ;
rrdtool restore -r /tmp/xml $j;
chown munin:munin $j ;
done
```

If you are impatient, rebuild one host's graphs and look at it, or just wait 5 min and check.

```
# su munin -c "/usr/share/munin/munin-graph --nolazy --host DNSOverview "
```

[1] See the post to munin-user mailing list by Greg Connor.

CHAPTER 7

Developing Munin

Guidelines for developing Munin.

7.1 Munin development environment

7.1.1 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

7.1.2 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.

7.1.3 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
```

7.1.4 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
```

7.1.5 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
```

7.1.6 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 definitions 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
```

7.1.7 Run plugins in a sandbox

Very simple plugins can be executed directly. Other plugins may require additional environment settings (e.g. for storing the plugin state or for *sourcing the shell helpers*). Testing a symlink configuration also requires a bit of manual preparation.

Many non-trivial situations can be simplified with the sandbox plugin wrapper `dev_scripts/plugin`:

```
dev_scripts/plugin run ./my_foo_plugin
dev_scripts/plugin run if_ suggest
dev_scripts/plugin run_as if_eth0 if_
MUNIN_CAP_MULTIGRAPH=1 dev_scripts/plugin run diskstats config
MUNIN_CAP_DIRTYCONFIG=1 dev_scripts/plugin run df
```

7.1.8 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.

7.2 Munin development tests

7.2.1 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.

7.3 Data Structures

The following documentation discusses the flow, handling and storage of data used by the various Munin components. This includes partially internal processing details, in order to provide a good understanding of data structures within Munin for interested developers.

7.3.1 Munin Node

The Munin Node parses its configuration once during startup.

Node Configuration

The Node configuration specifies the details of the node service, which is responding to requests from the Munin Master.

The Node configuration is read from plain text files (e.g. `/etc/munin/munin-node.conf`). Its content is stored in a simple perl object (usually called `$config`). All settings are available as object attributes (e.g. `$config->{default_plugin_user}`).

Configuration settings with different names (for compatibility over multiple Munin releases) are normalized while parsing (see `Munin::Node::Config::_parse_line`). One example for normalization is `hostname` and `host_name` - both are normalized to `fqdn`.

Plugin Configuration

During the startup of the Munin Node server, the list of plugins (e.g. below `/etc/munin/plugins`) ist collected once. Additionally the plugin configuration files (e.g. below `/etc/munin/plugin-conf.d`) are parsed. Afterwards all configuration settings (including wildcard configurations) for all locally enabled plugins are stored in the hash `$config->{sconf}`.

Configuration settings for a specific plugin are accessible as a hash (`$config->{sconf}{$plugin_name}`).

7.3.2 Munin Master

Periodic operations of the Munin Master are executed by `munin-cron` (via `munin-update` and `munin-limits`). Web resources (html and graphs) are generated by `munin-html` and `munin-graph`. Delivery happens either directly (via CGI) or indirectly via `munin-httpd`.

Master Configuration

The Master configuration specifies the details of the update operation (collecting data from nodes), outbound connection details, resource locations, logging and process management.

The Master configuration is read from plain text files (e.g. `/etc/munin/munin.conf`) by `munin-update` via `Munin::Master::Update->run()`. Its content is stored in the *database storage* in table `param` (see `_db_params_update`).

All other processes query the database, whenever they need a configuration setting.

Service Configurations

The `config` of every plugin is requested periodically via *Munin's network protocol*. The resulting information describe the service, as well as its datasets.

Database Storage

The database is used for exchanging information between different parts of munin. By default the database is an sqlite3 file stored as `/var/lib/munin/datafile.sqlite`.

Table "param"

The *param* table is used for storing Munin Master config parameters. It is populated during every initialization of `munin-update` (`Munin::Master::Update::_db_params_update`).

Field	Value
name	name of the configuration setting (e.g. `logdir`)
value	configured value of the configuration setting

Every configuration setting is stored in the database. The default value is stored in case of a missing explicit setting.

Table "grp"

The *grp* table contains all group/member relationships. Nodes and groups can be members of groups. Thus hierarchical relationships with multiple levels are supported.

Field	Value
id	unique ID
p_id	ID of parent group (special group `0`: this group has no parent)
name	name of the group
path	not used anymore

Table "node"

The *node* table stores the identity of a Munin Node.

Field	Value
id	unique ID
grp_id	ID of the group (or `0` if the node is not part of a group)
name	name of the node
path	not used anymore

Table "node_attr"

The *node_attr* table contains all attributes related to a Munin None.

Field	Value
id	ID of the related *node*
name	name of the attribute (e.g. `use_node_name`)
value	string representation of the value (e.g. `1`)

Table “service”

The *service* table stores the identity of the *service* datastructure.

Field	Value
id	unique ID
node_id	ID of the node this service belongs to
name	unique name of the service within the node
service_title	human readable name of the service
graph_info	human readable description of the service (see *graph_info*)
subgraphs	number of related graphs produced by this service
path	not used anymore

Table “service_attr”

The *service_attr* table contains all attributes related to a service.

Field	Value
id	ID of the related *service*
name	name of the attribute (e.g. `graph_vlabel`)
value	string representation of the value (e.g. `Bytes`)

Table “service_categories”

Multiple category names can be assigned to a *service*.

Field	Value
id	ID of the related *service*
category	category name assigned to the *service*

Table “ds”

Every dataset is described by *Field-level attributes* (see *Nomenclature*). The collected values for a dataset are stored in an RRD file.

Field	Value
id	unique ID
service_id	ID of the *service* to which this dataset belongs
name	unique name of the dataset within the *service*
type	*field type* (e.g. *DERIVE*)
ordr	datasets of a *service* are sorted according to their respective *ordr* attribute
unknown	TODO
warning	description of the *warning* range (TODO: format?)
critical	description of the *critical* range (TODO: format?)
path	not used anymore

Table “ds_attr”

The *ds_attr* table contains all attributes related to a dataset.

Field	Value
id	ID of the related *ds*
name	name of the attribute (e.g. `draw`)
value	string representation of the value (e.g. `AREASTACK`)

Table “url”

The *url* table is used for caching the URL *path* of entities (groups, nodes or services). The path is calculated, when the entity is created.

Field	Value
id	ID of the related entity
type	text specifying the type of the related entity (one of: *group*, *node* and *service*)
path	URL path to be used when accessing the service graphs (e.g. `group1/group2/host/plugin/graphFoo/subGraphBar`)

Table “state”

The *state* table is used for storing the current state of datasets. These values can be used for determining *alert* conditions.

Field	Value
id	ID of the related entity
type	type of the related entity (currently only used as *ds*)
previously_epoch	timestamp of the second to last retrieved value
previously_value	second to last retrieved value
last_epoch	timestamp of the most recently retrieved value
last_value	most recently retrieved value
alarm	current state of the related entity (one of: *unknown*, *critical*, *warning*, “” (empty))
num_unknowns	count of recent successively received `unknown` values (used for delayed “unknown” detection via *<unknown_limit>*)

CHAPTER 8

Developing Plugins

Guidelines for developing Munin plugins

8.1 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.

8.1.1 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
```

Alternatively can also run the plugin in your *development sandbox*.

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␣
↪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%).

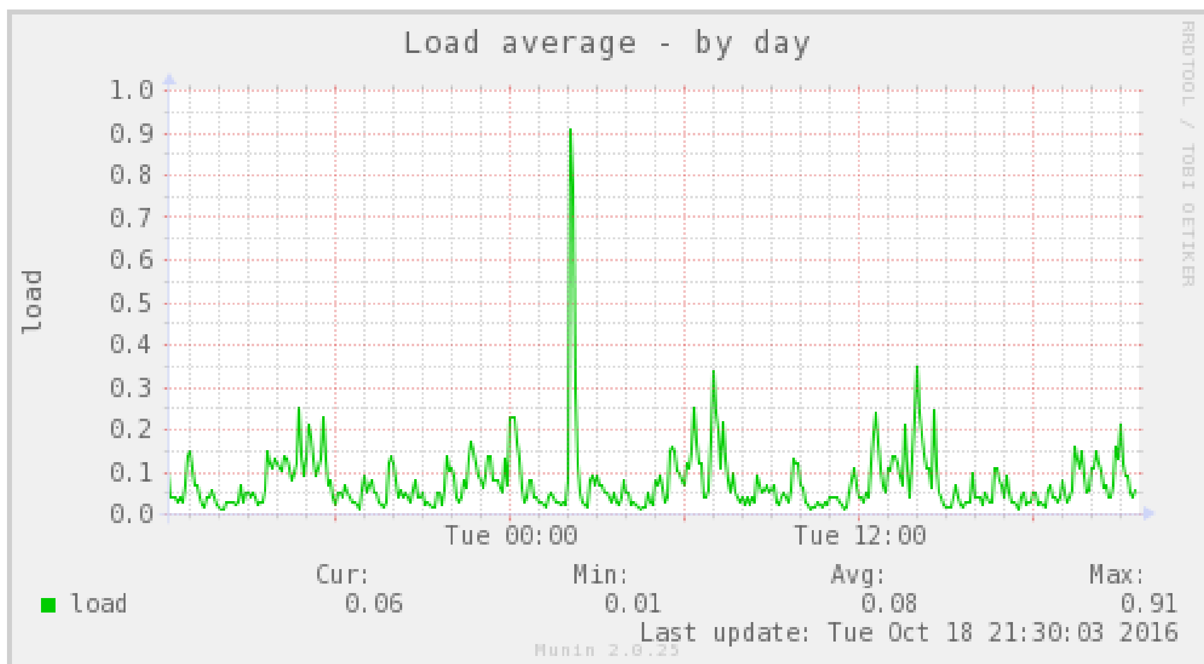
graph_scale no makes munin (rrd) not scale the number. Normally a reading of 1000 would be scaled to `1k` and 1000000 to `1M` (according to scales set with `--base` explained above).

Pick a suitable `graph_category` from the *list of well-known categories*.

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.

8.1.2 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
↪";
```

8.1.3 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 Unixes 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 22763978  191841  ␣
↪ 0    0    0     0       0          0
  eth0:        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 2836088627   ␣
↪4372    0    0     0      0      42          0
  sit0:        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
```

(continues on next page)

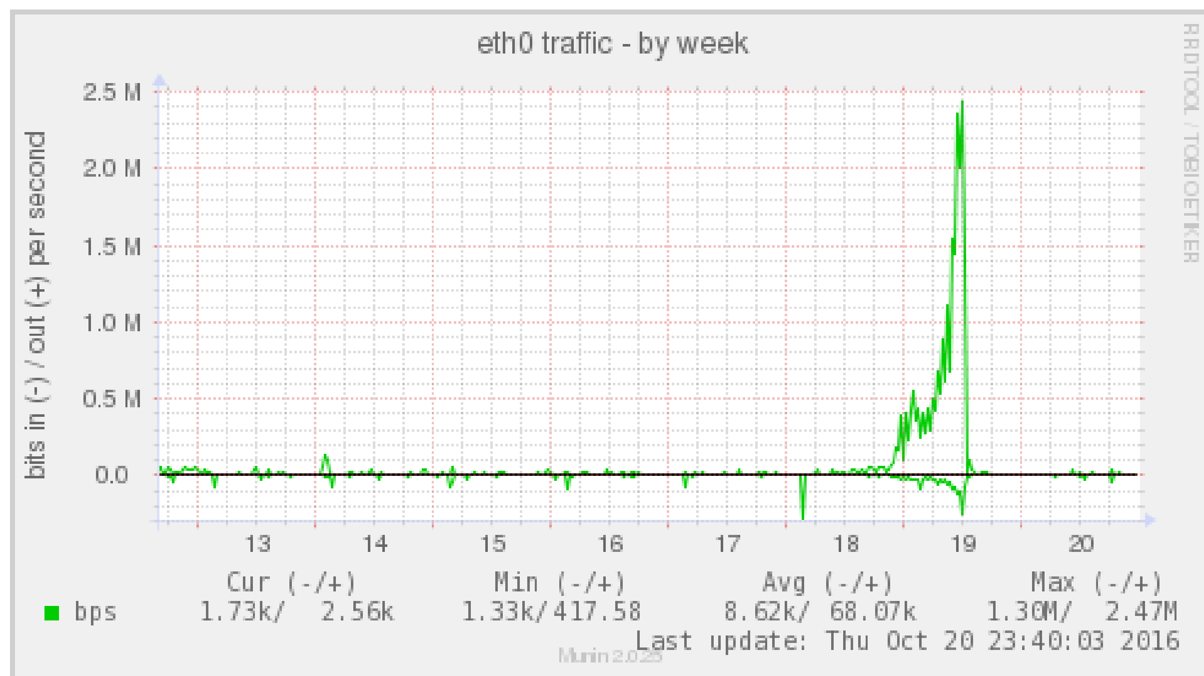
(continued from previous page)

```
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 usuitable 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.

8.1.4 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);

...
```

Python plugin

```
...

def clean_fieldname(text):
    if text == "root":
        return "_root"
    else:
        return re.sub(r"(^[^A-Za-z_]|[^A-Za-z0-9_])", "_", text)

...

fieldname = clean_fieldname(label)

...
```

8.1.5 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 page *PluginConcise*.

8.1.6 See also

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

8.2 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.

8.2.1 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.

8.2.2 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 ]);

```

(continues on next page)

(continued from previous page)

```
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 interogate 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

```

(continues on next page)

(continued from previous page)

```
  #%# 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
```

8.3 Best Current Practices for good plugin graphs

These are some guidelines that will make it easier to understand the graphs produced by your plugins.

8.3.1 Graph labeling

- The different labels should be short enough to fit the graph
- The label should be specific: "transaction volume" is better than "volume", "5 min load average" is better than "load average".
- If the measure is a rate the time unit should be given in the vertical label: "bytes / ${graph_period}" is better than "bytes" or the even worse "throughput".
- All the `graph_*` values specified by the plugin can be used in the title and vlabel values.

`${graph_*}` in plugin output will be magically replaced with the correct value by Munin.

This is a good example of all this: exim_mailstats

8.3.2 Values

Plugins that measure rates should strive to use absolute counters (COUNTER, DERIVE) rather than averages (GAUGE) calculated by an OS tool. E.g. `iostat` on Solaris or `ifconfig` (*see the demonstration plugin*) will output counters rather than short term averages. Counters will be much more correct since Munin can average the measure over its own sample interval instead - this will for example pick up short peaks in loads that Munin might otherwise not see.

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

8.3.3 Graph scaling

```
graph_args --base 1000 --lower-limit 0
graph_scale no
```

See *graph_args* for its documentation.

Choosing a scaler:

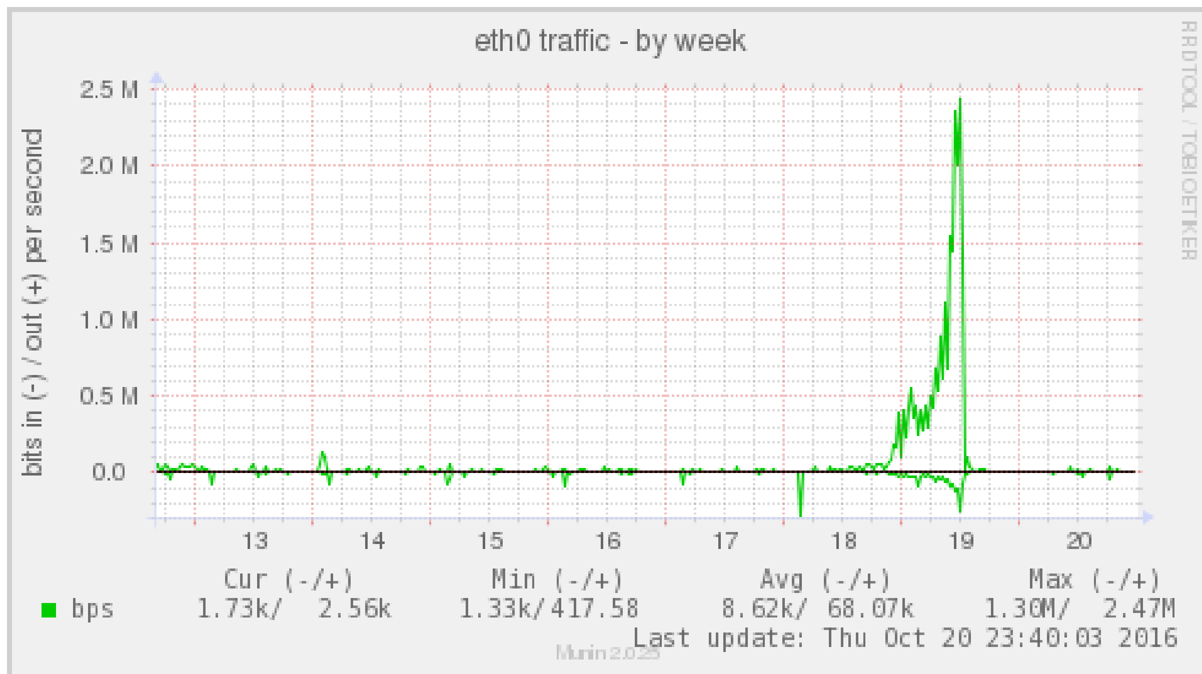
- For disk bytes use 1024 as base (df and other Unix tools still use this though disks are sold assuming 1K=1000)
- For RAM bytes use 1024 as base (RAM is sold that way and always accounted for that way)
- For network bits or bytes use 1000 as base (ethernet and line vendors use this)
- For anything else use 1000 as base

The key is to choose the base that people are used to dealing with the units in. Of the four points above, what units to use for disk storage is most in doubt: the sale of disks the last 10-15 years with 1K=1000 and the recent addition of `--si` options to GNU tools tell us that people are starting to think of disks that way too. But 1024 is ''very" basic to the design of disks and filesystems on a low level so the 1024 is likely to remain.

In addition, most people want to see network speeds in bits not bytes. If your readings are in bytes you might multiply the number by 8 yourself to get bits, or you may leave it to Munin (actually rrd). If the throughput number is reported in `down.value` the `config` output may specify `down.cdef down,8,*` to multiply the down number by 8 (this syntax is known as Reverse Polish Notation).

8.3.4 Direction

For a rate measurement plugin that can report on data both going in and out, such as the if_("eth0") plugin that would report on bytes (or packets) going in and out, it makes sense to graph incoming and outgoing on the same graph. The convention in Munin has become that outgoing data is graphed above the x-axis (i.e., positive) and incoming data is graphed below the y-axis like this:



This is achieved by using the following field attributes. This example assumes that your plugin generates two fieldnames `inputrate` and `outputrate`. The input rate goes under the x-axis so it needs to be manipulated:

```
inputrate.graph no
outputrate.negative inputrate
```

The first disables normal graphing of inputrate. The second activates a hack in munin to get the input and output graphs in the same color and on opposite sides of the x-axis.

8.3.5 Legends

As of version 1.2 Munin supports explanatory legends on both the graph and field level. Many plugins - even the CPU use plugin - should make use of this. The CPU "io wait" number for example will only get larger than 0 if the CPU has nothing else to do in the time interval. Many (nice) graphs will only be completely clear once a rather obscure man page has been read (or in the Linux case perhaps even the kernel source). Using the legend possibilities Munin supports will help this.

Graph legends are added by using the *graph_info* attribute, while field legends use the *fieldname.info* attribute.

8.3.6 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 *well-known categories* for a list of the categories currently in use.

8.3.7 Legal characters

The legal characters in a field name are documented in *Notes on field names*

8.3.8 Documentation

Extended documentation should be added within the documentation header of the plugin script. See our instructions on POD style documentation in the wiki.

8.4 The Concise guide to plugin authoring

First of all you should have read *the HOWTO*. This page is an attempt to be brief and yet complete rather than helpful to a beginner.

8.4.1 What is a plugin?

A plugin is a stand-alone program. In its most common form it is a small perl program or shell script.

The plugins are run by `munin-node` and invoked when contacted by the Munin master. When this happens the `munin-node` runs each plugin twice. Once with the argument `config` to get the graph configuration and once with no argument to get the graph data.

8.4.2 Run Time Arguments

A plugin is invoked twice every update-cycle. Once to emit `config` information and once to `fetch` values. Please see *Data exchange between master and node* for more information about the over-the-wire transactions.

config

This argument is mandatory.

The `config` output describes the plugin and the graph it creates. The full set of attributes you can use is found in the *config reference*.

fetch

This argument is mandatory.

When the node receives a `fetch` command for a plugin, the plugin is invoked without any arguments on the command line and is expected to emit one or more `field.value` attribute values. One for each thing the plugin observes as defined by the `config` output. Plotting of graphs may be disabled by the `config` output.

Please note the following about plugin values:

- If the plugin - for any reason - has no value to report, then it may send the value `U` for **undefined**. It *should* report values for every field it has configured during `config`.
- In 1.3.4 and later: The plugin may back-date values by reporting them on the format `field.value <epoch>:<value>`, where **<epoch>** is the number of seconds since *1970/1/1 00:00* (unix epoch) and **<value>** is the value in the ordinary way. This can be useful for systems where the plugin receives old data and gives the correct time axis on the graph.

8.4.3 Install Time Arguments

Install time is managed by one utility: *munin-node-configure*. It can both *auto configure* host plugins and (given the right options) it will *auto configure* snmp plugins for a specific host.

To do this it looks inside the plugins to determine how much *plug-and-play* ability features have been implemented for the user. For details read the *section about magic markers*.

An automated installation will run `munin-node-configure` and hereby selects the plugins that are ready to run on the specific node. The integration is then done by symlinking them in the service directory (usually `/etc/munin/plugins`).

Every plugin with a *family=auto* magic marker (`#%# family=auto`) will be interrogated automatically with the `autoconf` and perhaps the `suggest` methods. You can change the family list with the `--families` option.

Any `snmp__*` plugins will be auto configured and installed by `munin-node-configure --snmp` with the appropriate arguments.

Every plugin magic marked with *family=snmpauto* and *capabilities=snmpconf* will be interrogated - when appropriately run by `munin-node-configure`.

autoconf

A plugin with a *capabilities=autoconf* magic marker will first be invoked with `autoconf` as the sole argument. When invoked thus the plugin should do one of these two:

1. Print "yes" to signal that the plugin thinks it can be useful on this host
2. Print "no" to signal that the plugin does not think so.

The plugin should always exit 0, even if the response is "no".

If the answer was "yes" and it's not a wildcard plugin, the plugin will be linked into the plugins catalog of munin-node.

Example:

```
# ./load autoconf
yes
```

If the answer is "no" the plugin may give a reason in parentheses. This may help the user to troubleshoot why the plugin is not used/does not work.

```
# ./load autoconf
no (No /proc/loadavg)
```

Note: If a plugin is autoconf-igurable it **SHOULD** not terminate in a uncontrolled way during autoconfiguration. Please make sure that whatever happens it ends up printing a "yes" or a "no" with a reasonable reason for why not.

In particular plugins written in Perl, Python and the like **SHOULD** not require non-core modules without protecting the code section. In perl one would do something like this (from apache_accesses at the time I write this):

```
if (! eval "require LWP::UserAgent;") {
    $ret = "LWP::UserAgent not found";
}
...
if ( defined $ARGV[0] and $ARGV[0] eq "autoconf" ) {
    if ($ret) {
        print "no ($ret)\n";
        exit 0;
    }
}
```

If the plugin is to be distributed with Munin the **SHOULDs** above are **MUSTs**.

suggest

Munin creates one graph per plugin. To create many graphs from one plugin, you can write a wildcard plugin.

These plugins take one or more bits of configuration from the file name it is run as. The plugin is stored as one file in the directory for available plugins, but is linked as multiple files in the directory for enabled plugins. This creates one graph per link name, using just one plugin as source.

Example:

```
/etc/munin/plugins/if_eth0 -> /usr/share/munin/plugins/if_
/etc/munin/plugins/if_eth1 -> /usr/share/munin/plugins/if_
```

As you see: wildcard plugins are easily recognized by their names ending in _.

A wildcard plugin that has a *capabilities=suggest* magic marker will - after `autoconf` - be invoked with `suggest` as the sole argument. It then should examine the system and determine which of the similar things it can report on and output a list of these.

```
# ./if_ suggest
eth0
eth1
```

This means that the plugin can be run as `if_eth0` and `if_eth1`. The plugin will then have to examine what's in C is called `ARGV[0]`, and in perl and shell scripts as `$0` to determine what exactly the start command was.

snmpconf

As stated above a plugin magic marked with *family=snmpauto* and *capabilities=snmpconf* will be invoked with `snmpconf` as the sole argument. A SNMP plugin is by definition a wildcard plugin, it may examine any host that supports SNMP. The name convention for a SNMP plugin is examplified by the df plugin: `snmp__df`. The hostname goes between the two consecutive underlines (_), and when invoked, the plugin may examine `$0`, as any wildcard plugin must, to determine the name of the host it is to work with. It may also contain two wildcards as in `snmp__if_`. Here the index of the network interface appears after the third underline (_) at the end of the string. e.g. `snmp_foo.example.com_if_1`.

On the occasion of being run with `snmpconf` the plugin shall output one or more of the following:

For a plugin that is to monitor a number of (enumerated) items: the items have to be counted and indexed in the MIB and the plugin can express this so:

- The word `number` followed by a OID giving the number of items
- The word `index`, followed by a OID ending with a trailing dot on to the table of indices

Both the `snmp__df` and the `snmp__if_` plugins use this. The df plugin because it monitors multiple storage resources and wants to monitor only fixed disks. It expresses this by asking for the index OID for storage be present. The `snmp__if_` plugin uses both `number` and `index`. The number OID gives the number of network interfaces on the device.

For a plugin named in the pattern analogous to `snmp__if_` each of the indices will be used at the end of the plugin name, e.g. `snmp_switch_if_10` for a device named switch, interface index 10.

- The word `require` followed by an OID or the root of an OID that must exist on a SNMP agent. The OID may optionally be followed by a string or RE-pattern that specifies what sort of response is expected. For a indexed plugin (one that gives "number" and "index" the indecies will be appended to the require OID to check that OID for each indexed item (e.g. a interface)

Example:

```
# ./snmp__if_ snmpconf
number  1.3.6.1.2.1.2.1.0
index   1.3.6.1.2.1.2.2.1.1.
require 1.3.6.1.2.1.2.2.1.5. [0-9]
require 1.3.6.1.2.1.2.2.1.10. [0-9]
```

A require supports any perl RE, so for example one could require (6|32) from a OID to filter out only certain kinds of items.

If all the named items `require` and `number`, `index` given are found (and matched if a RE is given) the plugin will be activated by `munin-node-configure`.

munin-node-configure will send queries to devices that the user has claimed to be interested in, to see if these OIDs exist and have matching values if required. If so the plugin will be linked into the `munin-node` service directory (usually `/etc/munin/plugins`).

8.4.4 Configuration

Plugins are configured through files on each node.

The `munin-node` plugin configuration files reside in `/etc/munin/plugin-conf.d/`. These are used by `munin-node` to determine which privileges a plugin should get (which user and group runs the plugin) and which settings of environment variables should be done for the plugins. Each file in `/etc/munin/plugin-conf.d/` can contain configuration for one or more plugins.

The configuration files are read in alphabetical order and configuration of the last read file overrides earlier configuration.

The format is:

```
[name or wildcard]
  user <username>
  group <group>
  env.<variable name> <variable content>
```

Privileges

Munin usually runs each plugin as an unprivileged user.

To run the plugin as a specific user:

```
[example]
 user someuser
```

To run a plugin with an additional group:

```
[example]
 group somegroup
```

Environment variables

To set the variable `logfile` to `/var/log/example.log`:

```
[example]
 env.logfile /var/log/some.log
```

When using environment variables in your plugins, the plugin should contain sensible defaults.

Example `/bin/sh` code. This adds an environment variable called `$LOG`, and sets it to the value of `$logfile` (from `env.logfile` in the Munin plugin configuration), with a default of `/var/log/syslog` if `$logfile` is empty:

```
#!/bin/sh

LOG=${logfile:-/var/log/syslog}
```

Example configuration

This plugin reads from `/var/log/example.log`, which is readable by user `root` and group `adm`. We set an environment variable for the logfile, and we need additional privileges to be able to read it. Choosing the least amount of privileges, we choose to run the plugin with the group `adm` instead of user `root`.

```
[example]
 group adm
 env.logfile /var/log/example.log
```

Activating a Munin plugin

To activate a plugin it needs to be executable and present in the munin plugin directory, commonly `/etc/munin/plugins`. It can be copied or symlinked here.

Plugins shipped with munin-node are placed in the directory for available Munin plugins, commonly `/usr/share/munin/plugins`. To activate these, make symlinks to the Munin plugin directory, commonly `/etc/munin/plugins`.

8.4.5 Running a Munin plugin interactively

A munin plugin is often run with modified privileges and with a set of environment variables. To run a plugin within its configured environment, use the *munin-run* command. It takes a plugins service link name as the first argument and any plugin argument as the next.

Example (with long lines broken):

```
ssm@mavis:~$ 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
load.warning 10
load.critical 120
graph_info The load average of the machine describes how many processes \
           are in the run-queue (scheduled to run "immediately").
load.info Average load for the five minutes.
```

```
ssm@mavis:~$ munin-run load
load.value 0.11
```

8.5 Plugin Gallery

In the gallery you can browse description and graph images for our Munin Plugins. It is not ready and complete yet. Example graph images are still missing and many plugins have empty documentation pages (due to missing perldoc sections in the plugin script).

Here some examples pages with graph images:

- packages.py - complete
- quota2percent_ - complete
- oracle_sysstat - complete
- ejabberd - Image only, missing perldoc
- apache_activity - Image only, missing perldoc

The HTML-Presentation is auto-generated in a daily cronjob at our project server. It generates the plugins documentation page, that is accessible via *munindoc* otherwise. Example graphs for the plugins have to be placed in our github repositories.

Help from contributors is welcome. Please take a look at the instructions in the next section below.

The gallery has two showrooms. One called Core Collection for the plugins that we deliver with the distribution of Munin-Node and one called 3rd-Party Collection for the plugins from the wild, that were uploaded to our Contrib-Repository. Especially the latter need a lot of documentation work and we are happy if you add info in perldoc format and representative example graph images to the contrib repo. The more descriptive content is there, the more helpful the Plugin Gallery will be.

8.5.1 Categories

The plugins category is the main navigation criterion of the galley. So the first step of the build procedure is the search for the keyword `graph_category` within the plugin scripts and parse the string, that follows in the same line. It makes things easier if you don't use spaces within in the cagetories name. Please use the character *underscore* instead if a separator is needed.

The following pages contain information and recommendations concerning categories:

- *graph-category*
- *Well-known plugin categories*
- *Best Current Practices for good plugin graphs*

8.5.2 Rules for plugin contributors

To make sure that we can auto-generate the portrait pages for each plugin please pay attention to the following instructions.

1. Add **documentation about your plugin in perldoc style** (information about perldoc) to show with *munin-doc* and in the Plugin Gallery on the web. (See *Best Current Practices*).

- Add these sections at the start or end of your plugins script file.

2. Upload the plugins files to Github contrib directory.

- Put the plugins script in a subdirectory named after the software or product that it monitors, e.g. apache, mariadb, postfix. In case of plugins targeting specific operating systems, place these in a subdirectory with that name, e.g. `debian` or `vmware`. The directory's name will act as an outline on **2nd level** of the plugin gallery (within the plugin category index pages).
- **Don't use generic terms as directory name** like "mail". We already use *generic terms* to navigate on the 1st level in the plugin gallery and also in the Munin overview!

3. Choose and upload a Munin generated graph of your plugin for demonstration purpose.

- Take one in original size of the Munin website plugin page. Please do not upload scaled images. Each image should be a file in PNG format.
- Place the image in the subdirectory `example-graphs` of your plugins directory. This is one level deeper in the file hierarchy.
- The name of the image file should begin with the name of your plugins script file followed by `-day.png` for a daily graph, `-week.png` for a weekly graph, `-month.png` for a monthly graph, `-year.png` for a yearly graph, e.g. `cpu-day.png` or `smart_-month.png`.

4. Upload **more image files** to the subdirectory `example-graphs` in PNG-Format if you want **to illustrate** the documentation section **Interpretation**

- The filename of such an additional image should match the following format <*plugins_name*> `-n.png` with `n` standing for a digit between 1 and 9, e.g. `cpu-1.png`

8.5.3 Current state of the project

We have scripts to auto-generate the HTML presentation called "Munin Plugin Gallery" per daily cronjob.

ToDo

Whenever the scripts fails to find the relationship between plugins and categories, we put these into category `other`. It would be good to improve the plugins data concerning the category or to improve the parse method to decrease the number of these unrelated plugins.

8.6 Advanced Topics for Plugin Development

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

8.6.1 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 comment blocks (lines starting with #) 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.

8.6.2 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;
```

(continues on next page)

(continued from previous page)

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

8.6.3 Storing the Plugin's State

Very few plugins need to access state information from previous executions of this plugin itself. The *munin-node* prepares the necessary environment for this task. This includes a separate writable directory that is owned by the user running the plugin and a file that is unique for each *master* that is requesting data from this plugin. These two storage locations serve different purposes and are accessible via environment variables:

- *MUNIN_PLUGSTATE*: directory to be used for storing files that should be accessed by other plugins
- *MUNIN_STATEFILE*: single state file to be used by a plugin that wants to track its state from the last time it was requested by the same master

Note: The datatype *DERIVE* is an elegant alternative to using a state file for tracking the *rate of change* of a given numeric value.

8.6.4 Portability

Plugins should run on a wide variety of platforms.

Shell Plugins

Please prefer */bin/sh* over */bin/bash* (or other shells) if you do not need advanced features (e.g. arrays). This allows such plugins to run on embedded platforms and some *BSD systems that do not contain advanced shells by default. When using */bin/sh* as the interpreter, a feature set similar to busybox's *ash* or Debian's *dash* can be expected (i.e. use *shellcheck -s dash PLUGIN* for code quality checks).

The availability of the following tools can be assumed:

- all the goodies within coreutils
- awk (e.g. gawk)
  - it is recommended to stick to the POSIX set of features (verify via *POSIXLY_CORRECT=1; export POSIXLY_CORRECT*)
- find
- grep
- sed

In order to avoid external tools (e.g. *bc* or *dc*), the shell's arithmetic substition (e.g. *a=$((b + 3))*) should be used for integer operations and *awk* (e.g. *awk '{print $1/1000}'*) for non-trivial calculations.

Python Plugins

Python2 is approaching its end-of-life in 2020 and Python3 was released 2008. Thus new plugins should be written in Python3 only.

Core modules (included in CPython) should be preferred over external modules, whenever possible (e.g. use urllib instead of requests).

8.7 Requirements for 'Vetted Plugins'

This is a draft paper written in 2009 by niclan and dipohl. It should be reviewed, completed (at least merged with the docs for plugin contributors, that were published on github) and then decreed by the developer team.

Purpose is to define requirements for quality plugins that get starred / recommended in the plugin gallery and also may be accepted for the official distribution of munin-node.

8.7.1 Usability

Vetted plugins should work well out of the box, and should autoconf correctly in most cases. The *Concise guide to plugin authoring* describes plugins methods *autoconf* and *suggest*.

Their graph should be significant and easy to comprehend. See the *Guide for good plugin graphs*.

8.7.2 Security

Important demands should be added here..

8.7.3 Documentation

POD Style documentation in the plugin (See *Munin documentation* for the details)

Examples on what information should be included in a plugin POD can be found in apache and buddyinfo (here contribution includes also example graph images for the *Munin Plugin Gallery* :-)

8.7.4 Packaging

Vetted plugins shouldn't conflict with plugins in other packages as well (ie: plugins in munin-node or munin-plugins-extra).

These plugins can go in `$(DESTDIR)/usr/share/munin/plugins`.

CHAPTER 9

Reference

This section contains man pages and other reference material

9.1 Nomenclature

9.1.1 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.

9.1.2 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:

```
acme;webservers;www1.example.net:https_requests
---- ---------- ---------------- --------------
 |     |           |                 |
 |     |           |                 `--- service "https_requests"
 |     |           `-------------------- node "www1.example.net"
 |     `------------------------------- group "webservers"
 `------------------------------------- group "acme"
```

Data source FQN

The fully qualified name to a data source consists of a service fqn, a dot, and a data source name.

Note: A data source normally corresponds to one line in a graph.

Example:

```
acme;webservers;www1.example.net:https_requests.denied
---- ---------- ---------------- -------------- ------
 |     |           |                 |              |
 |     |           |                 |              `--- data source "denied"
 |     |           |                 `------------------ service "https_requests"
 |     |           `------------------------------------ node "www1.example.net"
 |     `----------------------------------------------- group "webservers"
 `----------------------------------------------------- group "acme"
```

9.2 Man pages

9.2.1 munin

DESCRIPTION

Munin is a group of programs to gather data from hosts, graph them, create html-pages, and optionally warn contacts about any off-limit values.

Munin master

The Munin master contains the following command line programs:

- *munin-cron* to run other munin programs every 5 minutes. It is run from cron.
- *munin-update* to gather data from machines running munin-node. Usually run by *munin-cron*.
- *munin-limits* to check for any off-limit values. Usually run by *munin-cron*.
- *munin-httpd* runs the munin master web interface.

Munin 2.0 contains the following additional command line programs:

- *munin-graph* to create graphs from data contained in rrd-files. Usually run by *munin-cron* unless `graph_strategy` is set to "cgi" in *munin.conf*.
- *munin-cgi-graph* is used to provide graph images on demand (if `graph_strategy` is set to "cgi").
- *munin-html* to draw html-pages on an Munin installation. Usually run by *munin-cron* unless `html_strategy` is set to "cgi" in *munin.conf*.
- *munin-cgi-html* is used to provide HTML pages on demand (if `graph_strategy` is set to "cgi").

Munin node

The munin node consists of the following programs

Daemons

- *munin-node* runs on all nodes where data is collected.
- *munin-asyncd* is a daemon that runs alongside a munin-node. It queries the local *munin-node*, and spools the results.

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*,

9.2.2 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.

9.2.3 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.

9.2.4 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.

9.2.5 munin-cgi-graph

DESCRIPTION

The munin-cgi-graph program is intended to be run from a web server. It can either run as CGI, or as FastCGI.

OPTIONS

munin-cgi-graph is controlled using environment variables. See environment variables *PATH_INFO* and *QUERY_STRING*.

Note: The munin-cgi-graph script may be called with the command line options of *munin-graph*. However, the existence of this should not be relied upon.

ENVIRONMENT VARIABLES

The following environment variables are used to control the output of munin-cgi-graph:

`PATH_INFO`
This is the remaining part of the URI, after the path to the munin-cgi-graph script has been removed.

The group, host, service and timeperiod values are extracted from this variable. The group may be nested.

`CGI_DEBUG`
If this variable is set, debug information is logged to STDERR, and to /var/log/munin/munin-cgi-graph.log

`QUERY_STRING`
A list of key=value parameters to control munin-cgi-graph. If QUERY_STRING is set, even to an empty value, a no_cache header is returned.

HTTP_CACHE_CONTROL
If this variable is set, and includes the string “no_cache”, a no_cache header is returned.

HTTP_IF_MODIFIED_SINCE
Returns 304 if the graph is not changed since the timestamp in the HTTP_IF_MODIFIED_SINCE variable.

EXAMPLES

When given an URI like the following:

http://munin/munin-cgi/munin-cgi-graph/example.org/client.example.org/cpu-week.png

munin-cgi-graph will be called with the following environment:

PATH_INFO=/example.org/client.example.org/cpu-week.png

To verify that munin is indeed graphing as it should, you can use the following command line:

```
sudo -u www-data \
PATH_INFO=/example.org/client.example.org/irqstats-day.png \
/usr/lib/munin/cgi/munin-cgi-graph | less
```

The “less” is strictly not needed, but is recommended since munin-cgi-graph will output binary data to your terminal.

You can add the *CGI_DEBUG* variable, to get more log information. Content and debug information is logged to STDOUT and STDERR, respectively. If you only want to see the debug information, and not the HTTP headers or the content, you can redirect the file descriptors:

```
sudo -u www-data \
CGI_DEBUG=yes \
PATH_INFO=/example.org/client.example.org/irqstats-day.png \
/usr/lib/munin/cgi/munin-cgi-graph 2>&1 >/dev/null | less
```

SEE ALSO

See *munin* for an overview over munin.

9.2.6 munin-cgi-html

DESCRIPTION

The **munin-cgi-html** program is intended to be run from a web server. It can either run as CGI, or as FastCGI.

OPTIONS

munin-cgi-html takes no options. It is controlled using environment variables.

ENVIRONMENT VARIABLES

The following environment variables are used to control the output of munin-cgi-html:

PATH_INFO
This is the remaining part of the URI, after the path to the munin-cgi-html script has been removed.

The group, host, service and timeperiod values are extracted from this variable. The group may be nested.

EXAMPLES

PATH_INFO

"/" refers to the top page.

"/example.com/" refers to the group page for "example.com" hosts.

"/example.com/client.example.com/" refers to the host page for "client.example.com" in the "example.com" group

COMMAND-LINE

When given an URI like the following: http://munin.example.org/munin-cgi/munin-cgi-html/example.org

munin-cgi-html will be called with the following environment:

PATH_INFO=/example.org

To verify that munin is able to create HTML pages, you can use the following command line:

```
sudo -u www-data \
PATH_INFO=/example.org \
/usr/lib/munin/cgi/munin-cgi-html
```

SEE ALSO

See *munin* for an overview over munin.

munin-cgi-graph.

9.2.7 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*

For munin 2.0 it additionally runs the following programs (unless configured for CGI):

1. *munin-graph*
2. *munin-html*

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*,

9.2.8 munin-graph

DESCRIPTION

munin-graph script is one of the munin master components run from the *munin-cron* script.

If "graph_strategy" is set to "cron", munin-graph creates static graphs from all RRD files in the munin database directory.

If graph_strategy is set to "cgi", munin-graph will not create graphs. This is the proper setting when you run *munin-httpd*.

OPTIONS

Some options can be negated by prefixing them with "no". Example: –fork and –nofork

--fork
By default munin-graph forks subprocesses for drawing graphs to utilize available cores and I/O bandwidth. Can be negated with –nofork [–fork]

--n <processes>
Max number of concurrent processes [6]

--force
Force drawing of graphs that are not usually drawn due to options in the config file. Can be negated with –noforce [–noforce]

--lazy
Only redraw graphs when needed. Can be negated with –nolazy [–lazy]

--help
View this message.

--version
View version information.

--debug
Log debug messages.

--screen
If set, log messages to STDERR on the screen.

--cron
Behave as expected when run from cron. (Used internally in Munin.) Can be negated with –nocron

--host <host>
Limit graphed hosts to <host>. Multiple –host options may be supplied.

--only-fqn <FQN>
For internal use with CGI graphing. Graph only a single fully qualified named graph,

For instance: –only-fqn root/Backend/dafnes.example.com/diskstats_iops

Always use with the correct –host option.

--config `<file>`
Use <file> as configuration file. [/etc/munin/munin.conf]

--list-images
List the filenames of the images created. Can be negated with –nolist-images. [–nolist-images]

--output-file `| -o`
Output graph file. (used for CGI graphing)

--log-file `| -l`
Output log file. (used for CGI graphing)

--day
Create day-graphs. Can be negated with –noday. [–day]

--week
Create week-graphs. Can be negated with –noweek. [–week]

--month
Create month-graphs. Can be negated with –nomonth. [–month]

--year
Create year-graphs. Can be negated with –noyear. [–year]

--sumweek
Create summarised week-graphs. Can be negated with –nosumweek. [–summweek]

--sumyear
Create summarised year-graphs. Can be negated with –nosumyear. [–sumyear]

--pinpoint `<start,stop>`
Create custom-graphs. <start,stop> is the time in the standard unix Epoch format. [not active]

--size_x `<pixels>`
Sets the X size of the graph in pixels [175]

--size_y `<pixels>`
Sets the Y size of the graph in pixels [400]

--lower_limit `<lim>`
Sets the lower limit of the graph

--upper_limit `<lim>`
Sets the upper limit of the graph

Note: *--pinpoint* and *--only-fqn* must not be combined with any of *--day*, *--week*, *--month* or *--year* (or their negating forms). The result of doing that is undefined.

SEE ALSO

See *munin* for an overview over munin.

munin-cron, *munin-httpd*

9.2.9 munin-html

DESCRIPTION

munin-html is one of the munin master components run from the *munin-cron* script.

If `html_strategy` is unset, or set to "cron", munin-html creates static HTML pages. If set to "cgi", it will not generate static pages, this is the proper setting when *munin-httpd* is used.

OPTIONS

munin-html has one significant option, which configuration file to use.

Several other options are recognized and ignored as "compatibility options", since *munin-cron* passes all options through to the underlying components, of which munin-html is one.

--config `<file>`
    Use <file> as configuration file. [/etc/munin/munin.conf]

--help
    View this message.

--debug
    Log debug messages.

--screen
    If set, log messages to STDERR on the screen.

--version
    View version information.

--nofork
    Compatibility. No effect.

--service `<service>`
    Compatibility. No effect.

--host `<host>`
    Compatibility. No effect.

SEE ALSO

See *munin* for an overview over munin.

munin-cron, *munin-httpd*

9.2.10 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 was introduced after the 2.0.x release series. It replaces the FastCGI scripts munin-cgi-graph and munin-cgi-html, that were used in munin 2.0.

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

9.2.11 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<gt>. Multiple –contact options may be supplied. [unset]

--host <host>
Limit hosts to those of <host<gt>. 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

9.2.12 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.

9.2.13 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 –noparanoia [–noparanoia]

--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
```

(continues on next page)

(continued from previous page)

```
pid_file /var/run/munin/munin-node.pid
background 1
setsid 1

# Which port to bind to;

host [::]
port 4949
user root
group root

# Regexps 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.

9.2.14 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/]

--sconffile <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.

9.2.15 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<gt>. 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

9.2.16 munin.conf

DESCRIPTION

This is the configuration file for the munin master. It is used by *munin-update*, *munin-limits* and *munin-httpd*.

Location in packages is usually `/etc/munin/` while if compiled from source it is often found in `/etc/opt/munin/`.

The structure is:

1. One general/global section
2. Zero or more group section(s)
3. One or more host section(s)

Group and host sections are defined by declaring the group or host name in brackets. Everything under a section definition belongs to that section, until a new group or host section is defined.

Note: As **the global section** is not defined through brackets, it **must be found prior to any group or host sections**. 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`). It must be writable for the user running *munin-cron*. RRD files are placed in subdirectories *$dbdir/$domain/*

htmldir <path>
The directory shown by *munin-httpd*. It must be writable for the user running *munin-httpd*. For munin 2.0: The directory where *munin-html* stores generated HTML pages, and where *munin-graph* stores graphs.

logdir <path>
The directory where munin stores its logfiles (Default: `/var/log/munin`). It must be writable by the user running munin-cron.

rundir <path>
Directory for files tracking munin's current running state. Default: `/var/run/munin`

tmpldir <path>
Directories for templates used by *munin-httpd* (munin 2.0: *munin-html*) to generate HTML pages. Default `/etc/munin/templates`

staticdir <path>
Where to look for the static www files.

cgitmpdir <path>
Temporary cgi files are here. It has to be writable by the cgi user (For Munin stable 2.0 usually `nobody` or `httpd`).

includedir <path>
(Exactly one) directory to include all files from. Default `/etc/munin/plugin-conf.d/`

local_address <address>
Sets the local IP address that *munin-update* should bind to when contacting the nodes. May be used several times (one line each) on a multi-homed host. Should default to the most appropriate interface, based on routing decision.

Note: This directive can be overwritten via settings on lower hierarchy levels (group, node).

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-httpd* 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-httpd* to color the graphs. This option overrides the existing palette. The palette must be space-separated 24-bit hex color code.

graph_period <second>
You can choose the time reference for “DERIVE” like graphs, and show “per minute” => `minute`, “per hour” => `hour` values instead of the default “per second”.

html_dynamic_images 1
Munin HTML templates use this variable to decide whether to use dynamic (“lazy”) loading of images with javascript so that images are loaded as they are scrolled in view. This prevents excessive load on the web server. Default is 0 (off).

max_graph_jobs 6
Available since Munin 1.4.0. Maximum number of parallel processes used by munin-graph when calling rrdgraph. The optimal number is very hard to guess and depends on the number of cores of CPU, the I/O bandwidth available, if you have SCSI or (S)ATA disks and so on. You will need to experiment. Set on the command line with the `-n n` option. Set to 0 for no forking.

munin_cgi_graph_jobs 6
munin-cgi-graph is invoked by the web server up to very many times at the same time. This is not optimal since it results in high CPU and memory consumption to the degree that the system can thrash. Again the default is 6. Most likely the optimal number for `max_cgi_graph_jobs` is the same as `max_graph_jobs`.

cgiurl_graph /munin-cgi/munin-cgi-graph
If the automatic CGI url is wrong for your system override it here.

max_size_x `4000`
The max width of images in pixel. Default is 4000. Do not make it too large otherwise RRD might use all RAM to generate the images.

max_size_y `4000`
The max height of images in pixel. Default is 4000. Do not make it too large otherwise RRD might use all RAM to generate the images.

graph_strategy `<cgi|cron>`
This option is available only in munin 2.0. In munin 2.0 graphics files are generated either via cron or by a CGI process.

If set to "cron", *munin-graph* will graph all services on all nodes every run interval.

If set to "cgi", *munin-graph* will do nothing. Instead graphs are generated by the webserver on demand.

html_strategy `<cgi|cron>`
This option is available only in munin 2.0. In munin 2.0 HTML files are generated either via cron (default) or by a CGI process.

If set to "cron", *munin-html* will recreate all html pages every run interval.

If set to "cgi", *munin-html* will do nothing. Instead HTML files are generated by the webserver on demand. This setting implies `graph_strategy cgi`

max_processes `16`
munin-update runs in parallel.

The default max number of processes is 16, and is probably ok for you. Should be not higher than 4 x CPU cores.

If set too high, it might hit some process/ram/filedesc limits. If set too low, munin-update might take more than 5 min. If you want munin-update to not be parallel set it to 1.

rrdcached_socket `/var/run/rrdcached.sock`
RRD updates are per default, performed directly on the rrd files. To reduce IO and enable the use of the rrdcached, uncomment it and set it to the location of the socket that rrdcached uses.

graph_data_size `<normal|huge|custom>`
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.

With "custom" you can define your own resolution. See *the instruction on custom RRD sizing* for the details.

Changing this directive has no effect on existing graphs

contact.your_contact_name.command `<command>`
Define which contact command to run. See the tutorial *Let Munin croak alarm* for detailed instruction about the configuration.

contact.your_contact_name.text `<text>`
Text to pipe into the command.

contact.your_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".

`domain_order` `<group1> <group2> ..`
Change the order of domains/groups. Default: Alphabetically sorted

GROUP DIRECTIVES

If you want to set directives on the group level you have to start the group section with the groups name in square brackets.

```
[mygroup.mydomain]
```

`node_order` `<node1> <node2> ..`
Changes the order of nodes in a domain. Default: Alphabetically sorted.

`contacts` `<no|your_contact_name1 your_contact_name2 ...>`
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" and then **all** defined contacts will get informed when values go over or below thresholds.

Note: This directive can be overwritten via settings on lower levels of the hierarchy (node, plugin).

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".

use_node_name `<yes|no>`
Overrides the name supplied by the node. Allowed values: "yes" and "no". Defaults to "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*.

Note: This directive can also be used on hierarchy level plugin to change the name by which the plugin 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.

graph_height `<value>`
The graph height for a specific service. Default is 200.

Affects: *munin-httpd*.

graph_width `<value>`
The graph width for a specific service. Default is 400.

Affects: *munin-httpd*.

For a complete list see the reference of *global plugin attributes*.

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.

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*.

For a complete list see the reference of *plugin data source attributes*.

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.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]
```

(continues on next page)

(continued from previous page)

```
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

9.2.17 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 logigng, 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 presence 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
```

(continues on next page)

(continued from previous page)

```
# 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*

9.2.18 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.

9.3 Examples and Templates

9.3.1 Examples

Examples of munin and related configuration are gathered here.

Apache CGI Configuration

This example describes how to generate graphs and HTML files dynamically (on demand) via Apache.

Virtualhost configuration

Add a new virtualhost, using the following example:

```
<VirtualHost *:80>
    ServerName munin.example.org
    ServerAlias munin

    ServerAdmin  info@example.org

    DocumentRoot /var/www

    Alias /munin/static/ /etc/munin/static/
    <Directory /etc/munin/static>
        Require all granted
    </Directory>

    ScriptAlias /munin-cgi/munin-cgi-graph /usr/lib/munin/cgi/munin-cgi-graph
    ScriptAlias /munin /usr/lib/munin/cgi/munin-cgi-html
    <Directory /usr/lib/munin/cgi>
        Require all granted
        <IfModule mod_fcgid.c>
            SetHandler fcgid-script
        </IfModule>
        <IfModule !mod_fcgid.c>
            SetHandler cgi-script
        </IfModule>
    </Directory>

    CustomLog /var/log/apache2/munin.example.org-access.log combined
    ErrorLog  /var/log/apache2/munin.example.org-error.log
</VirtualHost>
```

Apache Cron Configuration

This example describes how to use Apache for delivering graphs and HTML files that were generated via cron.

Virtualhost configuration

Add a new virtualhost, using the following example:

```
<VirtualHost *:80>
    ServerName munin.example.org
    ServerAlias munin

    ServerAdmin  info@example.org
```

(continues on next page)

(continued from previous page)

```
    DocumentRoot /var/www

    Alias /munin/static/ /etc/munin/static/
    <Directory /etc/munin/static>
        Require all granted
    </Directory>

    Alias /munin /var/cache/munin/www
    <Directory /var/cache/munin/www>
        Require all granted
    </Directory>

    CustomLog /var/log/apache2/munin.example.org-access.log combined
    ErrorLog  /var/log/apache2/munin.example.org-error.log
</VirtualHost>
```

Apache Proxy Configuration

This example describes how to run a separate *munin-httpd* process and proxy all requests via Apache to this instance.

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

    # serve static files directly
    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 Proxy Configuration

This example describes how to use lighttpd in front of *munin-httpd*.

Webserver configuration

```
alias.url += ( "/munin-static" => "/etc/munin/static" )
alias.url += ( "/munin"        => "/var/cache/munin/www/" )
```

(continues on next page)

(continued from previous page)

```
$HTTP["url"] =~ "^/munin" {
  proxy.server = (""    => (( "host" => "127.0.0.1", "port" => 4948)))
}
```

Nginx Static Configuration

This example describes how to set up nginx for serving content generated by munin via cron.

Site Configuration

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/fastcgi-graph.sock;
    include fastcgi_params;
}
```

If this is a dedicated Munin server, you might want to redirect the frontpage as well:

```
location = / {
    rewrite ^/$ munin/ redirect; break;
}
```

Nginx Proxy Configuration

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).

Site Configuration

```
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 for good plugin graphs*

--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 *overrun*. (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 \
```

(continues on next page)

(continued from previous page)

```
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
```

(continues on next page)

(continued from previous page)

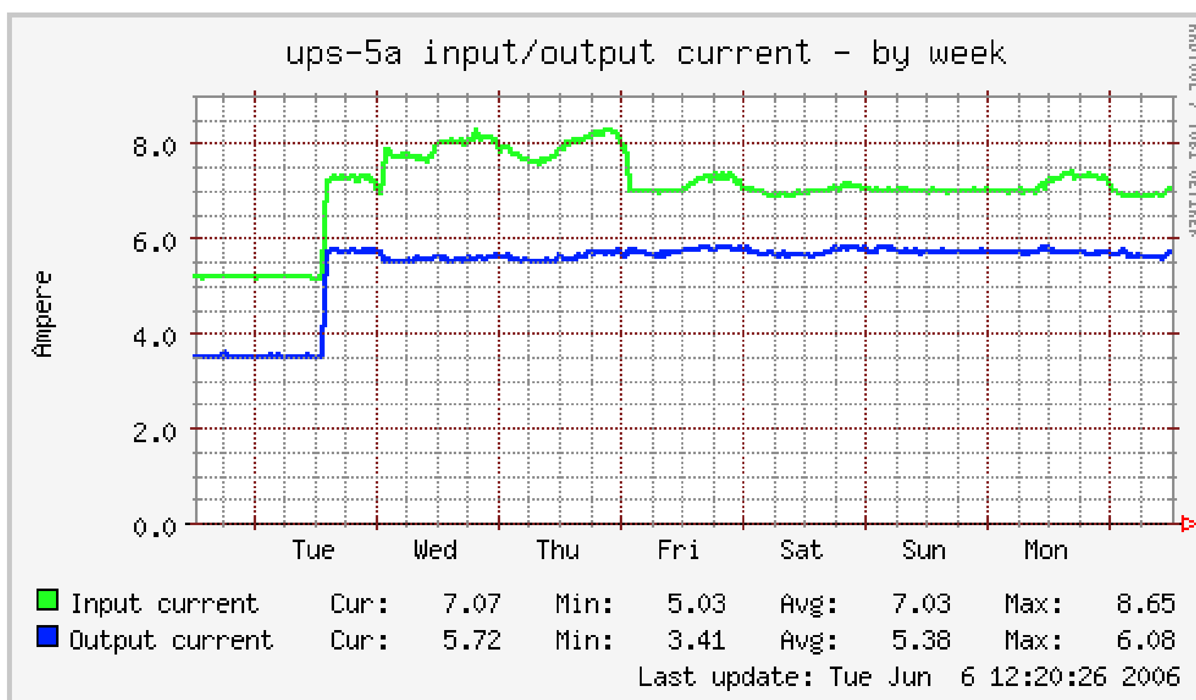
```
   |                 `------------------------- The real plugin's name (symlink)
   `----------------------------------------- The host name from which to seek
↪information
```

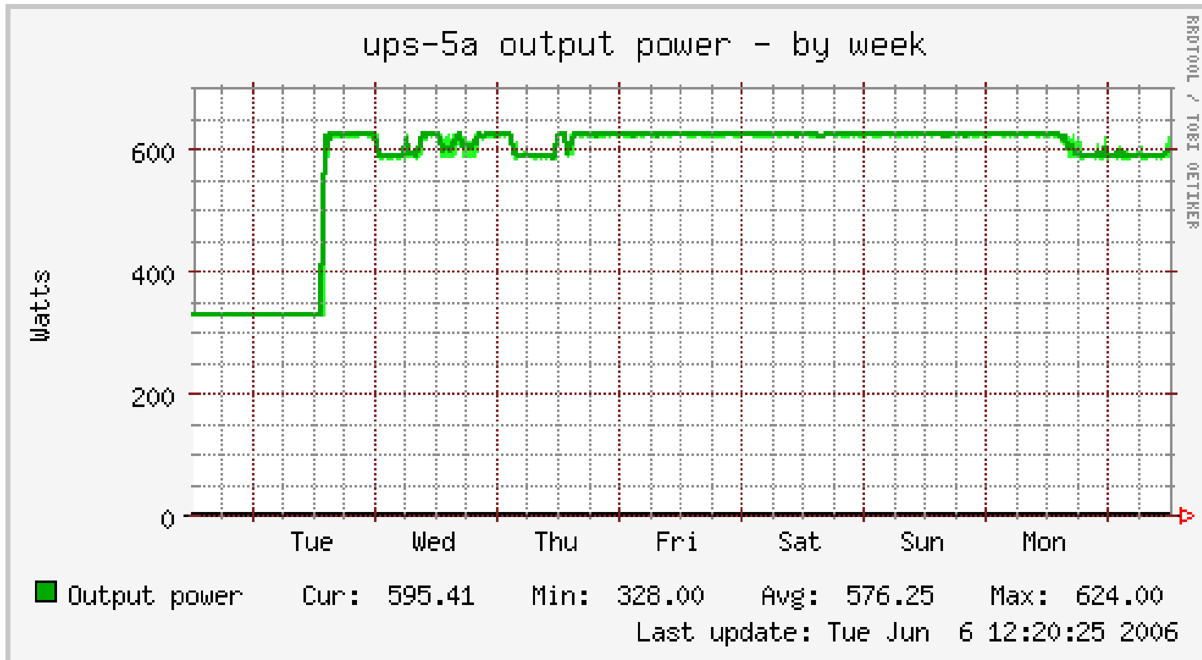
- 22 - 24: Similarly for “outputtotal”.
- 26 - 35: Like the above, but for power instead. Note that this virtual plugin graphs only “one” value, and as such, only “one” “sum” mechanism is used.

Result graphs

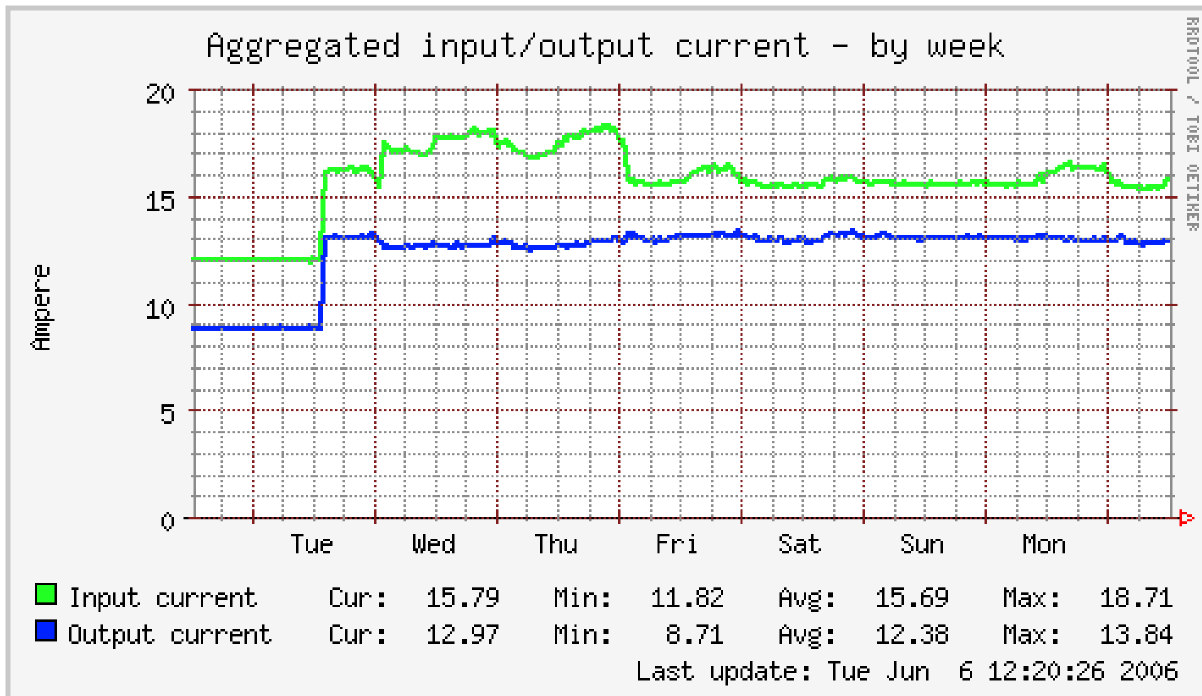
The graphs below show one of the UPSes, and the aggregated values. The graphs used are by week, because they had a nice dip in the beginning of the graphing period :-)

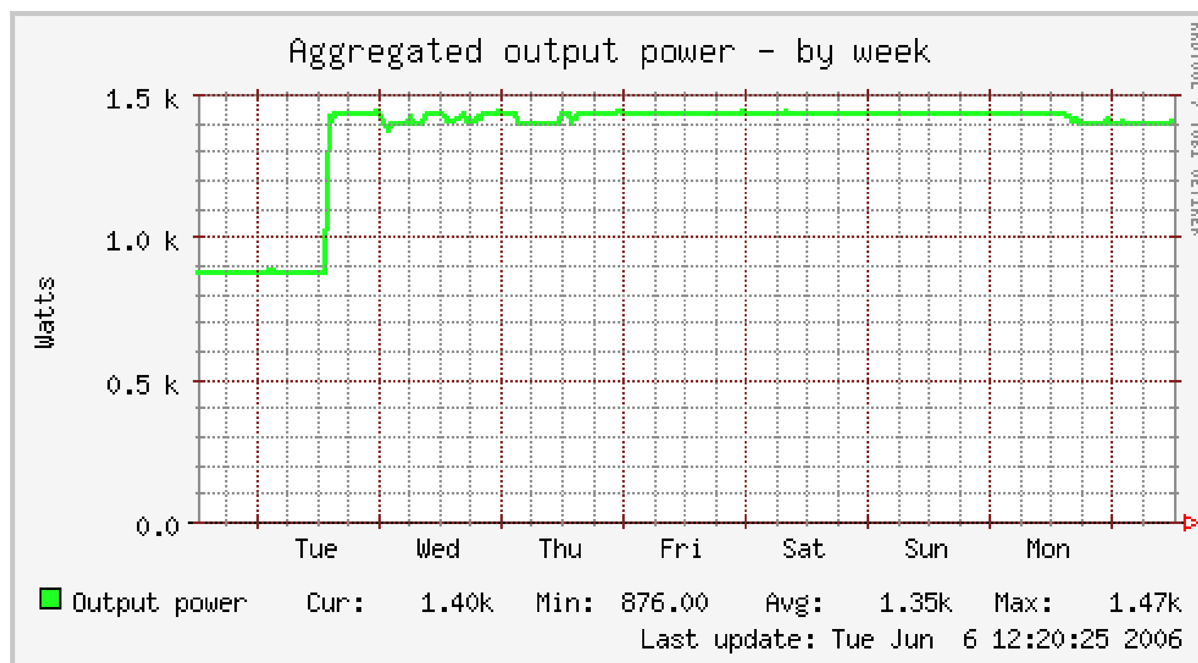
Source graphs for one of the UPSes:





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
```

(continues on next page)

(continued from previous page)

```
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

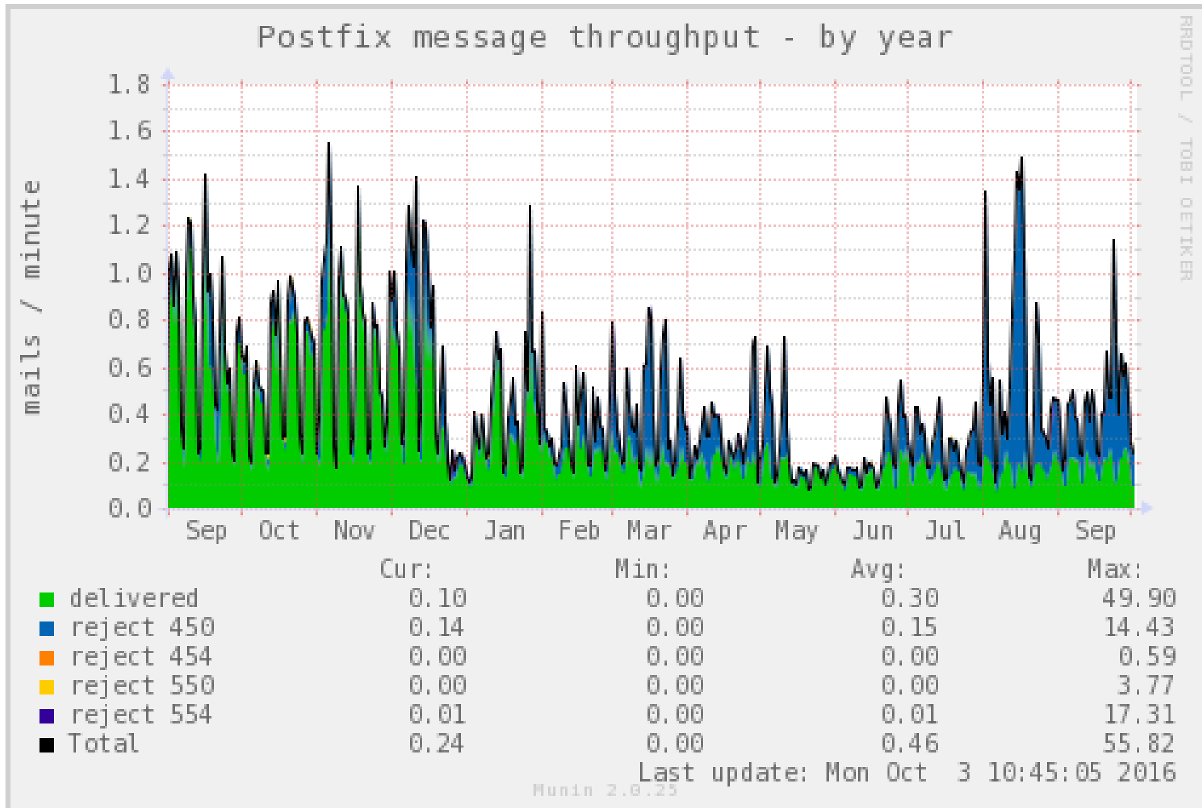
- 1 - 2: Configuration of Node foo.example.com.
- 4 - 5: Configuration of Node bar.example.com.
- 7 - 8: Configuration of Node bar.example.com.
- 10: Define the name of the virtual node. The name of a node group can optionally be added here, e.g. "[Virtual; aggregates.example.com]".
- 11: Make sure that `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.

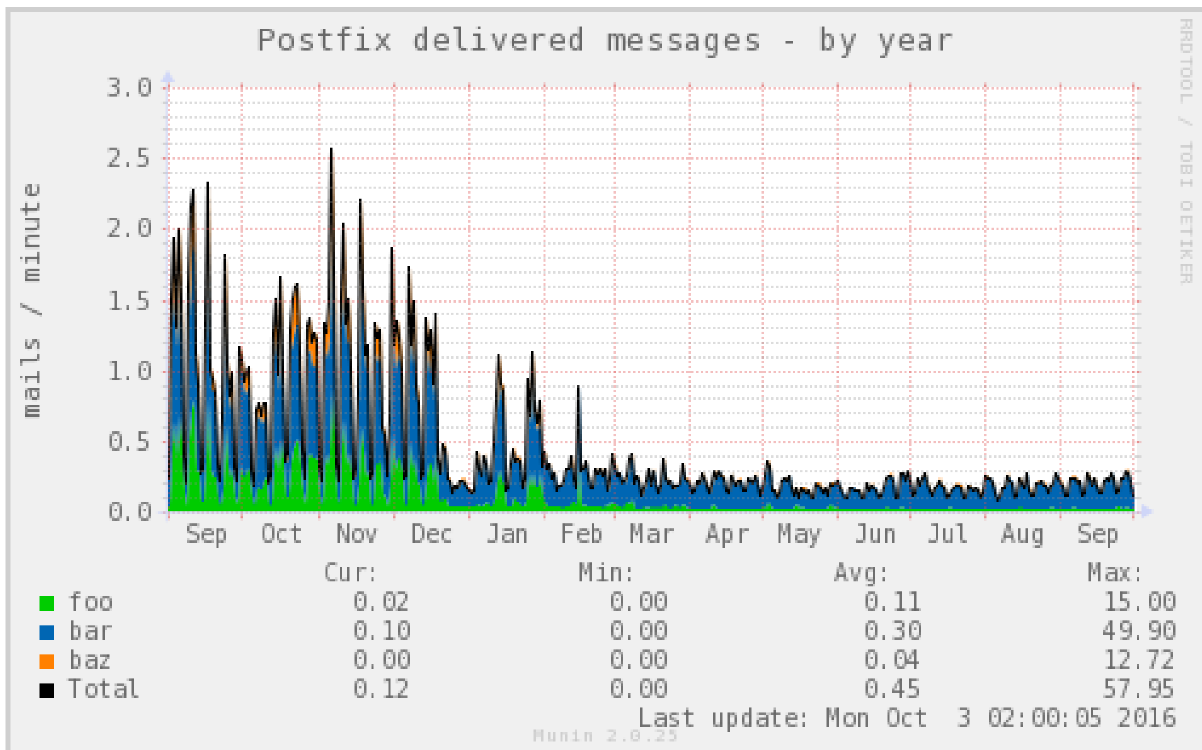
- 17: No fetch from node should be done for this virtual plugin (which is named implicitly herewith to "total_mailstats").
- 18 - 24: Define the virtual plugin's config in the same way as set for its *ancestor* plugin `postfix_mailstats` (see output in section *Plugin involved* above). We set a different 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 arithmetic operations 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)

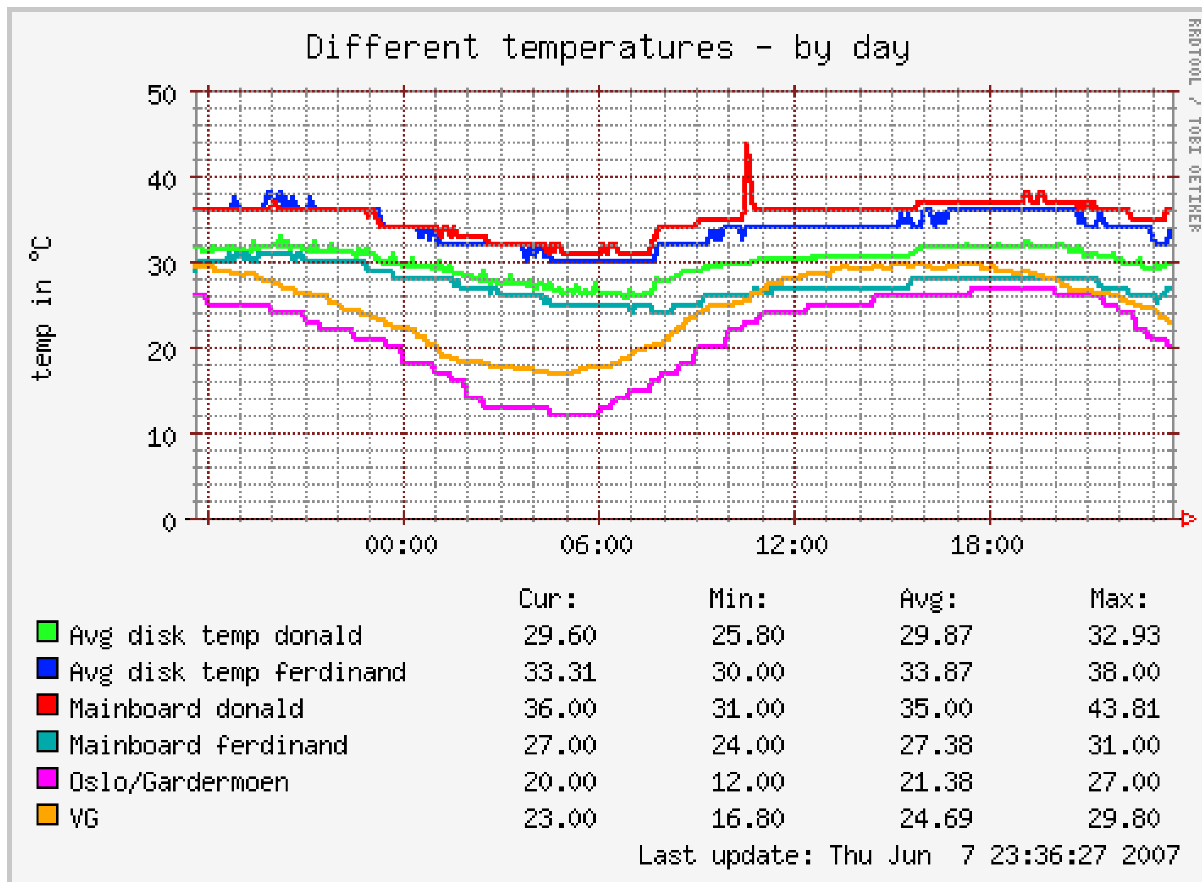
Line 06 is only a placeholder, which will be populated later by using *sum* and *cdef*.

```
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.ENGM \
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

Virtual plugin to graph distribution by percentages

Why?

In the FAQ it's explained how to create a percentage graph with two data sources by using a somewhat heavy *cdef* trick. Graphing more than two data sources using that method would

1. require CDEF incantations from somewhere beyond hell
2. require Munin to perform the same calculations in CDEF over and over again

Of course, an alternative is to create a wildcard version of the plugin; one that gives the numbers and one that gives the ratio(s). This will, however, dump the load on the *munin-node*.

Creating graphs showing distribution by percentage is often required, while lots of the Munin plugins deal with occurrences and hits. For example, the numbers of hits and misses in a reverse proxy are indeed interesting to see the throughput, while the hits/misses ratio is interesting to see how efficient the caching is.

(Yes, I realise this example was a bad one as it has only two data sources. Bear with me ;-)

How?

By combining loaning data from other data sources, and massaging them with CDEF, Munin is able to create another *view* of existing graphs not currently graphing in percentage shares by itself, showing the data distributed by percentage.

The *munin.conf* extract below is based on the `fw_conntrack` plugin, which has as much as 5 relevant data sources. The sections are commented below through line numbers.

Sample munin.conf extract

```
01 [foo.example.com]
02
03         [...]
04
05         conntrack_percent.update no
06         conntrack_percent.graph_category network
07         conntrack_percent.graph_args --base 1000 -l 0 -u 100 -r
08         conntrack_percent.graph_scale no
09         conntrack_percent.graph_title Connections through firewall, by␣
↪percentage
10         conntrack_percent.graph_vlabel Connections (percentage)
11
12        conntrack_percent.graph_order \
13                established=fw_conntrack.established \
14                fin_wait=fw_conntrack.fin_wait \
15                time_wait=fw_conntrack.time_wait \
16                syn_sent=fw_conntrack.syn_sent \
17                udp=fw_conntrack.udp \
18                total=fw_conntrack.nated \
19                in_established=fw_conntrack.nated \
20                in_fin_wait=fw_conntrack.nated \
21                in_time_wait=fw_conntrack.nated \
22                in_syn_sent=fw_conntrack.nated \
23                in_udp=fw_conntrack.nated
24
25        conntrack_percent.established.graph no
26        conntrack_percent.fin_wait.graph no
27        conntrack_percent.time_wait.graph no
28        conntrack_percent.syn_sent.graph no
29        conntrack_percent.udp.graph no
30
31        conntrack_percent.total.graph no
32        conntrack_percent.total.cdef established,fin_wait,time_wait,syn_sent,udp,
↪0.00001,+,+,+,+,+
33
34        conntrack_percent.in_established.cdef established,total,/,100,*
35        conntrack_percent.in_established.label Established
36        conntrack_percent.in_established.draw AREA
37
38        conntrack_percent.in_fin_wait.cdef fin_wait,total,/,100,*
39        conntrack_percent.in_fin_wait.label FIN_WAIT
40        conntrack_percent.in_fin_wait.draw STACK
41
42        conntrack_percent.in_time_wait.cdef time_wait,total,/,100,*
43        conntrack_percent.in_time_wait.label TIME_WAIT
44        conntrack_percent.in_time_wait.draw STACK
45
46        conntrack_percent.in_syn_sent.cdef syn_sent,total,/,100,*
47        conntrack_percent.in_syn_sent.label SYN_SENT
```

(continues on next page)

(continued from previous page)

```
48        conntrack_percent.in_syn_sent.draw STACK
49
50        conntrack_percent.in_udp.cdef udp,total,/,100,*
51        conntrack_percent.in_udp.label UDP connections
52        conntrack_percent.in_udp.draw STACK
```

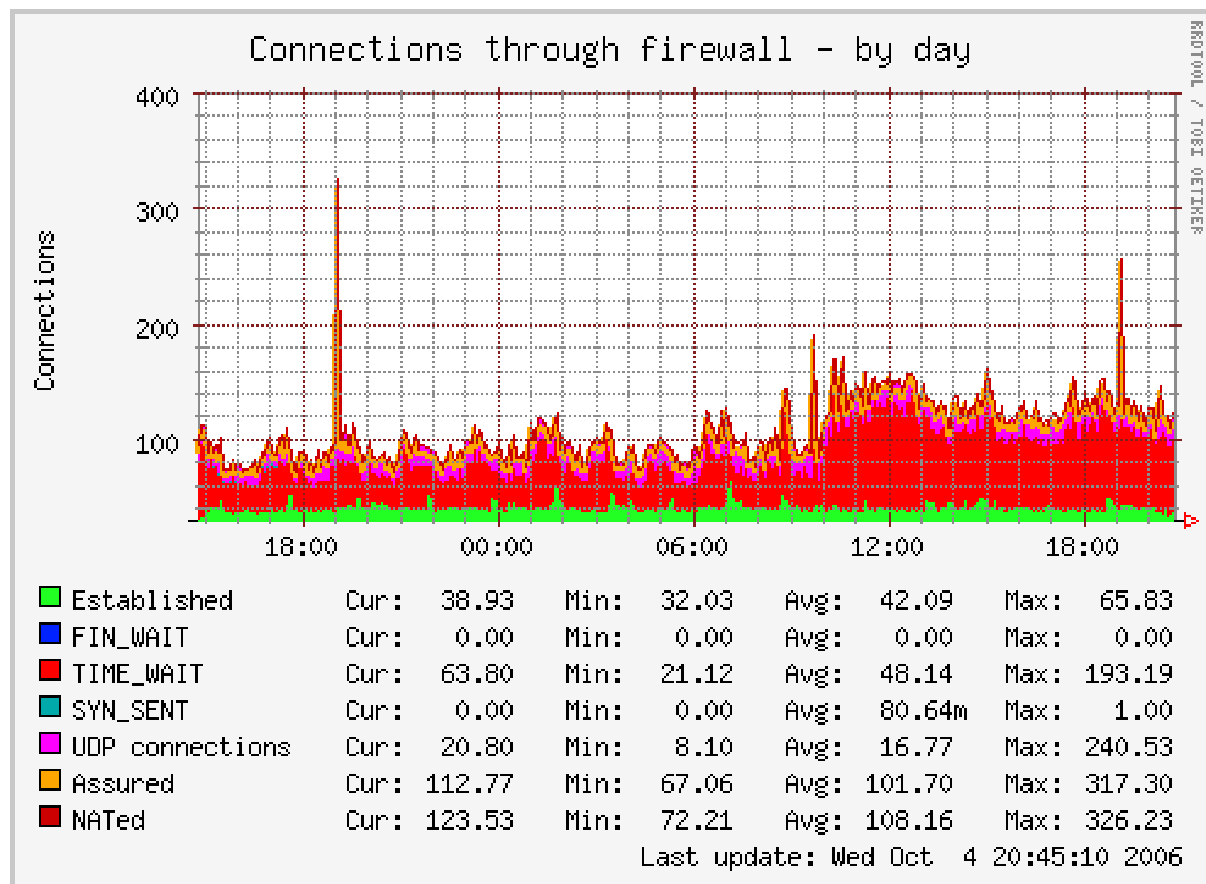
The config explained

- 01 - A standard definition for a node.
- 03 - Any other (required) information for defining a node, like `address`, has been omitted for readability.
- 05 - This virtual plugin should not be run by *munin-update*.
- 07 - Set min and max values for the graphs, and also include `-r` which tells RRD that the graph shall not exceed 100% even if the summarized values (because of rounding errors) do.
- 08 - No auto-scaling.
- 09-10 - The graph title and its vertical title.
- 12-23 - Use *graph_order* to *loan* data from other graphs, and to define not yet existing field names. Note that the first seven field names (11-17) loan data from corresponding graphs to be used later, while the rest (18-25) just need some dummy data for the definitions.
- 25-29 - Make sure Munin won't try to graph the base data.
- 31 - Munin shouldn't try to do anything funny with the total value either, as this value is only used internally when creating the graph.
- 32 - Here, the fieldname 'total' is defined as the sum of the 5 original data sources (`in_*`). We add the value 0.00001 to the value if all the original data sources should yield the value '0', in which case Munin will crash and burn.
- 34 - Define the value for the first field name, derived as a percentage share of the total sum.
- 35 - The field name needs a label. This should correspond to the original graph (in this case, fw_conntrack).
- 36 - Finally for this field name, define it as *draw type* AREA.
- 38-40 - Like the above, but use STACK instead. Repeat until all 5 data sources are covered.

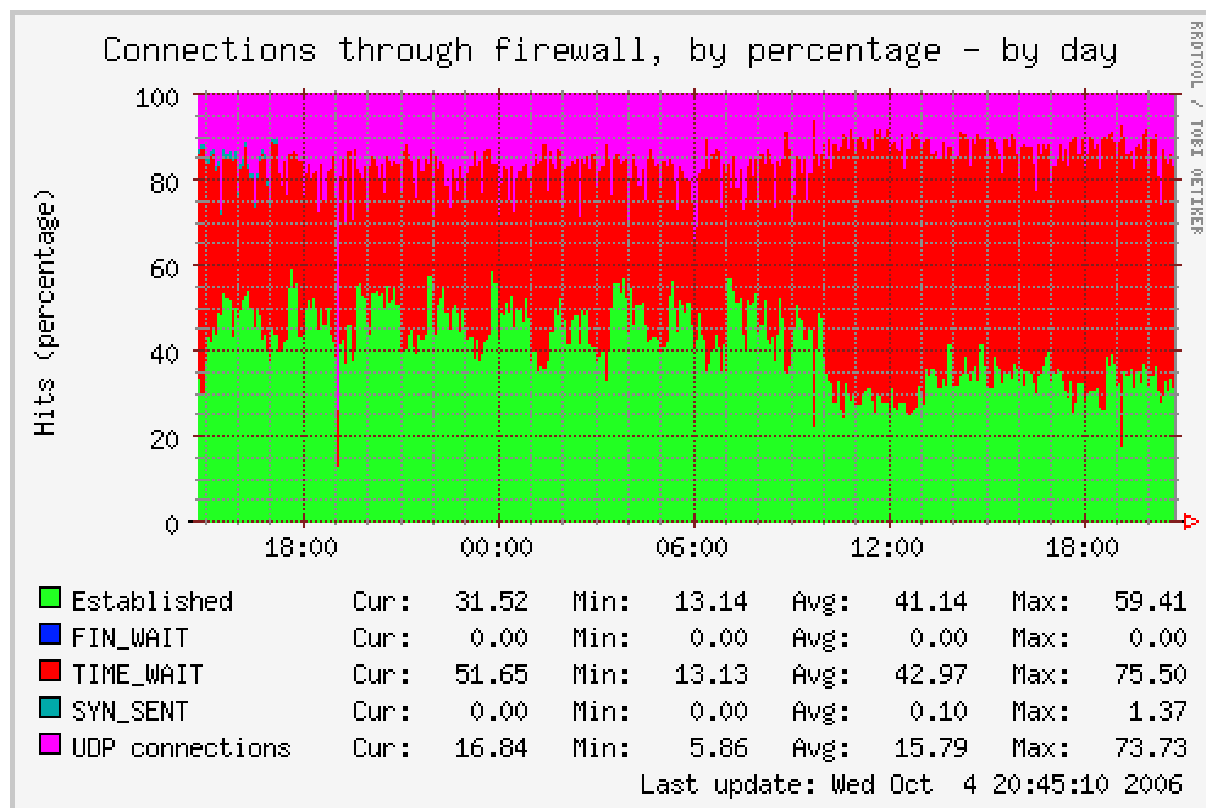
When the configuration is complete, Munin will be able to create percentage graphs from your already existing data, i.e. the new graph(s) should appear immediately.

Sample graphs

The plugin's own graph:



The resulting graph of the above munin.conf section, loaning data from the plugin and manipulating them:



Note that the data sources `assured` and `nated` differ from the other data sources the `fw_conntrack` plugin monitors. Consequently they are excluded from the percentage graph.

Further reading

- Loaning data from other graphs
- *Graph aggregation by 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 tousands 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.

Another option you might consider, is using munin-async. It requires modifications on all nodes, so it might not be an option, but I felt compeled 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.

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-httpd systemd service `/etc/systemd/system/munin-httpd.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

9.4 Other reference material

9.4.1 Directories

dbdir

This directory (usually `/var/lib/munin`) is used to store the Munin master database.

It contains subdirectories for the RRD files per group of hosts as well as files to store variable states that the munin master will need.

The RRD files are named in the following way: `<dbdir>/<group>/<nodename>-<servicename>-<fieldname>-[acdg].rrd`

Example:

```
/var/lib/munin/SomeGroup/foo.example.com-cpu-irq-d.rrd
                  --------- --------------- --- --- -
                      |            |          |   |  `-- Data type (a = absolute, c =␣
↪counter, d = derive, g = gauge)
                      |            |          |   `----- Field name / data source: 'irq
↪'
                      |            |          `--------- Plugin name: 'cpu'
                      |            `-------------------- Node name: 'foo.example.com'
                      `--------------------------------- Group name: 'SomeGroup'
```

plugindir

This directory (usually `/usr/share/munin/plugins`) contains all the plugins **available** to run on the node.

servicedir

This directory (usually `/etc/munin/plugins`) contains symlinks to all the plugins that **are selected** to run on the node. These will be shown when we are connected to *munin node* and say `list`.

pluginconfdir

This directory (usually `/etc/munin/plugin-conf.d`) contains plugin configuration.

rundir

This directory (usually `/var/run/munin`) contains files needed to track the munin run state. PID files, lock files, and possibly sockets.

logdir

Attention! On a host where Munin master resides there may be two of them! One (usually `/var/log/munin`) contains the log files for Munin master related applications and another (usually `/var/log/munin-node`) contains the logfiles of *munin node*.

9.4.2 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 yes|no

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 None (If not set, the order of the graphs follows the order in which the data sources are read; i.e. the order that the plugin itself provides.)

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 yes|no

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 [a-zA-Z0-9-.]

Type required

Description Sets the title of the graph

See also

Default The plugin's file name

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 [a-zA-Z0-9-.]

Type optional

Description Fully qualified host name (FQDN). Override the host name for which the plugin is run. Should normally **not** be set in the plugin. It is meant to be used when the munin-node acts as proxy to monitor remote hosts e.g. per SNMP plugins. In these cases you have to add an own entry for the remote host in the Munin master configuration to pick up these additional host names.

See also *Using SNMP plugins*

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 fieldname.stack? and *Graph aggregation 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}.unknown_limit**

Value positive integer

Type optional

Description Defines the number of *unknown* values to be received successively, before the state of the dataset changes from *ok* to *unknown*. Use a higher value, if you want to tolerate a certain number of non-computable values, before an alarm should be raised. This attribute is available since Munin 3.0.

See also *Let Munin croak alarm*

Default 3

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
```

9.4.3 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, PosgreSQL, 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 lastest 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

9.4.4 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 usablity. 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 Schnepp 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*

CHAPTER 10

Other docs

10.1 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.

10.2 Developer Talk

- Proposed extensions

10.2.1 Specifications

- Munin Relay : a multiplexing relay (not yet implemented)
- REST API (RFC)
- Stream plugin (RFC)

10.2.2 Ideas

- Some thoughts about Tags
- Some thoughts about basic module API for plugins written in Perl, Python and Ruby

10.2.3 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 deducts 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 11

Indices and Tables

- genindex

Index

Symbols

–always-send <severity list>
munin-limits command line option, 122
–base <value>
command line option, 143
–cleanup
munin-async command line option, 114
–cleanupandexit
munin-async command line option, 114
–config <configfile>
munin-node command line option, 126
munin-run command line option, 127
–config <file>
command line option, 123
munin-cron command line option, 118
munin-graph command line option, 119
munin-html command line option, 121
munin-limits command line option, 122
–config_file <file>
munin-update command line option, 128
–contact <contact>
munin-limits command line option, 122
–cron
munin-graph command line option, 119
–day
munin-graph command line option, 120
–debug
command line option, 123
munin-graph command line option, 119
munin-html command line option, 121
munin-limits command line option, 122
munin-node command line option, 126
munin-run command line option, 127
munin-update command line option, 128
–exitnoterror
command line option, 124
–families <family,...>
command line option, 124
–fix-permissions | -f
munin-check command line option, 116
–force
munin-graph command line option, 119
munin-limits command line option, 122
–force-run-as-root
munin-limits command line option, 122
–fork
munin-asyncd command line option, 115
munin-graph command line option, 119
munin-update command line option, 128
–help
command line option, 123
munin-graph command line option, 119
munin-html command line option, 121
munin-limits command line option, 122
munin-node command line option, 126
munin-run command line option, 127
munin-update command line option, 129
–help | -h
munin-async command line option, 115
munin-asyncd command line option, 115
munin-check command line option, 116
–host <host>
munin-cron command line option, 118
munin-graph command line option, 119
munin-html command line option, 121
munin-limits command line option, 122
munin-update command line option, 128
–host <hostname:port>
munin-asyncd command line option, 115
–hostname <hostname>
munin-async command line option, 114
–interval <seconds>
munin-asyncd command line option, 115
–lazy
munin-graph command line option, 119
–libdir <dir>
command line option, 123
–list-images
munin-graph command line option, 120
–log-file | -l
munin-graph command line option, 120
–logarithmic
command line option, 143
–lower-limit <value>
command line option, 143
–lower_limit <lim>
munin-graph command line option, 120

–month
munin-graph command line option, 120
–n <processes>
munin-graph command line option, 119
–newer <version>
command line option, 124
–nocleanup
munin-asyncd command line option, 115
–nofork
munin-html command line option, 121
–only-fqn <FQN>
munin-graph command line option, 119
–output-file | -o
munin-graph command line option, 120
–paranoia
munin-node command line option, 126
munin-run command line option, 127
–pidebug
command line option, 123
munin-node command line option, 126
munin-run command line option, 128
–pinpoint <start,stop>
munin-graph command line option, 120
–remove-also
command line option, 124
–retain <count>
munin-asyncd command line option, 115
–rigid
command line option, 144
–sconfdir <dir>
command line option, 123
munin-run command line option, 127
–sconffile <file>
munin-run command line option, 127
–screen
munin-graph command line option, 119
munin-html command line option, 121
munin-limits command line option, 122
munin-update command line option, 128
–service <service>
munin-cron command line option, 118
munin-html command line option, 121
munin-limits command line option, 122
munin-update command line option, 128
–servicedir <dir>
command line option, 123
munin-run command line option, 127
–shell
command line option, 124
–size_x <pixels>
munin-graph command line option, 120
–size_y <pixels>
munin-graph command line option, 120
–snmp <host|cidr,...>
command line option, 124
–snmpauthpass <password>
command line option, 125
–snmpauthproto <protocol>
command line option, 125
–snmpcommunity <string>
command line option, 124
–snmpdomain <domain>
command line option, 124
–snmpport <port>
command line option, 124
–snmpprivpass <password>
command line option, 125
–snmpprivproto <protocol>
command line option, 125
–snmpusername <name>
command line option, 125
–snmpversion <ver>
command line option, 124
–spool | -s <spooldir>
munin-asyncd command line option, 115
–spooldir | -s <spooldir>
munin-async command line option, 114
–spoolfetch
munin-async command line option, 114
–suggest
command line option, 124
–sumweek
munin-graph command line option, 120
–sumyear
munin-graph command line option, 120
–timeout <seconds>
munin-update command line option, 128
–units-exponent <value>
command line option, 143
–upper-limit <value>
command line option, 143
–upper_limit <lim>
munin-graph command line option, 120
–vectorfetch
munin-async command line option, 114
–verbose | -v
munin-async command line option, 115
munin-asyncd command line option, 115
–version
command line option, 123
munin-graph command line option, 119
munin-html command line option, 121
munin-run command line option, 128
munin-update command line option, 129
–week
munin-graph command line option, 120
–year
munin-graph command line option, 120

A

address <value>
munin.conf command line option, 133
aggregate
plugin, 144, 148, 151
Aggregating munin plugins, 144, 148, 151
allow

command line option, 136
apache configuration
example, munin-cgi, 139
example, munin-cron, 139
example, munin-httpd, 140
auto
magic marker family, 63
autoconf
magic marker capability, 63

B

background
command line option, 136

C

cap
protocol command, 38
capabilities
magic marker, 63
capability
autoconf, magic marker, 63
dirtyconfig, 51
multigraph, 50
suggest, magic marker, 63
cdef, 153
field, 151
CGI_DEBUG, 117
cgitmpdir <path>
munin.conf command line option, 130
cgiurl_graph /munin-cgi/munin-cgi-graph
munin.conf command line option, 130
cidr_allow
command line option, 136
cidr_deny
command line option, 136
command
cap, protocol, 38
config, protocol, 38
fetch, protocol, 38
list, protocol, 38
nodes, protocol, 38
quit, protocol, 38
version, protocol, 38
command line option
–base <value>, 143
–config <file>, 123
–debug, 123
–exitnoterror, 124
–families <family,...>, 124
–help, 123
–libdir <dir>, 123
–logarithmic, 143
–lower-limit <value>, 143
–newer <version>, 124
–pidebug, 123
–remove-also, 124
–rigid, 144
–sconfdir <dir>, 123
–servicedir <dir>, 123
–shell, 124
–snmp <host|cidr,...>, 124
–snmpauthpass <password>, 125
–snmpauthproto <protocol>, 125
–snmpcommunity <string>, 124
–snmpdomain <domain>, 124
–snmpport <port>, 124
–snmpprivpass <password>, 125
–snmpprivproto <protocol>, 125
–snmpusername <name>, 125
–snmpversion <ver>, 124
–suggest, 124
–units-exponent <value>, 143
–upper-limit <value>, 143
–version, 123
allow, 136
background, 136
cidr_allow, 136
cidr_deny, 136
global_timeout, 136
group, 136
host, 136
host_name, 135
ignore_file, 135
log_file, 136
log_level, 136
paranoia, 135
pid_file, 136
port, 137
setsid, 136
timeout, 136
user, 136
config
protocol command, 38
configuration
example plugin, 54
plugin, 53
contact.your_contact_name.command <command>
munin.conf command line option, 131
contact.your_contact_name.max_messages <number>
munin.conf command line option, 131
contact.your_contact_name.text <text>
munin.conf command line option, 131
contacts <no|your_contact_name1 your_contact_name2 ...>
munin.conf command line option, 132
contrib
magic marker family, 63
munin, 85
contribute
plugin, 103
contributing
munin documentation, 7
munin; documentation, 87
plugin; best current practices, 96
contribution
plugin, 107

critical <value>
munin.conf command line option, 134
custom_palette rrggbb rrggbb ...
munin.conf command line option, 130

D

data source
fqn, 113
loan, 144, 148, 151, 153
dbdir <path>
munin.conf command line option, 129
development
munin, 78
plugin, 85, 103, 107
plugin; concise, 99
dirtyconfig
capability, 51
protocol extension, 51
documentation
contributing munin, 7
plugin, 103
domain_order <group1> <group2> ..
munin.conf command line option, 132
ds
fqn, 113

E

environment
plugin, 45
environment variable
CGI_DEBUG, 116, 117
HTTP_CACHE_CONTROL, 116
HTTP_IF_MODIFIED_SINCE, 117
MUNIN_DEBUG, 126, 128
PATH_INFO, 116, 117
QUERY_STRING, 116
example
graph_args, 143
lighttpd configuration munin-httpd, 140
magic marker family, 63
munin-cgi apache configuration, 139
munin-cron apache configuration, 139
munin-cron nginx configuration, 141
munin-httpd apache configuration, 140
munin-httpd nginx configuration, 141
munin-node.conf, 126
munin.conf, 134
plugin configuration, 54
ssh transport munin.conf, 158
executing
plugin, 169
extension
dirtyconfig, protocol, 51
multigraph, protocol, 50

F

family
auto, magic marker, 63
contrib, magic marker, 63
example, magic marker, 63
magic marker, 63
manual, magic marker, 63
snmpauto, magic marker, 63
test, magic marker, 63
fetch
protocol command, 38
field
cdef, 151
sum, 151
fields
plugin, 161
fork <yes|no>
munin.conf command line option, 130
fqn, 111
data source, 113
ds, 113
group, 112
node, 112
plugin, 112
service, 112
fully qualified name, 111

G

gallery
plugin, 103
global_timeout
command line option, 136
graph
percentage distribution, 153
stacked, 148
graph_args, 143
example, 143
graph_category, 170
graph_data_size
master configuration;, rrd, 74
graph_data_size <normal|huge|custom>
munin.conf command line option, 131
graph_height <value>
munin.conf command line option, 133
graph_period <second>
munin.conf command line option, 130
graph_strategy <cgi|cron>
munin.conf command line option, 131
graph_width <value>
munin.conf command line option, 133
group
command line option, 136
fqn, 112

H

host
command line option, 136
host_name
command line option, 135
html_dynamic_images 1
munin.conf command line option, 130

html_strategy <cgi|cron>
munin.conf command line option, 131
htmldir <path>
munin.conf command line option, 129

I

ignore_file
command line option, 135
ignore_unknown <yes|no>
munin.conf command line option, 133
includedir <path>
munin.conf command line option, 130
installing
plugin, 53

L

lighttpd configuration
munin-httpd, example, 140
list
protocol command, 38
loan
data source, 144, 148, 151, 153
local_address <address>
munin.conf command line option, 130
log_file
command line option, 136
log_level
command line option, 136
logdir <path>
munin.conf command line option, 129

M

magic marker
capabilities, 63
capability autoconf, 63
capability suggest, 63
family, 63
family auto, 63
family contrib, 63
family example, 63
family manual, 63
family snmpauto, 63
family test, 63
plugin, 63
man pages, 113
manual
magic marker family, 63
master configuration;
rrd graph_data_size, 74
max_graph_jobs 6
munin.conf command line option, 130
max_processes 16
munin.conf command line option, 131
max_size_x 4000
munin.conf command line option, 130
max_size_y 4000
munin.conf command line option, 131
multigraph
capability, 50
protocol extension, 50
munin
contrib, 85
development, 78
documentation, contributing, 7
project, 78
Munin Exchange, 176
munin-async command line option
–cleanup, 114
–cleanupandexit, 114
–help | -h, 115
–hostname <hostname>, 114
–spooldir | -s <spooldir>, 114
–spoolfetch, 114
–vectorfetch, 114
–verbose | -v, 115
munin-asyncd command line option
–fork, 115
–help | -h, 115
–host <hostname:port>, 115
–interval <seconds>, 115
–nocleanup, 115
–retain <count>, 115
–spool | -s <spooldir>, 115
–verbose | -v, 115
munin-cgi
apache configuration example, 139
munin-check command line option
–fix-permissions | -f, 116
–help | -h, 116
munin-cron
apache configuration example, 139
nginx configuration example, 141
munin-cron command line option
–config <file>, 118
–host <host>, 118
–service <service>, 118
munin-graph command line option
–config <file>, 119
–cron, 119
–day, 120
–debug, 119
–force, 119
–fork, 119
–help, 119
–host <host>, 119
–lazy, 119
–list-images, 120
–log-file | -l, 120
–lower_limit <lim>, 120
–month, 120
–n <processes>, 119
–only-fqn <FQN>, 119
–output-file | -o, 120
–pinpoint <start,stop>, 120
–screen, 119
–size_x <pixels>, 120

–size_y <pixels>, 120
–sumweek, 120
–sumyear, 120
–upper_limit <lim>, 120
–version, 119
–week, 120
–year, 120

munin-html command line option
–config <file>, 121
–debug, 121
–help, 121
–host <host>, 121
–nofork, 121
–screen, 121
–service <service>, 121
–version, 121

munin-httpd
apache configuration example, 140
example lighttpd configuration, 140
nginx configuration, example, 141

munin-limits command line option
–always-send <severity list>, 122
–config <file>, 122
–contact <contact>, 122
–debug, 122
–force, 122
–force-run-as-root, 122
–help, 122
–host <host>, 122
–screen, 122
–service <service>, 122

munin-node command line option
–config <configfile>, 126
–debug, 126
–help, 126
–paranoia, 126
–pidebug, 126

munin-node.conf
example, 126

munin-run command line option
–config <configfile>, 127
–debug, 127
–help, 127
–paranoia, 127
–pidebug, 128
–sconfdir <dir>, 127
–sconffile <file>, 127
–servicedir <dir>, 127
–version, 128

munin-update command line option
–config_file <file>, 128
–debug, 128
–fork, 128
–help, 129
–host <host>, 128
–screen, 128
–service <service>, 128
–timeout <seconds>, 128
–version, 129

munin.conf
example, 134
example, ssh transport, 158

munin.conf command line option
address <value>, 133
cgitmpdir <path>, 130
cgiurl_graph /munin-cgi/munin-cgi-graph, 130
contact.your_contact_name.command <command>, 131
contact.your_contact_name.max_messages <number>, 131
contact.your_contact_name.text <text>, 131
contacts <no|your_contact_name1 your_contact_name2 ...>, 132
critical <value>, 134
custom_palette rrggbb rrggbb ..., 130
dbdir <path>, 129
domain_order <group1> <group2> .., 132
fork <yes|no>, 130
graph_data_size <normal|huge|custom>, 131
graph_height <value>, 133
graph_period <second>, 130
graph_strategy <cgi|cron>, 131
graph_width <value>, 133
html_dynamic_images 1, 130
html_strategy <cgi|cron>, 131
htmldir <path>, 129
ignore_unknown <yes|no>, 133
includedir <path>, 130
local_address <address>, 130
logdir <path>, 129
max_graph_jobs 6, 130
max_processes 16, 131
max_size_x 4000, 130
max_size_y 4000, 131
munin_cgi_graph_jobs 6, 130
node_order <node1> <node2> .., 132
notify_alias <node name>, 133
palette <default|old>, 130
port <port number>, 133
rrdcached_socket /var/run/rrdcached.sock, 131
rundir <path>, 129
ssh_command <command>, 131
ssh_options <options>, 131
staticdir <path>, 129
timeout <seconds>, 130
tmpldir <path>, 129
update <yes|no>, 133
use_node_name <yes|no>, 133
warning <value>, 133

munin; documentation
contributing, 87

munin_cgi_graph_jobs 6
munin.conf command line option, 130

MUNIN_DEBUG, 126, 128

N

nginx configuration
  example munin-httpd, 141
  example, munin-cron, 141
node
  fqn, 112
  virtual, 129, 144, 148, 151
node_order <node1> <node2> ..
  munin.conf command line option, 132
nodes
  protocol command, 38
notify_alias <node name>
  munin.conf command line option, 133

P

palette <default|old>
  munin.conf command line option, 130
paranoia
  command line option, 135
PATH_INFO, 116
percentage distribution
  graph, 153
pid_file
  command line option, 136
plugin
  aggregate, 144, 148, 151
  configuration, 53
  configuration, example, 54
  contribute, 103
  contribution, 107
  development, 85, 103, 107
  documentation, 103
  environment, 45
  executing, 169
  fields, 161
  fqn, 112
  gallery, 103
  installing, 53
  magic marker, 63
  testing, 55
  virtual, 144, 148, 151, 153
plugin; best current practices
  contributing, 96
plugin; concise
  development, 99
port
  command line option, 137
port <port number>
  munin.conf command line option, 133
project
  munin, 78
protocol
  command cap, 38
  command config, 38
  command fetch, 38
  command list, 38
  command nodes, 38
  command quit, 38
  command version, 38
  extension dirtyconfig, 51
  extension multigraph, 50

Q

QUERY_STRING, 116
quit
  protocol command, 38

R

RRD
  spikes;, 77
rrd
  graph_data_size master configuration;, 74
rrdcached_socket /var/run/rrdcached.sock
  munin.conf command line option, 131
rundir <path>
  munin.conf command line option, 129

S

service
  fqn, 112
setsid
  command line option, 136
snmpauto
  magic marker family, 63
spikes;
  RRD, 77
ssh transport
  munin.conf example, 158
ssh_command <command>
  munin.conf command line option, 131
ssh_options <options>
  munin.conf command line option, 131
stacked
  graph, 148
staticdir <path>
  munin.conf command line option, 129
suggest
  magic marker capability, 63
sum
  field, 151

T

test
  magic marker family, 63
testing
  plugin, 55
timeout
  command line option, 136
timeout <seconds>
  munin.conf command line option, 130
tmpldir <path>
  munin.conf command line option, 129
tuple: graph
  category, 170
tuple: munin-node.conf
  example, 137

tuple: plugin
    graph_category, 170

U

update <yes|no>
    munin.conf command line option, 133
use_node_name <yes|no>
    munin.conf command line option, 133
user
    command line option, 136

V

version
    protocol command, 38
vetted plugin, 107
virtual
    node, 129, 144, 148, 151
    plugin, 144, 148, 151, 153

W

warning <value>
    munin.conf command line option, 133