Nagios®

Nagios® Version 1.0 Documentation

Copyright © 1999-2002 Ethan Galstad
Email: nagios@nagios.org

Last Updated: 08-28-2002

[Table of Contents]

Nagios®

Version 1.0 Documentation

Table of Contents

About Nagios™

What Is This?

Nagios™ is a system and network monitoring application. It watches hosts and services that you specify, alerting you when things go bad and when they get better.

Nagios™ was originally designed to run under Linux, although it should work under most other unices as well. For more information on what operating systems Nagios will and will not run under, see the OS ports page at http://www.nagios.org/ports.shtml.

Some of the many features of Nagios™ include:

- Monitoring of network services (SMTP, POP3, HTTP, NNTP, PING, etc.)
- Monitoring of host resources (processor load, disk usage, etc.)
- Simple plugin design that allows users to easily develop their own service checks
- Parallelized service checks
- Ability to define network host hierarchy using "parent" hosts, allowing detection of and distinction between hosts that are down and those that are unreachable
- Contact notifications when service or host problems occur and get resolved (via email, pager, or user-defined method)
- Ability to define event handlers to be run during service or host events for proactive problem resolution
- Automatic log file rotation
- Support for implementing redundant monitoring hosts
- Optional web interface for viewing current network status, notification and problem history, log file, etc.

System Requirements

The only requirement of running Nagios is a machine running Linux (or UNIX variant) and a C compiler. You will probably also want to have TCP/IP configured, as most service checks will be performed over the network.

You are *not required* to use the CGIs included with Nagios. However, if you do decide to use them, you will need to have the following software installed...

1. A web server (preferrably Apache)
2. Thomas Boutell's gd library version 1.6.3 or higher (required by the statusmap and trends CGIs)

Licensing

Nagios™ is licensed under the terms of the GNU General Public License Version 2 as published by the Free Software Foundation. This gives you legal permission to copy, distribute and/or modify Nagios under certain conditions. Read the 'LICENSE' file in the Nagios distribution or read the online version of the license for more details.

Nagios^{TM} is provided AS IS with NO WARRANTY OF ANY KIND, INCLUDING THE WARRANTY OF DESIGN, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE.

Acknowledgements

Several people have contributed to Nagios by either reporting bugs, suggesting improvements, writing plugins, etc. A list of some of the many contributors to the development of Nagios can be found at http://www.nagios.org.

Downloading The Latest Version

You can check for new versions of Nagios at http://www.nagios.org.

Other Monitoring Utilities

In case you weren't aware, there are other network monitoring utilities available besides Nagios. I think Nagios is a pretty good contender, but I'm obviously biased. Here are links to a few other free monitoring utilities. There are many others that are not listed here - search the net (or Freshmeat) and you'll find them.

- Angel Network Monitor
- Autostatus
- Big Brother
- HiWAyS
- MARS
- Mon
- Netup (French)
- NocMonitor
- NodeWatch
- Penemo
- PIKT
- RITW
- Scotty
- Spong
- Sysmon

What's New in Version 1.0

Important: Make sure you read through the documentation (especially the FAQs) before sending a question to the mailing lists.

Many of the changes described below are the direct result of this project being renamed from *NetSaint*. Transitioning from NetSaint to Nagios will undoubtedly take some time, but it'll be well worth it.

Change Log

The change log for Nagios can be found online at http://www.nagios.org or in the **Changelog** file in the root directory of the source code distribution.

Changes

1. **User/Group Change**. The default user/group that Nagios runs under is now *nagios/nagios*.
2. **Directory Change**. The default directory that Nagios gets installed in is now */usr/local/nagios*.
3. **URL Changes**. The base URLs for accessing the HTML files and CGIs through the web interface are now */nagios/* and */nagios/cgi-bin/*, respectively.
4. **Config File Changes**. The main config file is now *nagios.cfg* and the CGI config file is now *cgi.cfg*.
5. **Process Check Command Changes**. The old process_check_command variable in the CGI config file has been renamed to nagios_check_command. Also, if you do not specify a check command, the CGIs will assume the Nagios process is running properly.
6. **Archive Changes**. Archived log files from Netsaint must be renamed from "netsaint-*date*.log" format to "nagios-*date*.log" format if you want to make them available to the Nagios CGIs. You can rename all your archived log files with the following command (assuming you've already moved them to their new directory location): `rename netsaint nagios netsaint*.log`
7. **Retention File Format Change**. The format of the retention file (or database, if that's what you were using) has changed to support more variables. No conversion utility is yet available, which means you'll either have to find a way to manually convert your retention data, or lose it when you make the changeover.
8. **Database Schema Changes**. The database schema for status, retention, comment, and extended information data has changed. If you were using database support previously, you'll either have to recreate the databases using the sample scripts provided in the *contrib/database* directory just alter your existing tables (an exercise which will be left to you). Also note that the default database name is now *nagios*.

New Features

1. **Template-Based Object Config File**. This is probably the biggest feature which has been added. Use of the template-based object config file format is optional, but highly recommended. Note that the older config file format is still supported if you really want it. The template-based config file is much easier to read, modify, and expand upon compared to the older format. It also allows you to specify host- and service-specific values for things like flap detection thresholds, flap detection and performance data processing options, etc. If you're interested in trying out the new template-based config file format, check out the **convertcfg** utility in the *contrib/* directory of the distribution - it can be used to quickly convert your old config files to the template-based format. More information on the template-based object config file can be found here.

2. **Template-Based Extended Info Config File**. This is similar to the template-based object configuration file format mentioned above. You can now store extended host and service information in a template-based config file. More information on doing this can be found here. If you wish, you can still use the older style of defining extended information directives in the CGI configuration file as described here.

3. **Host Dependencies**. You can now define optional host dependencies which will prevent notifications from being sent out for a host if one or more criteria fail. In the past there have been *implicit* dependencies between hosts that are related through "parenting", but this now allows you to create explicit dependencies between unrelated hosts. More information on dependencies can be found here.

4. **Host Escalations**. You can now define optional notification escalations for specific hosts. In the past you were only able to define escalations for entire *hostgroups*. While this was closely matched to non-escalated notification logic, it didn't provide much flexibility. Note that hostgroup escalations are still supported and can be used in conjunction with host escalations. More information on notification escalations can be found here.

5. **Freshness Checking**. Nagios now internally handles the concept of "freshness checking" of service check results. If freshness checking is enabled for a particular service, Nagios will force an active check of that service if the results from the last check are "stale" or "too old" (as determined by a threshold you specify). This makes implementing distributed monitoring servers much simpler, as you don't need an additional addon to make sure service results are "fresh". More information on how freshness checking works can be found here.

6. **Scheduled Downtime**. Scheduled downtime for hosts and service is now retained across program starts. Additionally, you can now distinguish between "fixed" and "flexible" downtime. Fixed downtime starts and stops at absolute times, while flexible downtime starts when a host or service first goes into a problem state. More information on scheduled downtime can be found here.

7. **State Stalking**. You can now enable "stalking" for different states on a per-host or per-service basis. Stalking provides you with more information about problems when you're analyzing archived log data. More information on state stalking can be found here.

8. **File-Based Performance Data Processing**. Nagios can now be compiled to dump performance data directly to a file in a format you define. This method is must faster and requires far less overhead that the default method of processing performance data. More information on the file-based option can be found here. General information about performance data can be found here.

9. **New Histogram CGI**. A new histogram CGI has been added. This CGI allows you to see better analyze

when host and service alerts occur over various periods of time.

10. **New Summary CGI**. A new summary CGI has been added. This CGI allows you to generated basic reports about host and service alerts over various periods of time. Reports can be created to show alert totals, top alert producers, most recent alerts, etc.

11. **Statusmap CGI Improvements**. Several new layout methods have been added to the statusmap CGI. You can also now specify a default layout method with the default_statusmap_layout directive.

12. **Availability CGI Improvements**. I have made several enhancements to the code in the availability CGI, including the ability to separate scheduled downtime from non-scheduled downtime.

13. **Configuration Directory**. You can now specify one or more directories that should be scanned for object configuration files by using the cfg_dir directive. You can use this in conjunction with (or instead of) the cfg_file directive.

14. **Custom CGI Headers/Footers**. You can now include optional headers and footers in the CGIs. This is most useful if you do custom Nagios installations for customers and want to include tag line, contact info, etc in each page. More information on doing this can be found here.

15. **Cleaning Of Dangerous Macro Output Characters**. Potentially dangerous characters can now be stripped from the $OUTPUT$ and $PERFDATA$ macros before they're used in notification commands, etc. by using the illegal_macro_output_chars directive. At a bare minimum, I highly recommend you strip out the characters shown in the example, or an attacker might be able to execute arbitrary commands as the nagios user!

Frequently Asked Questions (FAQs)

Online FAQ Database

A searchable FAQ database can now be found online at http://www.nagios.org/faqs.

Advice for Beginners

Congrats on choosing to try Nagios! Nagios is quite powerful and flexible, but unfortunately its not very friendly to newbies. Why? Because it takes a lot of work to get it installed and configured properly. That being said, if you stick with it and manage to get it up and running, you'll never want to be without it. :-) Here are some very important things to keep in mind for those of you who are first-time users of Nagios:

1. **Relax - its going to take some time.** Don't expect to be able to compile Nagios and start it up right off the bat. Its not that easy. In fact, its pretty difficult. If you don't want to spend time learning how things work and getting things running smoothly, don't bother using this software. Instead, pay someone to monitor your network for you or hire someone to install Nagios for you. :-)
2. **Read the documentation.** Nagios is difficult enough to configure when you've got a good grasp of what's going on, and nearly impossible if you don't. Do yourself a favor and read before blindly attempting to install and run Nagios. If you're the type who doesn't want to take the time to read the documentation, you'll probably find that others won't find the time to help you out when you have problems. RTFM.
3. **Use the sample config files.** Sample configuration files are provided with Nagios. Look at them, modify them for your particular setup and test them! The sample files are just that - samples. There's a very good chance that they won't work for you without modifications. Sample config files can be found in the *sample-config/* subdirectory of the Nagios distribution.
4. **Seek the help of others.** If you've read the documentation, reviewed the sample config files, and are still having problems, try sending a *descriptive* email message describing your problems to the *nagios-users* mailing list. Due to the amount of work that I have to do for this project, I am unable to answer most of the questions that get sent directly to me, so your best source of help is going to be the mailing list. If you've done some background reading and you provide a good problem description, odds are that someone will give you some pointers on getting things working properly.

Installing Nagios

Important: Installing and configuring Nagios is rather involved. You can't just compile the binaries, run the program and sit back. There's a lot to setup before you can actually start monitoring anything. Relax, take your time and read all the documentation - you're going to need it. Okay, let's get started...

Unpacking The Distribution

To unpack the Nagios distribution, type the following two commands at a shell prompt:

```
gunzip nagios-1.0.tar.gz
tar xf nagios-1.0.tar
```

If you downloaded the ZIP version of the distribution, type the following:

```
unzip nagios-1.0.zip
```

When you have finished executing these commands, you should find a **nagios-1.0** directory that has been created in your current directory. Inside that directory you will find all the files that compromise the core Nagios distribution.

Create Installation Directory

Create the base directory where you would like to install Nagios as follows...

```
mkdir /usr/local/nagios
```

Create User/Group

You're probably going to want to run Nagios under a normal user account, so add a new user (and group) to your system with the following commands (these will vary depending on what OS you're running):

```
adduser nagios
```

Run the Configure Script

Run the configure script to initialize variables and create a Makefile as follows...

```
./configure --prefix=prefix --with-cgiurl=cgiurl --with-htmurl=htmurl --with-nagios-user=someuser --with-nagios-grp=somegroup
```

- Replace *prefix* with the installation directory that you created in the step above (default is

/usr/local/nagios)

- Replace *cgiurl* with the actual url you will be using to access the CGIs (default is */nagios/cgi-bin*). Do NOT append a slash at the end of the url.
- Replace *htmurl* with the actual url you will be using to access the HTML for the main interface and documentation (default is */nagios/*)
- Replace *someuser* with the name of a user on your system that will be used for setting permissions on the installed files (default is *nagios*)
- Replace *somegroup* with the name of a group on your system that will be used for setting permissions on the installed files (default is *nagios*)

Compile Binaries

Compile Nagios and the CGIs with the following command:

make all

Installing The Binaries And HTML Files

Install the binaries and HTML files (documentation and main web page) with the following command:

make install

Installing An Init Script

If you wish, you can also install the sample init script to */etc/rc.d/init.d/nagios* with the following command:

make install-init

You may have to edit the init script to make sense with your particular OS and Nagios installation by editing paths, etc.

Directory Structure And File Locations

Change to the root of your Nagios installation directory with the following command...

cd /usr/local/nagios

You should see five different subdirectories. A brief description of what each directory contains is given in the table below.

Sub-Directory	Contents
bin/	Nagios core program
etc/	Main, resource, object, and CGI configuration files should be put here

sbin/	CGIs
share/	HTML files (for web interface and online documentation)
var/	Empty directory for the log file

Installing The Plugins

In order for Nagios to be of any use to you, you're going to have to download and install some plugins. Plugins are usually installed in the **libexec/** directory of your Nagios installation (i.e. */usr/local/nagios/libexec*). Plugins are scripts or binaries which perform all the service and host checks that constitute monitoring. You can grab the latest release of the plugins from the Nagios downloads page or directly from the SourceForge project page.

Setup The Web Interface

You're probably going to want to use the web interface, so you'll also have to read the instructions on setting up the web interface and configuring web authentication, etc. next.

Configuring Nagios

So now you have things compiled and installed, but you still haven't configured Nagios or defined objects (hosts, services, etc.) that should be monitored. Information on configuring Nagios and defining objects can be found here. There's a lot to configure, but don't let it discourage you - its well worth it.

Setting Up The Web Interface

Notes

In these instructions I will assume that you are running the Apache web server on your machine. If you are using some other web server, you'll have to make changes where appropriate. I am also assuming that you used the */usr/local/nagios* as the installation prefix.

Configure Script Alias For The CGIs

You'll need to create an alias for the CGIs as well. The default installation expects to find them accessible at **http://yourmachine/nagios/cgi-bin/**, although this can be changed using the **--with-cgiurl** option in the configure script. Anyway, add something like the following to your web server configuration file (i.e. **httpd.conf**) (changing it to match any directory differences on your system)...

```
ScriptAlias /nagios/cgi-bin/ /usr/local/nagios/sbin/
<Directory "/usr/local/nagios/sbin/">
    AllowOverride AuthConfig
    Options ExecCGI
    Order allow,deny
    Allow from all
</Directory>
```

Important! The Script-Alias line above must come before the Alias line below. Otherwise Apache will parse the lines differently.

Important! If you are installing Nagios on a multi-user system, you may want use CGIWrap to provide additional security between the CGIs and the external command file. If you decide to use CGIWrap, the ScriptAlias you'll end up using will most likely be different from that mentioned above. More information on doing this can be found here.

Configure Alias For The HTML Files

In order to make the HTML files accessible via the web server, you'll have to edit your Apache configuration file as follows...

Add the following to your web server configuration file (i.e. **httpd.conf**) as follows:

```
Alias /nagios/ /usr/local/nagios/share/
<Directory "/usr/local/nagios/share">
    Options None
    AllowOverride AuthConfig
    Order allow,deny
    Allow from all
</Directory>
```

This will allow you to use an URL like **http://yourmachine/nagios/** to view the HTML web interface and documentation. The alias should be the same value that you entered for the **--with-htmurl** argument to the configure script (default is */nagios/*).

Important! The Alias directive you just added for the HTML files must come *after* the ScriptAlias directive for the CGIs. If it doesn't, you'll get a 404 error when attempting to access the CGIs.

Restart The Web Server

Once you've finished editing the Apache configuration file, you'll need to restart the web server with a command like this...

```
/etc/rc.d/init.d/httpd restart
```

Verify Your Changes

Don't forget to check and see if the changes you made to Apache work. You should be able to point your web browser at **http://yourmachine/nagios** and get the web interface for Nagios. The CGIs may not display any information, but this will be remedied once you configure everything and start Nagios.

Configuring Web Authentication

Once you have configured the web interface properly, you'll need to enable web server authentication for accessing the CGIs and configure user authorization information. Details on doing this can be found here.

Configuring Nagios

Configuration Overview

There are several different configuration files that you're going to need to create or edit before you start monitoring anything. They are described below...

Main Configuration File

The main configuration file (usually */usr/local/nagios/etc/nagios.cfg*) contains a number of directives that affect how Nagios operates. This config file is read by both the Nagios process and the CGIs. This is the first configuration file you're going to want to create or edit.

Documentation for the main configuration file can be found here.

A sample main configuration file is generated automatically when you run the **configure** script before compiling the binaries. Look for it either in the distribution directory or the etc/ subdirectory of your installation. When you install the sample config files using the **make install-config** command, a sample main configuration file will be placed into your settings directory (usually /usr/local/nagios/etc). The default name of the main configuration file is **nagios.cfg**.

Resource File(s)

Resource files can be used to store user-defined macros. Resource files can also contain other information (like database connection settings), although this will depend on how you've compiled Nagios. The main point of having resource files is to use them to store sensitive configuration information and not make them available to the CGIs.

You can specify one or more optional resource files by using the resource_file directive in the main configuration file.

Object Configuration Files

Object configuration files (historically called "host" configuration files) are used to define hosts, services, hostgroups, contacts, contactgroups, commands, etc. This is where you define what things you want monitor and how you want to monitor them.

Documentation for the object configuration files can be found here.

CGI Configuration File

The CGI configuration file (usually */usr/local/nagios/etc/cgi.cfg*) contains a number of directives that affect the operation of the CGIs.

Documentation for the CGI configuration file can be found here.

A sample CGI configuration file is generated automatically when you run the **configure** script before compiling the binaries. When you install the sample config files using the **make install-config** command, the CGI configuration file will be placed in the same directory as the main and host config files (usually /usr/local/nagios/etc). The default name of the CGI configuration file is **cgi.cfg**.

Extedned Information Configuration Files

Extended information configuration files are used to define additional information for hosts and services that should be used by the CGI. This is where you define things like drawing coordinates, pretty icons, etc.

Documentation for the extended information configuration files can be found here.

Main Configuration File Options

Notes

When creating and/or editing configuration files, keep the following in mind:

1. Lines that start with a '#' character are taken to be comments and are not processed
2. Variables names must begin at the start of the line - no white space is allowed before the name
3. Variable names are case-sensitive

Sample Configuration

A sample main configuration file is created in the base directory of the Nagios distribution when you run the configure script. The default name of the main configuration file is **nagios.cfg** - its usually placed in the **etc/** subdirectory of you Nagios installation (i.e. */usr/local/nagios/etc/*).

Index

Performance data processing option

Orphaned service check option

Service freshness checking option
Service freshness checking option

Illegal object name characters
Illegal macro output characters

Administrator email address
Administrator pager

Log File

Format:	**log_file=<file_name>**
Example:	**log_file=/usr/local/nagios/var/nagios.log**

This variable specifies where Nagios should create its main log file. This should be the first variable that you define in your configuration file, as Nagios will try to write errors that it finds in the rest of your configuration data to this file. If you have log rotation enabled, this file will automatically be rotated every hour, day, week, or month.

Object Configuration File

Format:	**cfg_file=<file_name>**
Example:	**cfg_file=/usr/local/nagios/etc/hosts.cfg** **cfg_file=/usr/local/nagios/etc/services.cfg** **cfg_file=/usr/local/nagios/etc/commands.cfg**

This directive is used to specify an object configuration file that Nagios should use for monitoring. This file has traditionally been called the "host" config file, even though it may contain more than just host definitions. Object configuration files contain definitions for hosts, host groups, contacts, contact groups, services, commands, etc. You can seperate your configuration information into several files and specify multiple **cfg_file=** statements to have each of them processed.

Object Configuration Directory

Format:	**cfg_dir=<directory_name>**
Example:	**cfg_dir=/usr/local/nagios/etc/commands** **cfg_dir=/usr/local/nagios/etc/services** **cfg_dir=/usr/local/nagios/etc/hosts**

This directive is used to specify a directory which contains object configuration files that Nagios should use for monitoring. All files in the directory with a **.cfg** extension are processed as object config files. You can seperate your configuration files into different directories and specify multiple **cfg_dir=** statements to have all config files in each directory processed.

Resource File

Format: **resource_file=<file_name>**

Example: **resource_file=/usr/local/nagios/etc/resource.cfg**

This is used to specify an optional resource file that can contain $USERn$ macro definitions. $USERn$ macros are useful for storing usernames, passwords, and items commonly used in command definitions (like directory paths). The CGIs will *not* attempt to read resource files, so you can set restrictive permissions (600 or 660) on them to protect sensitive information. You can include multiple resource files by adding multiple resource_file statements to the main config file - Nagios will process them all. See the sample resource.cfg file in the base of the Nagios directory for an example of how to define $USERn$ macros.

Temp File

Format: **temp_file=<file_name>**

Example: **temp_file=/usr/local/nagios/var/nagios.tmp**

This is a temporary file that Nagios periodically creates to use when updating comment data, status data, etc. The file is deleted when it is no longer needed.

Status File

Format: **status_file=<file_name>**

Example: **status_file=/usr/local/nagios/var/status.log**

This is the file that Nagios uses to store the current status of all monitored services. The status of all hosts associated with the service you monitor are also recorded here. This file is used by the CGIs so that current monitoring status can be reported via a web interface. The CGIs must have read access to this file in order to function properly. This file is deleted every time Nagios stops and recreated when it starts.

Aggregated Status Updates Option

Format: **aggregate_status_updates=<0/1>**

Example: **aggregate_status_updates=1**

This option determines whether or not Nagios will aggregate updates of host, service, and program status data. If you do not enable this option, status data is updated every time a host or service checks occurs. This can result in high CPU loads and file I/O if you are monitoring a lot of services. If you want Nagios to only update status data (in the status file) every few seconds (as determined by the status_update_interval option), enable this option. If you want immediate updates, disable it. I would highly recommend using aggregated updates (even at short intervals) unless you have good reason not to. Values are as follows:

- 0 = Disable aggregated updates
- 1 = Enabled aggregated updates (default)

Aggregated Status Update Interval

Format: **status_update_interval=<seconds>**
Example: **status_update_interval=15**

This setting determines how often (in seconds) that Nagios will update status data in the status file. The minimum update interval is five seconds. If you have disabled aggregated status updates (with the aggregate_status_updates option), this option has no effect.

Nagios User

Format: **nagios_user=<username/UID>**
Example: **nagios_user=nagios**

This is used to set the effective user that the Nagios process should run as. After initial program startup and before starting to monitor anything, Nagios will drop its effective privileges and run as this user. You may specify either a username or a UID.

Nagios Group

Format: **nagios_group=<groupname/GID>**
Example: **nagios_group=nagios**

This is used to set the effective group that the Nagios process should run as. After initial program startup and before starting to monitor anything, Nagios will drop its effective privileges and run as this group. You may specify either a groupname or a GID.

Notifications Option

Format: **enable_notifications=<0/1>**
Example: **enable_notifications=1**

This option determines whether or not Nagios will send out notifications when it initially (re)starts. If this option is disabled, Nagios will not send out notifications for any host or service. Note: If you have state retention enabled, Nagios will ignore this setting when it (re)starts and use the last known setting for this option (as stored in the state retention file), *unless* you disable the use_retained_program_state option. If you want to change this option when state retention is active (and the use_retained_program_state is enabled), you'll have to use the appropriate external command or change it via the web interface. Values are as follows:

- 0 = Disable notifications
- 1 = Enable notifications (default)

Service Check Execution Option

Format: **execute_service_checks=<0/1>**
Example: **execute_service_checks=1**

This option determines whether or not Nagios will execute service checks when it initially (re)starts. If this option is disabled, Nagios will not actively execute any service checks and will remain in a sort of "sleep" mode (it can still accept passive checks unless you've disabled them). This option is most often used when configuring backup monitoring servers, as described in the documentation on redundancy, or when setting up a distributed monitoring environment. Note: If you have state retention enabled, Nagios will ignore this setting when it (re)starts and use the last known setting for this option (as stored in the state retention file), *unless* you disable the use_retained_program_state option. If you want to change this option when state retention is active (and the use_retained_program_state is enabled), you'll have to use the appropriate external command or change it via the web interface. Values are as follows:

- 0 = Don't execute service checks
- 1 = Execute service checks (default)

Passive Service Check Acceptance Option

Format: **accept_passive_service_checks=<0/1>**
Example: **accept_passive_service_checks=1**

This option determines whether or not Nagios will accept passive service checks when it initially (re)starts. If this option is disabled, Nagios will not accept any passive service checks. Note: If you have state retention enabled, Nagios will ignore this setting when it (re)starts and use the last known setting for this option (as stored in the state retention file), *unless* you disable the use_retained_program_state option. If you want to change this option when state retention is active (and the use_retained_program_state is enabled), you'll have to use the appropriate external command or change it via the web interface. Values are as follows:

- 0 = Don't accept passive service checks
- 1 = Accept passive service checks (default)

Event Handler Option

Format: **enable_event_handlers=<0/1>**
Example: **enable_event_handlers=1**

This option determines whether or not Nagios will run event handlers when it initially (re)starts. If this option is disabled, Nagios will not run any host or service event handlers. Note: If you have state retention enabled, Nagios will ignore this setting when it (re)starts and use the last known setting for this option (as stored in the state retention file), *unless* you disable the use_retained_program_state option. If you want to change this option when state retention is active (and the use_retained_program_state is enabled), you'll have to use the appropriate external command or change it via the web interface. Values are as follows:

- 0 = Disable event handlers
- 1 = Enable event handlers (default)

Log Rotation Method

Format: **log_rotation_method=<n/h/d/w/m>**
Example: **log_rotation_method=d**

This is the rotation method that you would like Nagios to use for your log file. Values are as follows:

- n = None (don't rotate the log - this is the default)
- h = Hourly (rotate the log at the top of each hour)
- d = Daily (rotate the log at midnight each day)
- w = Weekly (rotate the log at midnight on Saturday)
- m = Monthly (rotate the log at midnight on the last day of the month)

Log Archive Path

Format: **log_archive_path=<path>**
Example: **log_archive_path=/usr/local/nagios/var/archives/**

This is the directory where Nagios should place log files that have been rotated. This option is ignored if you choose to not use the log rotation functionality.

External Command Check Option

Format: **check_external_commands=<0/1>**
Example: **check_external_commands=1**

This option determines whether or not Nagios will check the command file for internal commands it should execute. This option must be enabled if you plan on using the command CGI to issue commands via the web interface. Third party programs can also issue commands to Nagios by writing to the command file, provided proper rights to the file have been granted as outlined in this FAQ. More information on external commands can be found here.

- 0 = Don't check external commands (default)
- 1 = Check external commands

External Command Check Interval

Format: **command_check_interval=<xxx>[s]**
Example: **command_check_interval=1**

If you specify a number with an "s" appended to it (i.e. 30s), this is the number of *seconds* to wait between external command checks. If you leave off the "s", this is the number of "time units" to wait between external command checks. Unless you've changed the interval_length value (as defined below) from the default value of 60, this number will mean minutes.

Note: By setting this value to **-1**, Nagios will check for external commands as often as possible. Each time Nagios checks for external commands it will read and process all commands present in the command file before continuing on with its other duties. More information on external commands can be found here.

External Command File

Format: **command_file=<file_name>**
Example: **command_file=/usr/local/nagios/var/rw/nagios.cmd**

This is the file that Nagios will check for external commands to process. The command CGI writes commands to this file. Other third party programs can write to this file if proper file permissions have been granted as outline in here. The external command file is implemented as a named pipe (FIFO), which is created when Nagios starts and removed when it shuts down. If the file exists when Nagios starts, the Nagios process will terminate with an error message. More information on external commands can be found here.

Downtime File

Format: **downtime_file=<file_name>**
Example: **downtime_file=/usr/local/nagios/var/downtime.log**

This is the file that Nagios will use for storing scheduled host and service downtime information. Comments can be viewed and added for both hosts and services through the extended information CGI.

Comment File

Format: **comment_file=<file_name>**

Example: **comment_file=/usr/local/nagios/var/comment.log**

This is the file that Nagios will use for storing service and host comments. Comments can be viewed and added for both hosts and services through the extended information CGI.

Lock File

Format: **lock_file=<file_name>**

Example: **lock_file=/tmp/nagios.lock**

This option specifies the location of the lock file that Nagios should create when it runs as a daemon (when started with the -d command line argument). This file contains the process id (PID) number of the running Nagios process.

State Retention Option

Format: **retain_state_information=<0/1>**

Example: **retain_state_information=1**

This option determines whether or not Nagios will retain state information for hosts and services between program restarts. If you enable this option, you should supply a value for the state_retention_file variable. When enabled, Nagios will save all state information for hosts and service before it shuts down (or restarts) and will read in previously saved state information when it starts up again.

- 0 = Don't retain state information (default)
- 1 = Retain state information

State Retention File

Format: **state_retention_file=<file_name>**

Example: **state_retention_file=/usr/local/nagios/var/status.sav**

This is the file that Nagios will use for storing service and host state information before it shuts down. When Nagios is restarted it will use the information stored in this file for setting the initial states of services and hosts before it starts monitoring anything. This file is deleted after Nagios reads in initial state information when it (re)starts. In order to make Nagios retain state information between program restarts, you must enable the retain_state_information option.

Automatic State Retention Update Interval

Format: **retention_update_interval=<minutes>**

Example: **retention_update_interval=60**

This setting determines how often (in minutes) that Nagios will automatically save retention data during normal operation. If you set this value to 0, Nagios will not save retention data at regular intervals, but it will still save retention data before shutting down or restarting. If you have disabled state retention (with the retain_state_information option), this option has no effect.

Use Retained Program State Option

Format: **use_retained_program_state=<0/1>**

Example: **use_retained_program_state=1**

This setting determines whether or not Nagios will set various program-wide state variables based on the values saved in the retention file. Some of these program-wide state variables that are normally saved across program restarts if state retention is enabled include the enable_notifications, enable_flap_detection, enable_event_handlers, execute_service_checks, and accept_passive_service_checks options. If you do not have state retention enabled, this option has no effect.

- 0 = Don't use retained program state
- 1 = Use retained program state (default)

Syslog Logging Option

Format: **use_syslog=<0/1>**

Example: **use_syslog=1**

This variable determines whether messages are logged to the syslog facility on your local host. Values are as follows:

- 0 = Don't use syslog facility
- 1 = Use syslog facility

Notification Logging Option

Format: **log_notifications=<0/1>**

Example: **log_notifications=1**

This variable determines whether or not notification messages are logged. If you have a lot of contacts or regular service failures your log file will grow relatively quickly. Use this option to keep contact notifications from being logged.

- 0 = Don't log notifications
- 1 = Log notifications

Service Check Retry Logging Option

Format: **log_service_retries=<0/1>**
Example: **log_service_retries=1**

This variable determines whether or not service check retries are logged. Service check retries occur when a service check results in a non-OK state, but you have configured Nagios to retry the service more than once before responding to the error. Services in this situation are considered to be in "soft" states. Logging service check retries is mostly useful when attempting to debug Nagios or test out service event handlers.

- 0 = Don't log service check retries
- 1 = Log service check retries

Host Check Retry Logging Option

Format: **log_host_retries=<0/1>**
Example: **log_host_retries=1**

This variable determines whether or not host check retries are logged. Logging host check retries is mostly useful when attempting to debug Nagios or test out host event handlers.

- 0 = Don't log host check retries
- 1 = Log host check retries

Event Handler Logging Option

Format: **log_event_handlers=<0/1>**
Example: **log_event_handlers=1**

This variable determines whether or not service and host event handlers are logged. Event handlers are optional commands that can be run whenever a service or hosts changes state. Logging event handlers is most useful when debugging Nagios or first trying out your event handler scripts.

- 0 = Don't log event handlers
- 1 = Log event handlers

Initial States Logging Option

Format: **log_initial_states=<0/1>**
Example: **log_initial_states=1**

This variable determines whether or not Nagios will force all initial host and service states to be logged, even if they result in an OK state. Initial service and host states are normally only logged when there is a problem on the first check. Enabling this option is useful if you are using an application that scans the log file to determine long-term state statistics for services and hosts.

- 0 = Don't log initial states (default)
- 1 = Log initial states

External Command Logging Option

Format: **log_external_commands=<0/1>**
Example: **log_external_commands=1**

This variable determines whether or not Nagios will log external commands that it receives from the external command file. Note: This option does not control whether or not passive service checks (which are a type of external command) get logged. To enable or disable logging of passive checks, use the log_passive_service_checks option.

- 0 = Don't log external commands
- 1 = Log external commands (default)

Passive Service Check Logging Option

Format: **log_passive_service_checks=<0/1>**
Example: **log_passive_service_checks=1**

This variable determines whether or not Nagios will log passive service checks that it receives from the external command file. If you are setting up a distributed monitoring environment or plan on handling a large number of passive checks on a regular basis, you may wish to disable this option so your log file doesn't get too large.

- 0 = Don't log passive service checks
- 1 = Log passive service checks (default)

Global Host Event Handler Option

Format: **global_host_event_handler=<command>**

Example: **global_host_event_handler=log-host-event-to-db**

This option allows you to specify a host event handler command that is to be run for every host state change. The global event handler is executed immediately prior to the event handler that you have optionally specified in each host definition. The *command* argument is the short name of a command that you define in your object configuration file. The maximum amount of time that this command can run is controlled by the event_handler_timeout option. More information on event handlers can be found here.

Global Service Event Handler Option

Format: **global_service_event_handler=<command>**
Example: **global_service_event_handler=log-service-event-to-db**

This option allows you to specify a service event handler command that is to be run for every service state change. The global event handler is executed immediately prior to the event handler that you have optionally specified in each service definition. The *command* argument is the short name of a command that you define in your object configuration file. The maximum amount of time that this command can run is controlled by the event_handler_timeout option. More information on event handlers can be found here.

Inter-Check Sleep Time

Format: **sleep_time=<seconds>**
Example: **sleep_time=1**

This is the number of seconds that Nagios will sleep before checking to see if the next service check in the scheduling queue should be executed. Note that Nagios will only sleep after it "catches up" with queued service checks that have fallen behind.

Inter-Check Delay Method

Format: **inter_check_delay_method=<n/d/s/x.xx>**
Example: **inter_check_delay_method=s**

This option allows you to control how service checks are initially "spread out" in the event queue. Using a "smart" delay calculation (the default) will cause Nagios to calculate an average check interval and spread initial checks of all services out over that interval, thereby helping to eliminate CPU load spikes. Using no delay is generally *not* recommended unless you are testing the service check parallelization functionality. Using no delay will cause all service checks to be scheduled for execution at the same time. This means that you will generally have large CPU spikes when the services are all executed in parallel. More information on how to estimate how the inter-check delay affects service check scheduling can be found here.Values are as follows:

- n = Don't use any delay - schedule all service checks to run immediately (i.e. at the same time!)

- d = Use a "dumb" delay of 1 second between service checks
- s = Use a "smart" delay calculation to spread service checks out evenly (default)
- x.xx = Use a user-supplied inter-check delay of x.xx seconds

Service Interleave Factor

Format: **service_interleave_factor=<s|*x*>**

Example: **service_interleave_factor=s**

This variable determines how service checks are interleaved. Interleaving allows for a more even distribution of service checks, reduced load on *remote* hosts, and faster overall detection of host problems. With the introduction of service check parallelization, remote hosts could get bombarded with checks if interleaving was not implemented. This could cause the service checks to fail or return incorrect results if the remote host was overloaded with processing other service check requests. Setting this value to 1 is equivalent to not interleaving the service checks (this is how versions of Nagios previous to 0.0.5 worked). Set this value to **s** (smart) for automatic calculation of the interleave factor unless you have a specific reason to change it. The best way to understand how interleaving works is to watch the status CGI (detailed view) when Nagios is just starting. You should see that the service check results are spread out as they begin to appear. More information on how interleaving works can be found here.

- *x* = A number greater than or equal to 1 that specifies the interleave factor to use. An interleave factor of 1 is equivalent to not interleaving the service checks.
- s = Use a "smart" interleave factor calculation (default)

Maximum Concurrent Service Checks

Format: **max_concurrent_checks=<max_checks>**

Example: **max_concurrent_checks=20**

This option allows you to specify the maximum number of service checks that can be run in parallel at any given time. Specifying a value of 1 for this variable essentially prevents any service checks from being parallelized. Specifying a value of 0 (the default) does not place any restrictions on the number of concurrent checks. You'll have to modify this value based on the system resources you have available on the machine that runs Nagios, as it directly affects the maximum load that will be imposed on the system (processor utilization, memory, etc.). More information on how to estimate how many concurrent checks you should allow can be found here.

Service Reaper Frequency

Format: **service_reaper_frequency=<frequency_in_seconds>**

Example: **service_reaper_frequency=10**

This option allows you to control the frequency *in seconds* of service "reaper" events. "Reaper" events process the results from parallelized service checks that have finished executing. These events consitute the core of the monitoring logic in Nagios.

Timing Interval Length

Format: **interval_length=<seconds>**
Example: **interval_length=60**

This is the number of seconds per "unit interval" used for timing in the scheduling queue, re-notifications, etc. "Units intervals" are used in the host configuration file to determine how often to run a service check, how often of re-notify a contact, etc.

Important: The default value for this is set to 60, which means that a "unit value" of 1 in the host configuration file will mean 60 seconds (1 minute). I have not really tested other values for this variable, so proceed at your own risk if you decide to do so!

Agressive Host Checking Option

Format: **use_agressive_host_checking=<0/1>**
Example: **use_agressive_host_checking=0**

Nagios tries to be smart about how and when it checks the status of hosts. In general, disabling this option will allow Nagios to make some smarter decisions and check hosts a bit faster. Enabling this option will increase the amount of time required to check hosts, but may improve reliability a bit. Unless you have problems with Nagios not recognizing that a host recovered, I would suggest **not** enabling this option.

- 0 = Don't use agressive host checking (default)
- 1 = Use agressive host checking

Flap Detection Option

Format: **enable_flap_detection=<0/1>**
Example: **enable_flap_detection=0**

This option determines whether or not Nagios will try and detect hosts and services that are "flapping". Flapping occurs when a host or service changes between states too frequently, resulting in a barrage of notifications being sent out. When Nagios detects that a host or service is flapping, it will temporarily suppress notifications for that host/service until it stops flapping. Flap detection is very experimental at this point, so use this feature with caution! More information on how flap detection and handling works can be found here. Note: If you have state retention enabled, Nagios will ignore this setting when it (re)starts and use the last known setting for this option (as stored in the state retention file), *unless* you disable the use_retained_program_state option. If you want to change this option when state retention is active (and the use_retained_program_state is enabled), you'll have to use the appropriate external command or change it via the web interface.

- 0 = Don't enable flap detection (default)
- 1 = Enable flap detection

Low Service Flap Threshold

Format: **low_service_flap_threshold=<percent>**
Example: **low_service_flap_threshold=25.0**

This option is used to set the low threshold for detection of service flapping. For more information on how flap detection and handling works (and how this option affects things) read this.

High Service Flap Threshold

Format: **high_service_flap_threshold=<percent>**
Example: **high_service_flap_threshold=50.0**

This option is used to set the low threshold for detection of service flapping. For more information on how flap detection and handling works (and how this option affects things) read this.

Low Host Flap Threshold

Format: **low_host_flap_threshold=<percent>**
Example: **low_host_flap_threshold=25.0**

This option is used to set the low threshold for detection of host flapping. For more information on how flap detection and handling works (and how this option affects things) read this.

High Host Flap Threshold

Format: **high_host_flap_threshold=<percent>**
Example: **high_host_flap_threshold=50.0**

This option is used to set the low threshold for detection of host flapping. For more information on how flap detection and handling works (and how this option affects things) read this.

Soft Service Dependencies Option

Format: **soft_state_dependencies=<0/1>**

Example: **soft_state_dependencies=0**

This option determines whether or not Nagios will use soft service state information when checking service dependencies. Normally Nagios will only use the latest hard service state when checking dependencies. If you want it to use the latest state (regardless of whether its a soft or hard state type), enable this option.

- 0 = Don't use soft service state dependencies (default)
- 1 = Use soft service state dependencies

Service Check Timeout

Format: **service_check_timeout=<seconds>**
Example: **service_check_timeout=60**

This is the maximum number of seconds that Nagios will allow service checks to run. If checks exceed this limit, they are killed and a CRITICAL state is returned. A timeout error will also be logged.

There is often widespread confusion as to what this option really does. It is meant to be used as a last ditch mechanism to kill off plugins which are misbehaving and not exiting in a timely manner. It should be set to something high (like 60 seconds or more), so that each service check normally finishes executing within this time limit. If a service check runs longer than this limit, Nagios will kill it off thinking it is a runaway processes.

Host Check Timeout

Format: **host_check_timeout=<seconds>**
Example: **host_check_timeout=60**

This is the maximum number of seconds that Nagios will allow host checks to run. If checks exceed this limit, they are killed and a CRITICAL state is returned and the host will be assumed to be DOWN. A timeout error will also be logged.

There is often widespread confusion as to what this option really does. It is meant to be used as a last ditch mechanism to kill off plugins which are misbehaving and not exiting in a timely manner. It should be set to something high (like 60 seconds or more), so that each host check normally finishes executing within this time limit. If a host check runs longer than this limit, Nagios will kill it off thinking it is a runaway processes.

Event Handler Timeout

Format: **event_handler_timeout=<seconds>**
Example: **event_handler_timeout=60**

This is the maximum number of seconds that Nagios will allow event handlers to be run. If an event handler exceeds this time limit it will be killed and a warning will be logged.

There is often widespread confusion as to what this option really does. It is meant to be used as a last ditch mechanism to kill off commands which are misbehaving and not exiting in a timely manner. It should be set to something high (like 60 seconds or more), so that each event handler command normally finishes executing within this time limit. If an event handler runs longer than this limit, Nagios will kill it off thinking it is a runaway processes.

Notification Timeout

Format:	**notification_timeout=<seconds>**
Example:	**notification_timeout=60**

This is the maximum number of seconds that Nagios will allow notification commands to be run. If a notification command exceeds this time limit it will be killed and a warning will be logged.

There is often widespread confusion as to what this option really does. It is meant to be used as a last ditch mechanism to kill off commands which are misbehaving and not exiting in a timely manner. It should be set to something high (like 60 seconds or more), so that each notification command finishes executing within this time limit. If a notification command runs longer than this limit, Nagios will kill it off thinking it is a runaway processes.

Obsessive Compulsive Service Processor Timeout

Format:	**ocsp_timeout=<seconds>**
Example:	**ocsp_timeout=5**

This is the maximum number of seconds that Nagios will allow an obsessive compulsive service processor command to be run. If a command exceeds this time limit it will be killed and a warning will be logged.

Performance Data Processor Command Timeout

Format:	**perfdata_timeout=<seconds>**
Example:	**perfdata_timeout=5**

This is the maximum number of seconds that Nagios will allow a host performance data processor command or service performance data processor command to be run. If a command exceeds this time limit it will be killed and a warning will be logged.

Obsess Over Services Option

Format:	**obsess_over_services=<0/1>**

Example: **obsess_over_services=1**

This value determines whether or not Nagios will "obsess" over service checks results and run the obsessive compulsive service processor command you define. I know - funny name, but it was all I could think of. This option is useful for performing distributed monitoring. If you're not doing distributed monitoring, don't enable this option.

- 0 = Don't obsess over services (default)
- 1 = Obsess over services

Obsessive Compulsive Service Processor Command

Format: **ocsp_command=<command>**
Example: **ocsp_command=obsessive_service_handler**

This option allows you to specify a command to be run after *every* service check, which can be useful in distributed monitoring. This command is executed after any event handler or notification commands. The *command* argument is the short name of a command definition that you define in your host configuration file. The maximum amount of time that this command can run is controlled by the ocsp_timeout option. More information on distributed monitoring can be found here.

Performance Data Processing Option

Format: **process_performance_data=<0/1>**
Example: **process_performance_data=1**

This value determines whether or not Nagios will process host and service check performance data.

- 0 = Don't process performance data (default)
- 1 = Process performance data

Orphaned Service Check Option

Format: **check_for_orphaned_services=<0/1>**
Example: **check_for_orphaned_services=0**

This option allows you to enable or disable checks for orphaned service checks. Orphaned service checks are checks which ahve been executed and have been removed from the event queue, but have not had any results reported in a long time. Since no results have come back in for the service, it is not rescheduled in the event queue. This can cause service checks to stop being executed. Normally it is very rare for this to happen - it might happen if an external user or process killed off the process that was being used to execute a service check. If this option is enabled and Nagios finds that results for a particular service check have not come back, it will log an error message and reschedule the service check. If you start seeing service checks that never seem to get rescheduled, enable this option and see if you notice any log messages about orphaned services.

- 0 = Don't check for orphaned service checks (default)
- 1 = Check for orphaned service checks

Service Freshness Checking Option

Format: **check_service_freshness=<0/1>**
Example: **check_service_freshness=0**

This option determines whether or not Nagios will periodically check the "freshness" of service checks. Enabling this option is useful for helping to ensure that passive service checks are received in a timely manner. More information on freshness checking can be found here.

- 0 = Don't check service freshness
- 1 = Check service freshness (default)

Service Freshness Check Interval

Format: **freshness_check_interval=<seconds>**
Example: **freshness_check_interval=60**

This setting determines how often (in seconds) Nagios will periodically check the "freshness" of service check results. If you have disabled service freshness checking (with the check_service_freshness option), this option has no effect. More information on freshness checking can be found here.

Illegal Object Name Characters

Format: **illegal_object_name_chars=<chars...>**
Example: **illegal_object_name_chars=`~!$%^&*"|'<>?,()=**

This options allows you to specify illegal characters that cannot be used in host names, service descriptions, or names of other object types. Nagios will allow you to use most characters in object definitions, but I recommend not using the characters shown in the example above. Doing may give you problems in the web interface, notification commands, etc.

Illegal Macro Output Characters

Format: **illegal_macro_output_chars=<chars...>**
Example: **illegal_macro_output_chars=`~$^&"|'<>**

This options allows you to specify illegal characters that should be stripped from macros before being used in notifications, event handlers, and other commands. This DOES NOT affect macros used in service or host check commands. You can choose to not strip out the characters shown in the example above, but I recommend you do not do this. Some of these characters are interpreted by the shell (i.e. the backtick) and can lead to security problems. The following macros are stripped of the characters you specify:

$OUTPUT$, **$PERFDATA$**

Administrator Email Address

Format:	**admin_email=<email_address>**
Example:	**admin_email=root@localhost.localdomain**

This is the email address for the administrator of the local machine (i.e. the one that Nagios is running on). This value can be used in notification commands by using the **$ADMINEMAIL$** macro.

Administrator Pager

Format:	**admin_pager=<pager_number_or_pager_email_gateway>**
Example:	**admin_pager=pageroot@localhost.localdomain**

This is the pager number (or pager email gateway) for the administrator of the local machine (i.e. the one that Nagios is running on). The pager number/address can be used in notification commands by using the **$ADMINPAGER$** macro.

Object Configuration Data

What is Object Data?

Object data is simply a generic term I use to describe various data definitions you need in order to monitor anything. Types of object definitions include:

- Services
- Hosts
- Host Groups
- Contacts
- Contact Groups
- Commands
- Time Periods
- Service Escalations
- Service Dependencies
- Host Escalations
- Host Dependencies
- Hostgroup Escalations

How Do You Define Object Data?

That all depends on how you compiled the core program and CGIs. Or more correctly, it depends on what options you supplied to the configure script before you compiled everything. There are two different methods for storing object definitions. They are...

- **Default (old) method** - This is the old style of configuring objects and is provided for backward compatabilty. I *do not* recommend using this method for storing object definitions.
- **Template-based method** - This is the new style of configuring object definitions that is flexible and easy to understand. You can use templates to define entries for multiple hosts, services, etc quickly and easily.

CGI Configuration File Options

Notes

When creating and/or editing configuration files, keep the following in mind:

1. Lines that start with a '#' character are taken to be comments and are not processed
2. Variables names must begin at the start of the line - no white space is allowed before the name
3. Variable names are case-sensitive

Sample Configuration

A sample CGI configuration file can be created by running the '**make config**' command. The default name of the CGI configuration file is **cgi.cfg**.

Index

Main Configuration File Location

Format: **main_config_file=<file_name>**

Example: **main_config_file=/usr/local/nagios/etc/nagios.cfg**

This specifies the location of your main configuration file. The CGIs need to know where to find this file in order to get information about configuration information, current host and service status, etc.

Physical HTML Path

Format: **physical_html_path=<path>**

Example: **physical_html_path=/usr/local/nagios/share**

This is the *physical* path where the HTML files for Nagios are kept on your workstation or server. Nagios assumes that the documentation and images files (used by the CGIs) are stored in subdirectories called *docs/* and *images/*, respectively.

URL HTML Path

Format: **url_html_path=<path>**

Example: **url_html_path=/nagios**

If, when accessing Nagios via a web browser, you point to an URL like **http://www.myhost.com/nagios**, this value should be */nagios*. Basically, its the path portion of the URL that is used to access the Nagios HTML pages.

Nagios Process Check Command

Format: **nagios_check_command=<command_line>**

Example: **nagios_check_command=/usr/local/nagios/libexec/check_nagios /usr/local/nagios/var/status.log 5 '/usr/local/nagios/bin/nagios -d /usr/local/nagios/etc/nagios.cfg'**

This is an *optional* command that the CGIs can use to check the status of the Nagios process. This provides the CGIs (as well as yourself) with some idea of whether or not Nagios is still running. If you do not specify a command to be run, the CGIs will assume that the Nagios process is running. If you do define a process check command, it should follow the same guidelines that are required of standard plugins. If the command returns a non-OK status, the CGIs will think the Nagios process is not running and will refuse to allow you to commit any commands via the command CGI.

Authentication Usage

Format: **use_authentication=<0/1>**

Example: **use_authentication=1**

This option controls whether or not the CGIs will use the authentication and authorization functionality when determining what information and commands users have access to. I would strongly suggest that you use the authentication functionality for the CGIs. If you decide not to use authentication, make sure to remove the command CGI to prevent unauthorized users from issuing commands to Nagios. The CGI will not issue commands to Nagios if authentication is disabled, but I would suggest removing it altogether just to be on the safe side. More information on how to setup authentication and configure authorization for the CGIs can be found here.

- 0 = Don't use authentication functionality
- 1 = Use authentication and authorization functionality (default)

Default User Name

Format: **default_user_name=<username>**

Example: **default_user_name=guest**

Setting this variable will define a default username that can access the CGIs. This allows people within a secure domain (i.e., behind a firewall) to access the CGIs without necessarily having to authenticate to the web server. You may want to use this to avoid having to use basic authentication if you are not using a secure server, as basic authentication transmits passwords in clear text over the Internet.

Important: Do *not* define a default username unless you are running a secure web server and are sure that everyone who has access to the CGIs has been authenticated in some manner! If you define this variable, anyone who has not authenticated to the web server will inherit all rights you assign to this user!

System/Process Information Access

Format: **authorized_for_system_information=<user1>,<user2>,<user3>,...<user*n*>**

Example: **authorized_for_system_information=nagiosadmin,theboss**

This is a comma-delimited list of names of *authenticated users* who can view system/process information in the extended information CGI. Users in this list are *not* automatically authorized to issue system/process commands. If you want users to be able to issue system/process commands as well, you must add them to the authorized_for_system_commands variable. More information on how to setup authentication and configure authorization for the CGIs can be found here.

System/Process Command Access

Format: **authorized_for_system_commands=<user1>,<user2>,<user3>,...<user*n*>**

Example: **authorized_for_system_commands=nagiosadmin**

This is a comma-delimited list of names of *authenticated users* who can issue system/process commands via the command CGI. Users in this list are *not* automatically authorized to view system/process information. If you want users to be able to view system/process information as well, you must add them to the authorized_for_system_information variable. More information on how to setup authentication and configure authorization for the CGIs can be found here.

Configuration Information Access

Format: **authorized_for_configuration_information=<user1>,<user2>,<user3>,...<user*n*>**

Example: **authorized_for_configuration_information=nagiosadmin**

This is a comma-delimited list of names of *authenticated users* who can view configuration information in the configuration CGI. Users in this list can view information on all configured hosts, host groups, services, contacts, contact groups, time periods, and commands. More information on how to setup authentication and configure authorization for the CGIs can be found here.

Global Host Information Access

Format: **authorized_for_all_hosts=<user1>,<user2>,<user3>,...<user*n*>**

Example: **authorized_for_all_hosts=nagiosadmin,theboss**

This is a comma-delimited list of names of *authenticated users* who can view status and configuration information for all hosts. Users in this list are also automatically authorized to view information for all services. Users in this list are *not* automatically authorized to issue commands for all hosts or services. If you want users able to issue commands for all hosts and services as well, you must add them to the authorized_for_all_host_commands variable. More information on how to setup authentication and configure authorization for the CGIs can be found here.

Global Host Command Access

Format: **authorized_for_all_host_commands=<user1>,<user2>,<user3>,...<user*n*>**

Example: **authorized_for_all_host_commands=nagiosadmin**

This is a comma-delimited list of names of *authenticated users* who can issue commands for all hosts via the command CGI. Users in this list are also automatically authorized to issue commands for all services. Users in this list are *not* automatically authorized to view status or configuration information for all hosts or services. If you want users able to view status and configuration information for all hosts and services as well, you must add them to the authorized_for_all_hosts variable. More information on how to setup authentication and configure authorization for the CGIs can be found here.

Global Service Information Access

Format: **authorized_for_all_services=<user1>,<user2>,<user3>,...<user*n*>**

Example: **authorized_for_all_services=nagiosadmin,theboss**

This is a comma-delimited list of names of *authenticated users* who can view status and configuration information for all services. Users in this list are *not* automatically authorized to view information for all hosts. Users in this list are *not* automatically authorized to issue commands for all services. If you want users able to issue commands for all services as well, you must add them to the authorized_for_all_service_commands variable. More information on how to setup authentication and configure authorization for the CGIs can be found here.

Global Service Command Access

Format: **authorized_for_all_service_commands=<user1>,<user2>,<user3>,...<user*n*>**

Example: **authorized_for_all_service_commands=nagiosadmin**

This is a comma-delimited list of names of *authenticated users* who can issue commands for all services via the command CGI. Users in this list are *not* automatically authorized to issue commands for all hosts. Users in this list are *not* automatically authorized to view status or configuration information for all hosts. If you want users able to view status and configuration information for all services as well, you must add them to the authorized_for_all_services variable. More information on how to setup authentication and configure authorization for the CGIs can be found here.

Statusmap CGI Background Image

Format: **statusmap_background_image=<gd2_image>**

Example: **statusmap_background_image=statusmapbg.gd2**

This option allows you to specify an image to be used as a background in the statusmap CGI. It is assumed that the image resides in the HTML images path (i.e. /usr/local/nagios/share/images). This path is automatically determined by appending "/images" to the path specified by the physical_html_path directive. Note: The image file must be in GD2 format (preferably in uncompressed format)!

Default Statusmap Layout Method

Format: **default_statusmap_layout=<layout_number>**

Example: **default_statusmap_layout=4**

This option allows you to specify the default layout method used by the statusmap CGI. Valid options are:

<layout_number> Value	Layout Method
0	User-defined coordinates
1	Depth layers
2	Collapsed tree
3	Balanced tree
4	Circular
5	Circular (Marked Up)
6	Circular (Balloon)

Statuswrl CGI Include World

Format:	**statuswrl_include=<vrml_file>**
Example:	**statuswrl_include=myworld.wrl**

This option allows you to include your own objects in the generated VRML world. It is assumed that the file resides in the path specified by the physical_html_path directive. Note: This file must be a fully qualified VRML world (i.e. you can view it by itself in a VRML browser).

Default Statuswrl Layout Method

Format:	**default_statuswrl_layout=<layout_number>**
Example:	**default_statuswrl_layout=4**

This option allows you to specify the default layout method used by the statuswrl CGI. Valid options are:

<layout_number> Value	Layout Method
0	User-defined coordinates
2	Collapsed tree
3	Balanced tree
4	Circular

CGI Refresh Rate

Format: **refresh_rate=<rate_in_seconds>**
Example: **refresh_rate=90**

This option allows you to specify the number of seconds between page refreshes for the status, statusmap, and extinfo CGIs.

Audio Alerts

Formats: **host_unreachable_sound=<sound_file>**
host_down_sound=<sound_file>
service_critical_sound=<sound_file>
service_warning_sound=<sound_file>
service_unknown_sound=<sound_file>

Examples: **host_unreachable_sound=hostu.wav**
host_down_sound=hostd.wav
service_critical_sound=critical.wav
service_warning_sound=warning.wav
service_unknown_sound=unknown.wav

These options allow you to specify an audio file that should be played in your browser if there are problems when you are viewing the status CGI. If there are problems, the audio file for the most critical type of problem will be played. The most critical type of problem is on or more unreachable hosts, while the least critical is one or more services in an unknown state (see the order in the example above). Audio files are assumed to be in the **media/** subdirectory in your HTML directory (i.e. */usr/local/nagios/share/media*).

Ping Syntax

Format: **ping_syntax=<command>**
Example: **ping_syntax=/bin/ping -n -U -c 5 $HOSTADDRESS$**

This option determines what syntax should be used when attempting to ping a host from the WAP interface (using the statuswml CGI. You must include the full path to the ping binary, along with all required options. The $HOSTADDRESS$ macro is substituted with the address of the host before the command is executed.

Authentication And Authorization In The CGIs

Notes

Throughout these instructions I will be assuming that you are running the Apache web server on your machine. If you are running some other web server, you will have to make some adjustments.

Definitions

Throughout these instructions I will be using the following terms, so you should understand what they mean...

- An *authenticated user* is an someone who has authenticated to the web server with a username and password and has been granted access to the CGIs by the web server
- An *authenticated contact* is an authenticated user whose username matches the short name of a contact definition in your object configuration file(s).

Index

Configuring Web Server Authentication

The first step to configuring your web server for authentication is to make sure the web server configuration file (i.e. **httpd.conf**) file contains an '**AuthOverride AuthConfig**' statement in it for the Nagios CGI-BIN directory. If it doesn't, you'll have to add something similiar to the following to your web server configuration file. Note that you will have to restart the web server in order for this change to take effect.

```
<Directory /usr/local/nagios/sbin>
AllowOverride AuthConfig
order allow,deny
allow from all
Options ExecCGI
</Directory>
```

If you also want to require authentication for access the HTML pages for Nagios, add something similiar to the following in the web server configuration file as well.

```
<Directory /usr/local/nagios/share>
AllowOverride AuthConfig
order allow,deny
allow from all
</Directory>
```

The second step is to create a file named **.htaccess** in the root your CGI directory (and optionally also you HTML directory) for Nagios (usualy /usr/local/nagios/sbin and /usr/local/nagios/share, respectively). The file(s) should have contents similiar to the following...

```
AuthName "Nagios Access"
AuthType Basic
AuthUserFile /usr/local/nagios/etc/htpasswd.users
require valid-user
```

Setting Up Authenticated Users

Now that you've configured the web server to require authentication for access to the CGIs, you'll need to configure users who can acess the CGIs. This is done by using the **htpasswd** command supplied with Apache.

Running the following command will create a new file called *htpasswd.users* in the */usr/local/nagios/etc* directory. It will also create an username/password entry for *nagiosadmin*. You will be asked to provide a password that will be used when *nagiosadmin* authenticates to the web server.

```
htpasswd -c /usr/local/nagios/etc/htpasswd.users nagiosadmin
```

Continue adding more users until you've created an account for everyone you want to access the CGIs. Use the following command to add additional users, replacing <username> with the actual username you want to add. Note that the **-c** option is not used, since you already created the initial file.

```
htpasswd /usr/local/nagios/etc/htpasswd.users <username>
```

Okay, so you're done with the first part of what needs to be done. If you point your web browser to your Nagios CGIs you should be asked for a username and password. If you have problems getting user authentication to work at this point, read your webserver documentation for more info.

Enabling Authentication/Authorization Functionality In The CGIs

The next thing you need to do is make sure that the CGIs are configured to use the authentication and authorization functionality in determining what information and/or commands users have access to. This is done be setting the use_authentication variable in the CGI configuration file to a non-zero value. Example:

```
use_authentication=1
```

Okay, you're now done with setting up basic authentication/authorization functionality in the CGIs.

Default Permissions To CGI Information

So what default permissions do users have in the CGIs by default when the authentication/authorization functionality is enabled?

CGI Data	Authenticated Contacts*	Other Authenticated Users*
Host Status Information	Yes	No
Host Configuration Information	Yes	No
Host History	Yes	No
Host Notifications	Yes	No
Host Commands	Yes	No
Service Status Information	Yes	No
Service Configuration Information	Yes	No
Service History	Yes	No
Service Notifications	Yes	No
Service Commands	Yes	No
All Configuration Information	No	No
System/Process Information	No	No
System/Process Commands	No	No

*Authenticated contacts** are granted the following permissions for each **service** for which they are contacts (but not for services for which they are not contacts)...

- Authorization to view service status information
- Authorization to view service configuration information
- Authorization to view history and notifications for the service
- Authorization to issue service commands

*Authenticated contacts** are granted the following permissions for each **host** for which they are contacts (but not for hosts for which they are not contacts)...

- Authorization to view host status information
- Authorization to view host configuration information
- Authorization to view history and notifications for the host
- Authorization to issue host commands
- Authorization to view status information for all services on the host
- Authorization to view configuration information for all services on the host
- Authorization to view history and notification information for all services on the host
- Authorization to issue commands for all services on the host

It is important to note that by default **no one** is authorized for the following...

- Viewing the raw log file via the showlog CGI
- Viewing Nagios process information via the extended information CGI
- Issuing Nagios process commands via the command CGI
- Viewing host group, contact, contact group, time period, and command definitions via the configuration CGI
-

You will undoubtably want to access this information, so you'll have to assign additional rights for yourself (and possibly other users) as described below...

Granting Additional Permissions To CGI Information

You can grant *authenticated contacts* or other *authenticated users* permission to additional information in the CGIs by adding them to various authorization variables in the CGI configuration file. I realize that the available options don't allow for getting really specific about particular permissions, but its better than nothing..

Additional authorization can be given to users by adding them to the following variables in the CGI configuration file...

- authorized_for_system_information
- authorized_for_system_commands
- authorized_for_configuration_information
- authorized_for_all_hosts
- authorized_for_all_host_commands
- authorized_for_all_services
- authorized_for_all_service_commands

CGI Authorization Requirements

If you are confused about the authorization needed to access various information in the CGIs, read the ***Authorization Requirements*** section for each CGI as described here.

Authentication On Secured Web Servers

If your web server is located in a secure domain (i.e., behind a firewall) or if you are using SSL, you can define a default username that can be used to access the CGIs. This is done by defining the default_user_name option in the CGI configuration file. By defining a default username that can access the CGIs, you can allow users to access the CGIs without necessarily having to authenticate to the web server.. You may want to use this to avoid having to use basic web authentication, as basic authentication transmits passwords in clear text over the Internet.

Important: Do *not* define a default username unless you are running a secure web server and are sure that everyone who has access to the CGIs has been authenticated in some manner! If you define this variable, anyone who has not authenticated to the web server will inherit all rights you assign to this user!

Extended Information Configuration

What is Extended Information?

Extended information consists of *optional* definitions for hosts and services that is used by the CGIs in the following ways:

- to provide URLs to additional information about the host or service
- to add pretty icons to the hosts and services displayed in the web interface
- to draw hosts in the statusmap and statuswrl CGIs at user-defined 2-D and 3-D coordinates

Where is Extended Information Defined?

That all depends on how you compiled the CGIs. Or more correctly, it depends on what options you supplied to the configure script before you compiled everything. There are several different methods for storing extended information definitions. They are as follows...

- **Default (old) method** - Definitions are stored in the CGI config file. This method is provided for backward compatabilty. I recommend that you use one of the methods described below.
- **Template-based method** - Definitions are stored in seperate configuration files. You can use templates to define entries for multiple hosts and services quickly and easily.
- **Database method** - Definitions are stored in a database.

Verifying Your Nagios Configuration

Verifying The Configuration From The Command Line

Once you've entered all the necessary data into the configuration files, its time to do a sanity check. Everyone make mistakes from time to time, so its best to verify what you've entered. Nagios automatically runs a "pre-flight check" before before it starts monitoring, but you also have the option of running this check manually before attempting to start Nagios. In order to do this, you must start Nagios with the **-v** command line argument as follows...

/usr/local/nagios/bin/nagios -v <main_config_file>

Note that you should be entering the path/filename of your *main* configuration file (i.e. */usr/local/nagios/etc/nagios.cfg*) as the second argument. Nagios will read your main configuration file and all object configuration files and verify that they contain valid data.

Relationships Verified During The Pre-Flight Check

During the "pre-flight check", Nagios verifies that you have defined the data relationships necessary for monitoring. Objects are all related and need to be setup properly in order for things to run. This is a list of the basic things that Nagios attempts to check before it will start monitoring...

1. Verify that all contacts are a member of at least one contact group.
2. Verify that all contacts specified in each contact group are valid.
3. Verify that all hosts are a member of at least one host group.
4. Verify that all hosts specified in each host group are valid.
5. Verify that all hosts have at least one service associated with them.
6. Verify that all commands used in service and host checks are valid.
7. Verify that all commands used in service and host event handlers are valid.
8. Verify that all commands used in contact service and host notifications are valid.
9. Verify that all notification time periods specified for services, hosts, and contact are valid.
10. Verify that all service check time periods specified for services are valid.

Fixing Configuration Errors

If you've forgotten to enter some critical data or just plain screwed things up, Nagios will spit out a warning or error message that should point you to the location of the problem. Error messages generally print out the line in the configuration file that seems to be the source of the problem. On errors, Nagios will often exit the pre-flight check and return to the command prompt after printing only the first error that it has encountered. This is done so that one error does not cascade into multiple errors as the remainder of the configuration data is verified. If you get any error messages you'll need to go and edit your configuration files to remedy the problem. Warning messages can *generally* be safely ignored, since they are only recommendations and not requirements.

Where To Go From Here

Once you've verified your configuration files and fixed any errors, you can be reasonably sure that Nagios will start monitoring the services you've specified. On to starting Nagios!

Starting Nagios

IMPORTANT: Before you actually start Nagios, you'll have to make sure that you have configured it properly and verified the config data!

Methods For Starting Nagios

There are basically four different ways you can start Nagios:

1. Manually, as a foreground process (useful for initial testing and debugging)
2. Manually, as a background process
3. Manually, as a daemon
4. Automatically at system boot

Let's examine each method briefly...

Running Nagios Manually as a Foreground Process

If you enabled the debugging options when running the configure script (and recompiled Nagios), this would be your first choice for testing and debugging. Running Nagios as a foreground process at a shell prompt will allow you to more easily view what's going on in the monitoring and notification processes. To run Nagios as a foreground process for testing, invoke Nagios like this...

/usr/local/nagios/bin/nagios <main_config_file>

Note that you must specify the path/filename of the main configuration file (i.e. */usr/local/nagios/etc/nagios.cfg*) on the command line.

To stop Nagios at any time, just press CTRL-C. If you've enabled the debugging options you'll probably want to redirect the output to a file for easier review later.

Running Nagios Manually as a Background Process

To run Nagios as a background process, invoke it with an ampersand as follows...

/usr/local/nagios/bin/nagios <main_config_file> &

Note that you must specify the path/filename of the main configuration file (i.e. */usr/local/nagios/etc/nagios.cfg*) on the command line.

Running Nagios Manually as a Daemon

In order to run Nagios in daemon mode you must supply the **-d** switch on the command line as follows...

/usr/local/nagios/bin/nagios -d <main_config_file>

Note that you must specify the path/filename of the main configuration file (i.e. */usr/local/nagios/etc/nagios.cfg*) on the command line.

Running Nagios Automatically at System Boot

When you have tested Nagios and are reasonably sure that it is not going to crash, you will probably want to have it start automatically at boot time. To do this (in Linux) you will have to create a startup script in your **/etc/rc.d/init.d/** directory. You will also have to create a link to the script in the runlevel(s) that you wish to have Nagios to start in. I'll assume that you know what I'm talking about and are able to do this.

A sample init script (named **daemon-init**) is created in the base directory of the Nagios distribution when you run the configure script. You can install the sample script to your /etc/rc.d/init.d directory using the '**make install-init**' command, as outlined in the installation instructions.

The sample init scripts are designed to work under Linux, so if you want to use them under FreeBSD, Solaris, etc. you may have to do a little hacking...

Stopping and Restarting Nagios

Directions on how to stop and restart Nagios can be found here.

Stopping And Restarting Nagios

Once you have Nagios up and running, you may need to stop the process or reload the configuration data "on the fly". This section describes how to do just that.

IMPORTANT: Before you restart Nagios, make sure that you have verified the configuration data using the -v command line switch, *especially* if you have made any changes to your config files. If Nagios encounters problem with one of the config files when it restarts, it will log an error and terminate.

Stopping And Restarting With The Init Script

If you have installed the sample init script to your /etc/rc.d/init.d directory you can stop and restart Nagios easily. If you haven't, skip this section and read how to do it manually below. I'll assume that you named the init script **Nagios** in the examples below...

Desired Action	Command	Description
Stop Nagios	**/etc/rc.d/init.d/nagios stop**	This kills the Nagios process
Restart Nagios	**/etc/rc.d/init.d/nagios restart**	This kills the current Nagios process and then starts Nagios up again
Reload Configuration Data	**/etc/rc.d/init.d/nagios reload**	Sends a SIGHUP to the Nagios process, causing it to flush its current configuration data, reread the configuration files, and start monitoring again

Stopping, restarting, and reloading Nagios are fairly simple with an init script and I would highly recommend you use one if at all possible.

Stopping and Restarting Nagios Manually

If you aren't using an init script to start Nagios, you'll have to do things manually. First you'll have to find the process ID that Nagios is running under and then you'll have to use the *kill* command to terminate the application or make it reload the configuration data by sending it the proper signal. Directions for doing this are outlined below...

Finding The Nagios Process ID

First off, you will need to know the process id that Nagios is running as. To do that, just type the following command at a shell prompt:

```
ps axu | grep nagios
```

The output should look something like this:

```
nagios  6808  0.0  0.7   840   352  p3 S    13:44   0:00 grep nagios
nagios 11149  0.2  1.0   868   488  ?  S   Feb 27   6:33 /usr/local/nagios/bin/nagios nagios.cfg
```

From the program output, you will notice that Nagios was started by user **nagios** and is running as process id **11149**.

Manually Stopping Nagios

In order to stop Nagios, use the *kill* command as follows...

```
kill 11149
```

You should replace **11149** with the actual process id that Nagios is running as on your machine.

Manually Restarting Nagios

If you have modified the configuration data, you will want to restart Nagios and have it re-read the new configuration. If you have changed the source code and recompiled the main Nagios executable you should *not* use this method. Instead, stop Nagios by killing it (as outlined above) and restart it manually. Restarting Nagios using the method below does not actually reload Nagios - it just causes Nagios to flush its current configuration, re-read the new configuration, and start monitoring all over again. To restart Nagios, you need to send the **SIGHUP** signal to Nagios. Assuming that the process id for Nagios is **11149** (taken from the example above), use the following command:

```
kill -HUP 11149
```

Remember, you will need to replace **11149** with the actual process id that Nagios is running as on your machine.

Nagios Plugins

What Are Plugins?

Plugins are compiled executables or scripts (Perl, shell, etc.) that can be run from a command line to check the status or a host or service. Nagios uses the results from plugins to determine the current status or hosts and services on your network. No, you can't get away without using plugins - Nagios is useless without them.

Obtaining Plugins

Plugin development for Nagios is being done at SourceForge. The Nagios plugin development project page (where the latest version of by plugins can always be found) is located at http://sourceforge.net/projects/nagiosplug/.

How Do I Use Plugin *X*?

Documentation on how to use individual plugins is *not* supplied with the core Nagios distribution. You should refer to the latest plugin distribution for information on using plugins. Karl DeBisschop, lead plugin developer/maintainer points out the following:

> All plugins that comply with minimal development guideline for this project include internal documentation. The documentation can be read executing plugin with the '-h' option ('--help' if long options are enabled). If the '-h' option does not work, that is a bug.

For example, if you want to know how the check_http plugin works or what options it accepts, you should try executing either:

./check_httpd --help

or

./check_httpd --h

Command Definition Examples For Services

It is important to note that command definitions found in sample config files in the core Nagios distribution are probably *not* accurate as to command line parameters, etc when it comes to the plugins. They are simply provided as examples of how to define commands.

Creating Custom Plugins

Creating your own plugins to perform custom host or service checks is easy. You can find information on how to write plugins at http://sourceforge.net/projects/nagiosplug/ .

Nagios Addons

The following is a description of various "addons" that are available for Nagios. These and other addons can be obtained from the downloads page on the Nagios website (www.nagios.org).

Index

nrpe - Daemon and plugin for executing plugins on remote hosts

Author: Me

Overview: Allows you to execute plugins on remote hosts in a relatively easy and transparent manner.

Files:

check_nrpe	- Plugin used to send execution requests to the nrpe agent on the remote host
nrpe	- Agent that runs on the remote host and processes plugin execution requests
nrpe.cfg	- Configuration file for the remote host agent

Description: This addon is designed to provide a way for executing plugins on a remote host. The check_nrpe plugin runs on the Nagios host and is used to send plugin execution requests to the nrpe agent on the remote host. The nrpe agent will then run an appropriate plugins on the remote host and return the plugin output and return code to the check_nrpe plugin on the Nagios host. The check_nrpe plugin then passes the remote plugin's output and return code back to Nagios as if it were its own. This allows for a rather transparent method of executing plugins on remote hosts. The nrpe agent can either be run as a standalone daemon or as a service under inetd.

Notes:

- When running in daemon mode, the nrpe agent authenticates plugin execution requests by doing a rudimentary comparison of the IP address of the calling host against a list of allowed IP addresses in the configuration file.
- When running under inetd, TCP wrappers can be employed to restrict access to the nrpe agent

nsca - Daemon and client program for sending passive check results across the network

Author: Me

Overview: Allows you to submit passive service checks results to another server on the network that is running Nagios.

Files:	nsca	- Daemon that runs on the central Nagios server and processes passive service check results submitted by clients
	nsca.cfg	- Configuration file for the nsca daemon
	send_nsca	- Client program that is executed from remote hosts and sends passive service check information to the nsca daemon on the central Nagios server
	send_nsca.cfg	- Configuration file for the send_nsca client
Description:	This addon allows you to send passive service check results from remote hosts to a central monitoring host that runs Nagios. The client can be used as a standalone program or can be integrated with remote Nagios servers that run an ocsp command to setup a distributed monitoring environment. Communication between the client and daemon can be encrypted via various algorithms (DES, 3DES, CAST, xTEA, Twofish, LOKI97, RJINDAEL, SERPENT, GOST, SAFER/SAFER+, etc.) if you have the mcrypt libraries installed on your systems.	

Determining Status and Reachability of Network Hosts

Monitoring Services on Down or Unreachable Hosts

The main purpose of Nagios is to monitor services that run on or are provided by physical hosts or devices on your network. It should be obvious that if a host or device on your network goes down, all services that it offers will also go down with it. Similarly, if a host becomes unreachable, Nagios will not be able to monitor the services associated with that host.

Nagios recognizes this fact and attempts to check for such a scenario when there are problems with a service. Whenever a service check results in a non-OK status level, Nagios will attempt to check and see if the host that the service is running on is "alive". Typically this is done by pinging the host and seeing if any response is received. If the host check commmand returns a non-OK state, Nagios assumes that there is a problem with the host. In this situation Nagios will "silence" all potential alerts for services running on the host and just notify the appropriate contacts that the host is down or unreachable. If the host check command returns an OK state, Nagios will recognize that the host is alive and will send out an alert for the service that is misbehaving.

Local Hosts

"Local" hosts are hosts that reside on the same network segment as the host running Nagios - no routers or firewalls lay between them. Figure 1 shows an example network layout. Host A is running Nagios and monitoring all other hosts and routers depicted in the diagram. Hosts B, C, D, E and F are all considered to be "local" hosts in relation to host A.

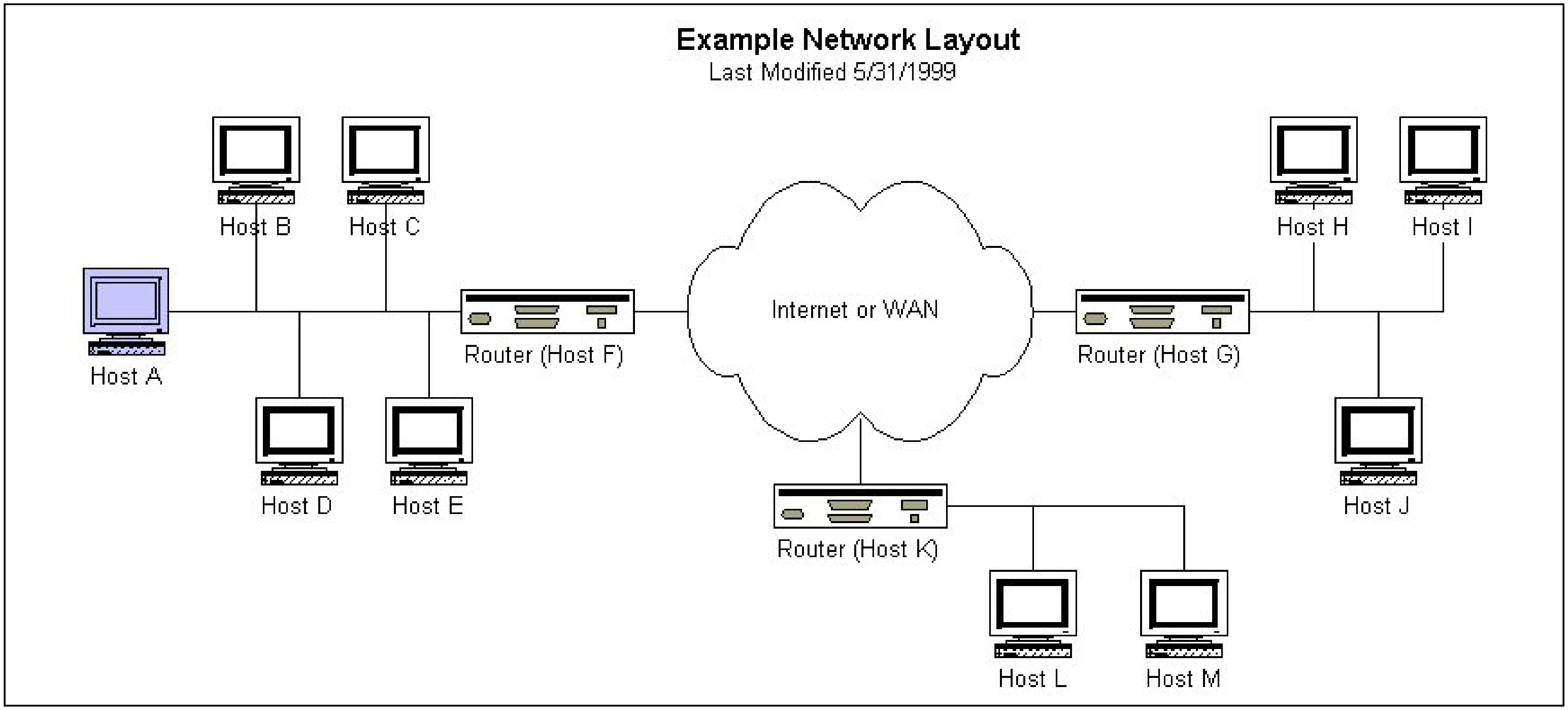
The <*parent_hosts*> option in the host definition for a "local" host should be left blank, as local hosts have no depencies or "parents" - that's why they're local.

Monitoring Local Hosts

Checking hosts that are on your local network is fairly simple. Short of someone accidentally (or intentially) unplugging the network cable from one of your hosts, there isn't too much that can go wrong as far as checking network connectivity is concerned. There are no routers or external networks between the host doing the monitoring and the other hosts on the local network.

If Nagios needs to check to see if a local host is "alive" it will simply run the host check command for that host. If the command returns an OK state, Nagios assumes the host is up. If the command returns any other status level, Nagios will assume the host is down.

Figure 1.



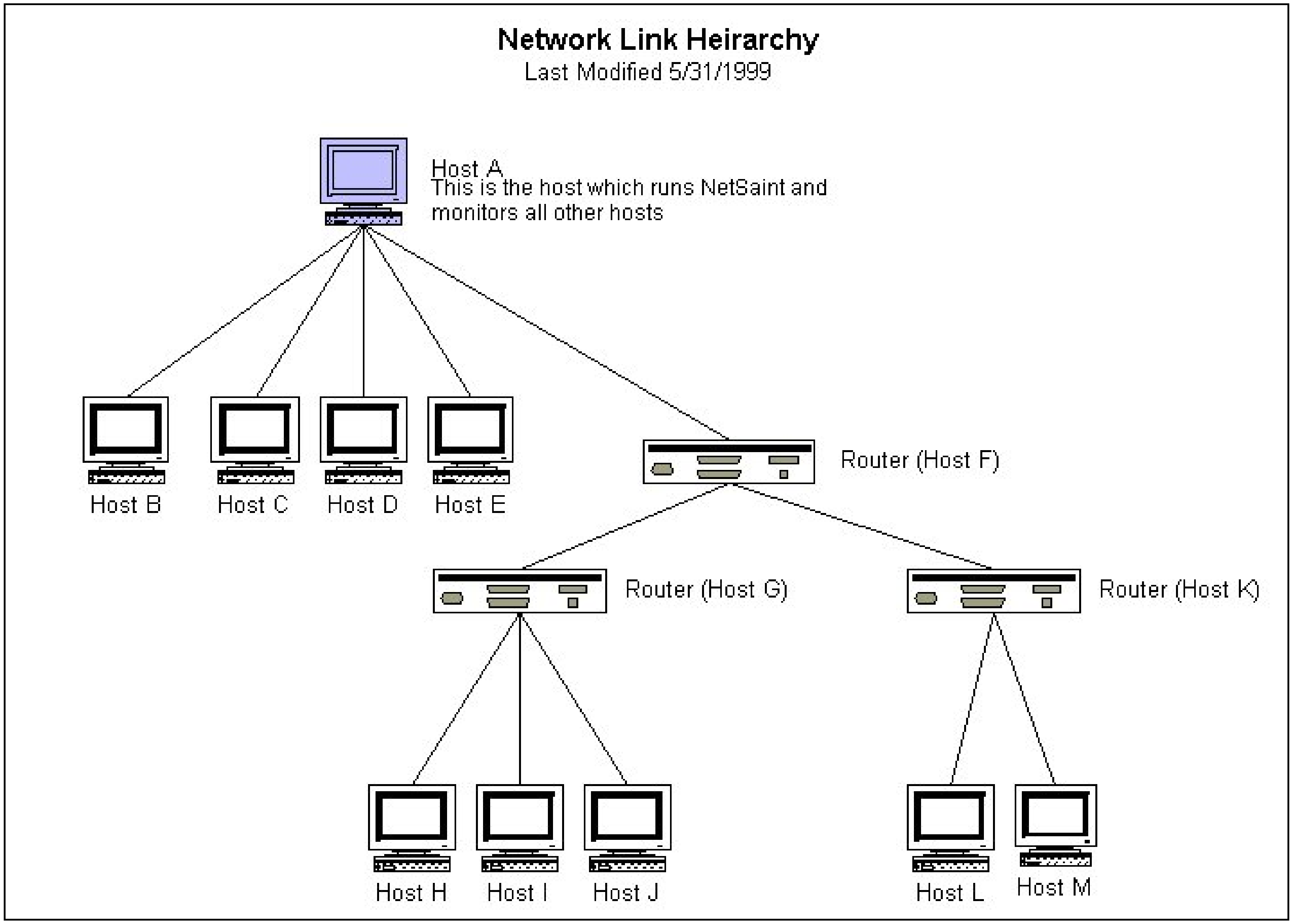
Remote Hosts

"Remote" hosts are hosts that reside on a different network segment than the host running Nagios. In the figure above, hosts G, H, I, J, K, L and M are all considered to be "remote" hosts in relation to host A.

Notice that some hosts are "farther away" than others. Hosts H, I and J are one hop further away from host A than host G (the router) is. From this observation we can construct a host dependency tree as show below in Figure 2. This tree diagram will help us in deciding how to configure each host in Nagios.

The **<parent_hosts>** option in the host definition for a "remote" host should be the short name(s) of the host(s) directly above it in the tree diagram (as show below). For example, the parent host for host H would be host G. The parent host for host G is host F. Host F has no parent host, since it is on the network segment as host A - it is a "local" host.

Figure 2.



Monitoring Remote Hosts

Checking the status of remote hosts is a bit more complicated that for local hosts. If Nagios cannot monitor services on a remote host, it needs to determine whether the remote host is down or whether it is unreachable. Luckily, the **<parent_hosts>** option allows Nagios to do this.

If a host check command for a remote host returns a non-OK state, Nagios will "walk" the depency tree (as shown in the figure above) until it reaches the top (or until a parent host check results in an OK state). By doing this, Nagios is able to determine if a service problem is the result of a down host, an down network link, or just a plain old service failure.

DOWN vs. UNREACHABLE Notification Types

I get lots of email from people asking why Nagios is sending notifications out about hosts that are unreachable. The answer is because you configured it to do that. If you want to disable UNREACHABLE notifications for hosts, modify the *notification_options* argument of your host definitions to not include the *u* (unreachable) option. More information can be found in this FAQ.

Network Outages

Introduction

The outages CGI is designed to help pinpoint the cause of network outages. For small networks this CGI may not be particularly useful, but for larger ones it will be. Pinpointing the cause of outages will help admins to more quickly find and resolve problems which are causing the biggest impact on the network.

It should be noted that the outages CGI will not attempt to find the *exact* cause of the problem, but will rather locate the hosts on your network which seem to be causing the most problems. Delving into the problem at a deeper level is left to the user, as there are any number of things which might actually be the cause of the problem.

Diagrams

The diagrams below help to show how the outages CGI goes about determining the cause of network outages. You can click on either image for a larger version...

Diagram 1

This diagram will serve as the basis for our example. All hosts shows in red are either down or unreachable (from the view of Nagios). All other hosts are up.

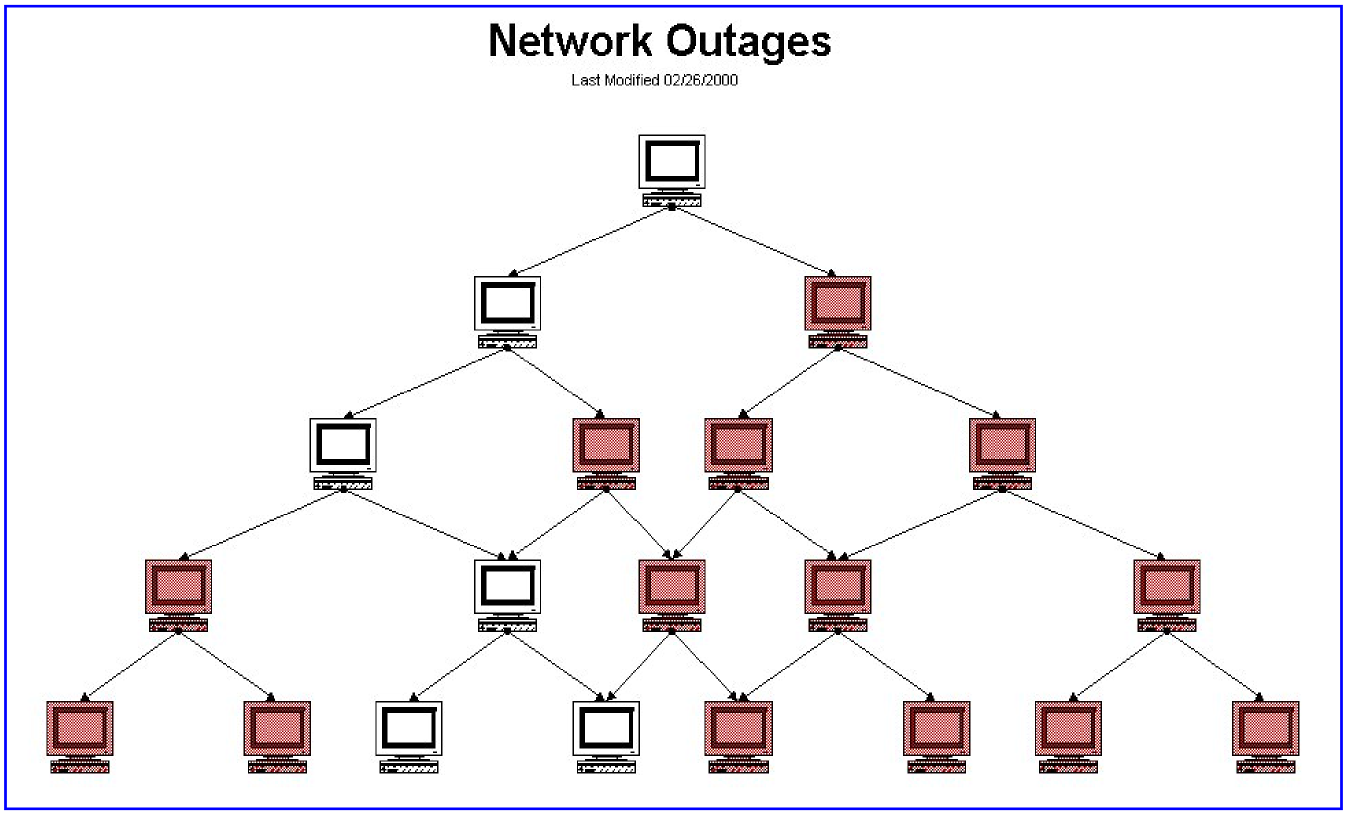
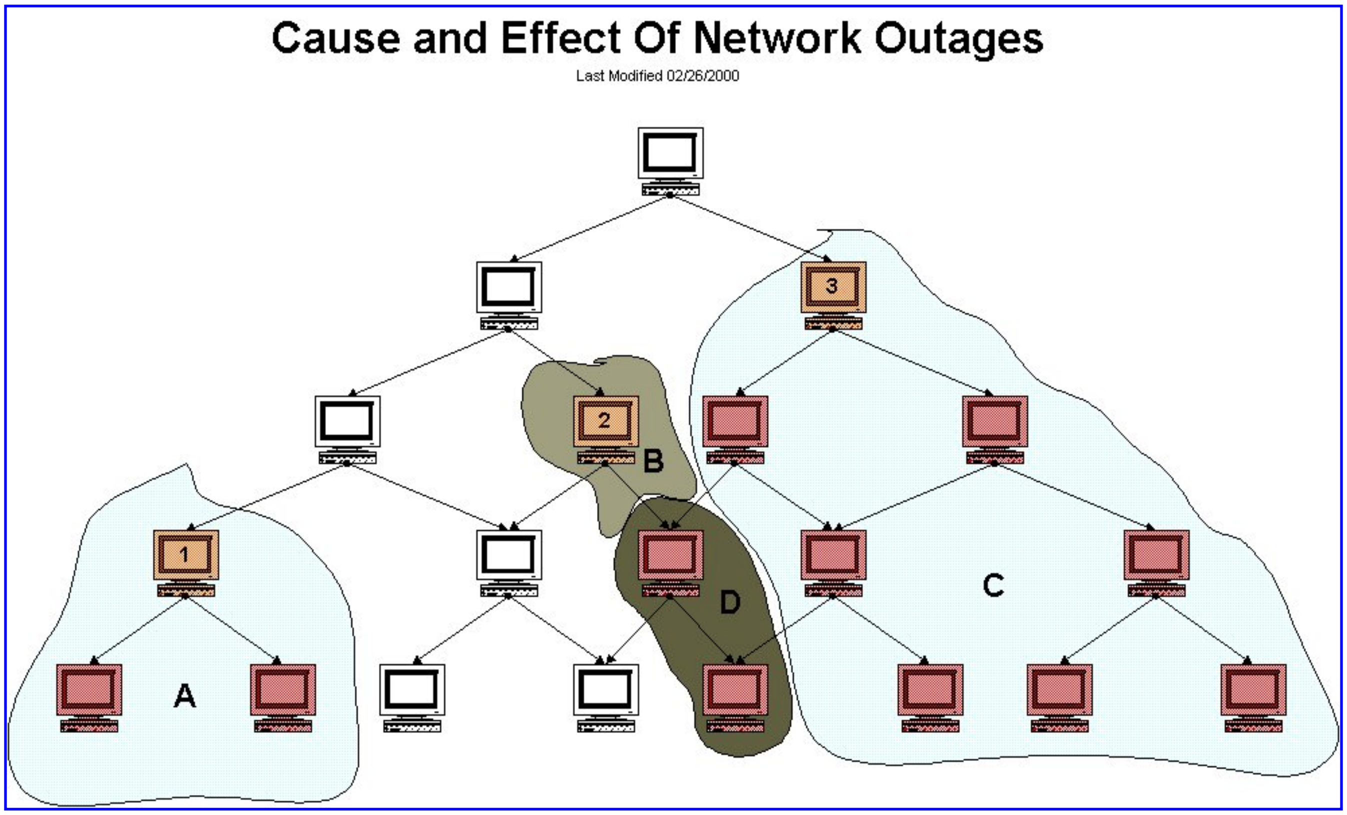


Diagram 2

This diagram pinpoints the causes of the network outages (from the view of Nagios), and shows various groups of hosts which are affected by the outages.



Determining The Cause Of Network Outages

So how does the outages CGI determine which hosts are the source of problems? *"Problem" hosts must be either in a DOWN or UNREACHABLE state* ***and*** *at least one of their immediate parent hosts must be UP*. Hosts which fit this criteria are flagged as being potential problem hosts.

In order to determine whether these flagged hosts are causing network outages, we must performs some other tests...

If *all* of the immediate child hosts of one of these flagged hosts is DOWN or UNREACHABLE *and* has no immediate parent host that is up, the flagged host is the cause of a network outage. If even one of the immediate children of a flagged host does *not* pass this test, then the flagged host is *not* the cause of a network outage.

Determining The Effects Of Network Outages

Along with telling you what hosts are causing problem on your network, the outages CGI will also tell you how many hosts and services are affected by a particular problem host. How is this determined? Take a look at diagram 2 above...

From the diagram it is clear that host 1 is blocking two child hosts (in domain A). Host 2 is solely responsbile for blocking only itself (domain B) and host 3 is solely responsibly for blocking 7 hosts (domain C). The outage effects of the two hosts in domain D are "shared" between hosts 2 and 3, since it is unclear as to which host is actually the cause of the outage. If either host 2 or 3 was UP, the these hosts might not be blocked.

The numbers of affected hosts for each problem host are as follows (the problem host is also included in these figures):

- Host 1: 3 affected hosts
- Host 2: 3 affected hosts
- Host 3: 10 affected hosts

Ranking Problems Based On Severity Level

The outages CGI will display all problem hosts, whether they are causing network outages or not. However, the CGI will tell you how many of the problem hosts (if any) are causing network outages.

In order to display the problem hosts in a somewhat useful manner, they are sorted by the severity of the effect they are having on the network. The severity level is determined by two things: The number of hosts which are affected by problem host and the number of services which are affected. Hosts hold a higher weight than services when it comes to calculating severity. The current code sets this weight ratio at 4:1 (i.e. hosts are 4 times more important than individual services).

Assuming that all hosts in diagram 2 have an equal number of services associated with them, host 3 would be ranked as the most severe problem, while hosts 1 and 2 would have the same severity level.

Notifications

Introduction

I've had a lot of questions as to exactly how notifications work. This will attempt to explain exactly when and how host and service notifications are sent out, as well as who receives them.

Index

When Do Notifications Occur?

The decision to send out notifications is made in the service check and host check logic. Host and service notifications occur in the following instances...

- When a hard state change occurs. More information on state types and hard state changes can be found here.
- When a host or service remains in a hard non-OK state and the time specified by the <*notification_interval*> option in the host or service definition has passed since the last notification was sent out (for that specified host or service). If you don't like the idea of recurring notifications, set the <*notification_interval*> value to 0 - this prevents notifications from getting sent out more than once for any given problem.

Who Gets Notified?

Each service definition has a <*contact_groups*> option that specifies what contact groups receive notifications for that particular service. Each contact group can contain one or more individual contacts. When Nagios sends out a service notification, it will notify each contact that is a member of any contact groups specified in the <*contactgroups*> option of the service definition. Nagios realizes that any given contact may be a member of more than one contact group, so it removes duplicate contact notifications before it does anything.

Each host may belong to one or more host groups. Each host group has a <*contact_groups*> option that specifies what contact groups receive notifications for hosts in that particular host group. When Nagios sends out a host notification, it will notify contacts that are members of all the contact groups that that should be notified for any and all host groups that the host is a member of. Nagios removes any duplicate contacts from the notification list before it does anything.

What Filters Must Be Passed In Order For Notifications To Be Sent?

Just because there is a need to send out a host or service notification doesn't mean that any contacts are going to get notified. There are several filters that potential notifications must pass before they are deemed worthy enough to be sent out. Even then, specific contacts may not be notified if their notification filters do not allow for the notification to be sent to them. Let's go into the filters that have to be passed in more detail...

Program-Wide Filter:

The first filter that notifications must pass is a test of whether or not notifications are enabled on a program-wide basis. This is initially determined by the enable_notifications directive in the main config file, but may be changed during runtime from the web interface. If notifications are disabled on a program-wide basis, no host or service notifications can be sent out - period. If they are enabled on a program-wide basis, there are still other tests that must be passed...

Service and Host Filters:

The first filter for host or service notifications is a check to see if the host or service is in a period of scheduled downtime. It it is in a scheduled downtime, **no one gets notified**. If it isn't in a period of downtime, it gets passed on to the next filter. As a side note, notifications for services are supressed if the host they're associated with is in a period of scheduled downtime.

The second filter for host or service notification is a check to see if the host or service is flapping (if you enabled flap detection). If the service or host is currently flapping, **no one gets notified**. Otherwise it gets passed to the next filter.

The third host or service filter that must be passed is the host- or service-specific notification options. Each service definition contains options that determine whether or not notifications can be sent out for warning states, critical states, and recoveries. Similiarly, each host definition contains options that determine whether or not notifications can be sent out when the host goes down, becomes unreachable, or recovers. If the host or service notification does not pass these options, **no one gets notified**. If it does pass these options, the notification gets passed to the next filter... Note: Notifications about host or service recoveries are only sent out if a notification was sent out for the original problem. It doesn't make sense to get a recovery notification for something you never knew was a problem.

The fourth host or service filter that must be passed is the time period test. Each host and service definition has a <*notification_period*> option that specifies which time period contains valid notification times for the host or service. If the time that the notification is being made does not fall within a valid time range in the specified time period, **no one gets contacted**. If it falls within a valid time range, the notification gets passed to the next filter... Note: If the time period filter is not passed, Nagios will reschedule the next notification for the host or service (if its in a non-OK state) for the next valid time present in the time period. This helps ensure that contacts are notified of problems as soon as possible when the next valid time in time period arrives.

The last set of host or service filters is conditional upon two things: (1) a notification was already sent out about a problem with the host or service at some point in the past and (2) the host or service has remained in the same non-OK state that it was when the last notification went out. If these two criteria are met, then Nagios will check and make sure the time that has passed since the last notification went out either meets or exceeds the value specified by the <*notification_interval*> option in the host or service definition. If not enough time has passed since the last notification, **no one gets contacted**. If either enough time has passed since the last notification or the two criteria for this filter were not met, the notification will be sent out! Whether or not it actually is sent to individual contacts is up to another set of filters...

Contact Filters:

At this point the notification has passed the program mode filter and all host or service filters and Nagios starts to notify all the people it should. Does this mean that each contact is going to receive the notification? No! Each contact has their own set of filters that the notification must pass before they receive it. Note: Contact filters are specific to each contact and do not affect whether or not other contacts receive notifications.

The first filter that must be passed for each contact are the notification options. Each contact definition contains options that determine whether or not service notifications can be sent out for warning states, critical states, and recoveries. Each contact definition also contains options that determine whether or not host notifications can be sent out when the host goes down, becomes unreachable, or recovers. If the host or service notification does not pass these options, **the contact will not be notified**. If it does pass these options, the notification gets passed to the next filter... Note: Notifications about host or service recoveries are only sent out if a notification was sent out for the original problem. It doesn't make sense to get a recovery notification for something you never knew was a problem...

The last filter that must be passed for each contact is the time period test. Each contact definition has a <*notification_period*> option that specifies which time period contains valid notification times for the contact. If the time that the notification is being made does not fall within a valid time range in the specified time period, **the contact will not be notified**. If it falls within a valid time range, the contact gets notified!

What Aren't Any Notification Methods Incorporated Directly Into Nagios?

I've gotten several questions about why notification methods (paging, etc.) are not directly incorporated into the Nagios code. The answer is simple - it just doesn't make much sense. The "core" of Nagios is not designed to be an all-in-one application. If service checks were embedded in Nagios' core it would be very difficult for users to add new check methods, modify existing checks, etc. Notifications work in a similiar manner. There are a thousand different ways to do notifications and there are already a lot of packages out there that handle the dirty work, so why re-invent the wheel and limit yourself to a bike tire? Its much easier to let an external entity (i.e. a simple script or a full-blown messaging system) do the messy stuff. Some messaging packages that can handle notifications for pagers and cellphones are listed below in the resource section.

Helpful Resources

There are many ways you could configure Nagios to send notifications out. Its up to you to decide which method(s) you want to use. Once you do that you'll have to install any necessary software and configure notification commands in your config files before you can use them. Here are just a few possible notification methods:

- Email
- Pager
- Phone (SMS)
- WinPopup message

- Yahoo, ICQ, or MSN instant message
- Audio alerts
- etc...

Basically anything you can do from a command line can be tailored for use as a notification command.

If you're interested in sending an alphanumeric notification to your pager or cellphone via email, you may be find the following information useful. Here are a few links to various messaging service providers' websites that contain information on how to send alphanumeric messages to pagers and phones...

- AT&T Wireless
- PageNet
- SprintPCS (SMS phones)

If you're looking for an alternative to using email for sending messages to your pager or cellphone, check out these packages. They could be used in conjuction with Nagios to send out a notification via a modem when a problem arises. That way you don't have to rely on email to send notifications out (remember, email may *not* work if there are network problems). I haven't actually tried these packages myself, but others have reported success using them...

- Gnokii (SMS software for contacting Nokia phones via GSM network)
- QuickPage (alphanumeric pager software)
- Sendpage (paging software)
- SMS Client (command line utility for sending messages to pagers and mobile phones)

If you want to try out a non-traditional method of notification, you might want to mess around with audio alerts. If you want to have audio alerts played on the monitoring server (with synthesized speech), check out Festival. If you'd rather leave the monitoring box alone and have audio alerts played on another box, check out the Network Audio System (NAS) and rplay projects.

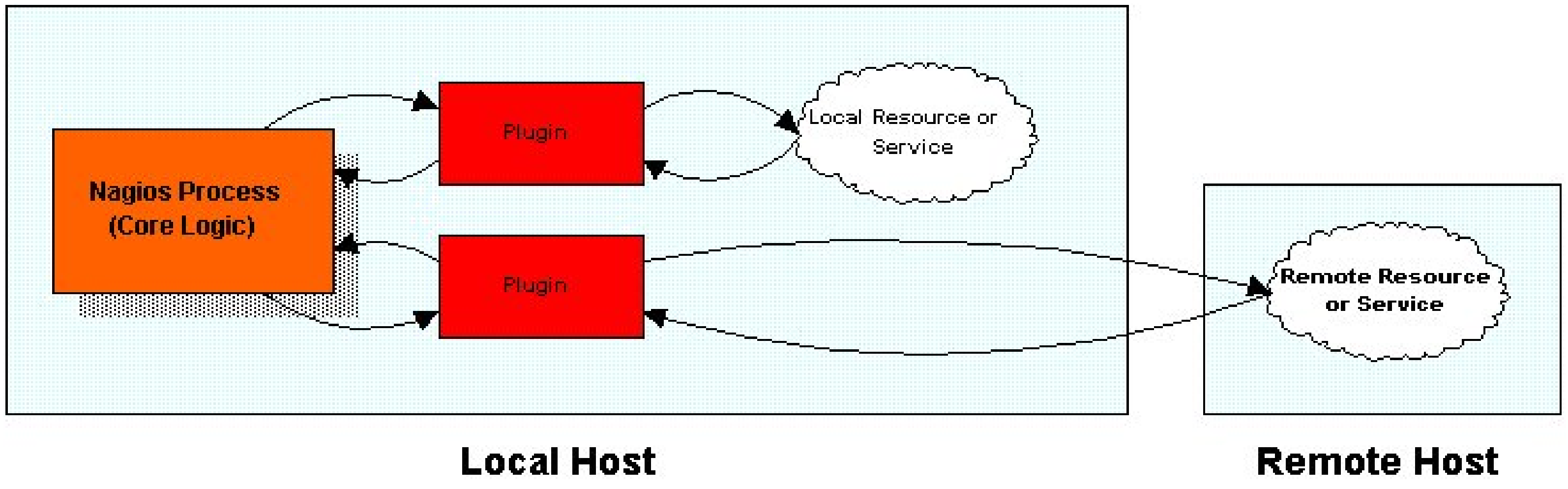
Lastly, there in an area in the contrib downloads section on the Nagios homepage for notification scripts that have been contributed by users. You might find these scripts useful, as they take care of a lot of the dirty work needed to send out alphanumeric notifications...

Plugin Theory

Introduction

Unlike many other monitoring tools, Nagios does not include any internal mechanisms for checking the status of services, hosts, etc. Instead, Nagios relies on external programs (called plugins) to do the all the dirty work. Nagios will execute a plugin whenever there is a need to check a service or host that is being monitored. The plugin does *something* (notice the very general term) to perform the check and then simply returns the results to Nagios. Nagios will process the results that it receives from the plugin and take any necessary actions (running event handlers, sending out notifications, etc).

The image below show how plugins are separated fromt the core program logic in Nagios. Nagios executes the plugins which then check local or remote resources or services of some type. When the plugins have finished checking the resource or service, they simply pass the results of the check back to Nagios for processing. A more complex diagram on how plugins work can be found in the documentation on passive service checks.


Nagios Process (Core Logic)
Plugin
Plugin
Local Resource or Service
Remote Resource or Service
Local Host
Remote Host

The Upside

The good thing about the plugin architecture is that you can monitor just about anything you can think of. If you can automate the process of checking something, you can monitor it with Nagios. There are already a lot of plugins that have been created in order to monitor basic resources such as processor load, disk usage, ping rates, etc. If you want to monitor something else, take a look at the documentation on writing plugins and roll your own. Its simple!

The Downside

The only real downside to the plugin architecture is the fact that Nagios has absolutely no idea what it is that you're monitoring. You could be monitoring network traffic statistics, data error rates, room temperate, CPU voltage, fan speed, processor load, disk space, or the ability of your super-fantastic toaster to properly brown your bread in the morning... As such, Nagios cannot produce graphs of changes to the exact values of resources you're monitoring over time. It can only track changes in the *state* of those resources. Only the plugins themselves know exactly what they're monitoring and how to perform checks. However, plugins can return optional performance data along with status information. This performance data can then be passed on to external applications which could produce graphs of service-specific information (i.e. disk space usage, processor load, etc.). More information on performance data can be found here.

Using Plugins For Service Checks

The correlation between plugins and service checks should be fairly obvious. When Nagios needs to check the status of a particular service that you have defined, it will execute the plugin you specified in the <*check_command*> argument of the service definition. The plugin will check the status of the service or resource you specify and return the results to Nagios.

Using Plugins For Host Checks

Using plugins to check the status of hosts may be a bit more difficult to understand. In each host definition you use the <*host_check_command*> argument to specify a plugin that should be executed to check the status of the host. Host checks are not performed on a regular basis - they are executed only as needed, usually when there are problems with one or more services that are associated with the host.

Host checks can use the same plugins as service checks. The only real difference is the important of the plugin results. If a plugin that is used for a host check results in a non-OK status, Nagios will believe that the host is down.

In most situations, you'll want to use a plugin which checks to see if the host can be pinged, as this is the most common method of telling whether or not a host is up. However, if you were monitoring some kind of super-fantastic toaster, you might want to use a plugin that would check to see if the heating elements turned on when the handle was pushed down. That would give a decent indication as to whether or not the toaster was "alive".

Service Check Scheduling

Index

Introduction

I've gotten a lot of questions regarding how service checks are scheduled in certain situations, along with how the scheduling differs from when the checks are actually executed and their results are processed. I'll try to go into a little more detail on how this all works...

Configuration Options

Before we begin, there are several configuration options that affect how service checks are scheduled, executed, and processed. For starters, each service definition contains three options that determine when and how each specific service check is scheduled and executed. Those three options include:

- *normal_check_interval*
- *retry_check_interval*
- *check_period*

There are also four configuration options in the main configuration file that affect service checks. These include:

- *inter_check_delay_method*
- *service_interleave_factor*
- *max_concurrent_checks*
- *service_reaper_frequency*

We'll go into more detail on how all these options affect service check scheduling as we progress. First off, let's see how services are initially scheduled when Nagios first starts or restarts...

Initial Scheduling

When Nagios (re)starts, it will attempt to schedule the initial check of all services in a manner that will minimize the load imposed on the local and remote hosts. This is done by spacing the initial service checks out, as well as interleaving them. The spacing of service checks (also known as the inter-check delay) is used to minimize/equalize the load on the local host running Nagios and the interleaving is used to minimize/equalize load imposed on remote hosts. Both the inter-check delay and interleave functions are discussed below.

Even though service checks are initially scheduled to balance the load on both the local and remote hosts, things will eventually give in to the ensuing chaos and be a bit random. Reasons for this include the fact that services are not all checked at the same interval, some services take longer to execute than others, host and/or service problems can alter the timing of one or more service checks, etc. At least we try to get things off to a good start. Hopefully the initial scheduling will keep the load on the local and remote hosts fairly balanced as time goes by...

Note: If you want to view the initial service check scheduling information, start Nagios using the **-s** command line option. Doing so will display basic scheduling information (inter-check delay, interleave factor, first and last service check time, etc) and will create a new status log that shows the exact time that all services are initially scheduled. Because this option will overwrite the status log, you should not use it when another copy of Nagios is running. Nagios does *not* start monitoring anything when this argument is used.

Inter-Check Delay

As mentioned before, Nagios attempts to equalize the load placed on the machine that is running Nagios by equally spacing out initial service checks. The spacing between consecutive service checks is called the inter-check delay. By giving a value to the inter_check_delay_method variable in the main config file, you can modify how this delay is calculated. I will discuss how the "smart" calculation works, as this is the setting you will want to use for normal operation.

When using the "smart" setting of the *inter_check_delay_method* variable, Nagios will calculate an inter-check delay value by using the following calculation:

inter-check delay = (total normal check interval for all services) / (total number of services)^2

Let's take an example. Say you have 1,000 services that each have a normal check interval of 5 minutes (obviously some services are going to be checked at different intervals, but let's look at an easy case...). The total check interal time for all services is 5,000 (1,000 * 5). That means that the average check interval for each service is 5 minutes (5,000 / 1,000). Give that information, we realize that (on average) we need to re-check 1,000 services every 5 minutes. This means that we should use an inter-check delay of 0.005 minutes (0.3 seconds) when spacing out the initial service checks. By spacing each service check out by 0.3 seconds, we can somewhat guarantee that Nagios is scheduling and/or executing 3 new service checks every second. By spacing the checks out evenly over time like this, we can hope that the load on the local server that is running Nagios remains somewhat balanced.

Service Interleaving

As discussed above, the inter-check delay helps to equalize the load that Nagios imposes on the local host. What about remote hosts? Is it necessary to equalize load on remote hosts? Why? Yes, it is important and yes, Nagios can help out with this. Equalizing load on remote hosts is especially important with the advent of service check parallelization. If you monitor a large number of services on a remote host and the checks were not spread out, the remote host might think that it was the victim of a SYN attack if there were a lot of open connections on the same port. Plus, attempting to equalize the load on hosts is just a nice thing to do...

By giving a value to the service_interleave_factor variable in the main config file, you can modify how the interleave factor is calculated. I will discuss how the "smart" calculation works, as this will probably be the setting you will want to use for normal operation. You can, however, use a pre-set interleave factor instead of having Nagios calculate one for you. Also of note, if you use an interleave factor of 1, service check interleaving is basically disabled.

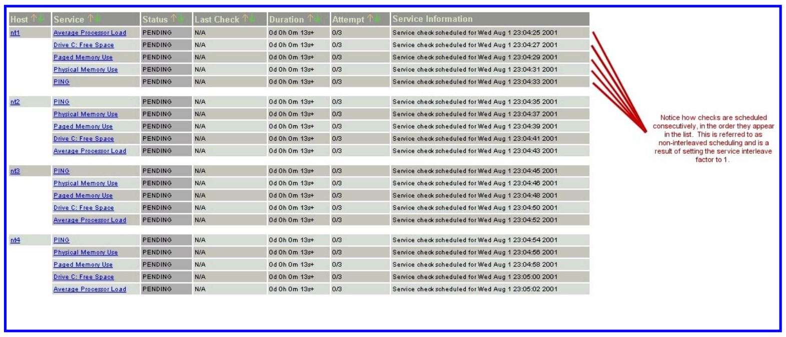
When using the "smart" setting of the *service_interleave_factor* variable, Nagios will calculate an interleave factor by using the following calculation:

interleave factor = ceil (total number of services / total number of hosts)

Let's take an example. Say you have a total of 1,000 services and 150 hosts that you monitor. Nagios would calculate the interleave factor to be 7. This means that when Nagios schedules initial service checks it will schedule the first one it finds, skip the next 6, schedule the next one, and so on... This process will keep repeating until all service checks have been scheduled. Since services are sorted (and thus scheduled) by the name of the host they are associated with, this will help with minimizing/equalizing the load placed upon remote hosts.

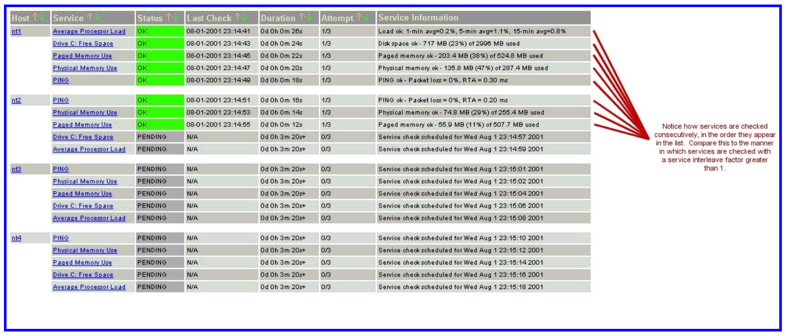
The images below depict how service checks are scheduled when they are not interleaved (*service_interleave_factor*=1) and when they are interleaved with the *service_interleave_factor* variable equal to 4.

Non-Interleaved Checks:



Host	Service	Status	Last Check	Duration	Attempt	Service Information
nt1	Average Processor Load	PENDING	N/A	0d 0h 0m 13s+	0/3	Service check scheduled for Wed Aug 1 23:04:25 2001
	Drive C: Free Space	PENDING	N/A	0d 0h 0m 13s+	0/3	Service check scheduled for Wed Aug 1 23:04:27 2001
	Paged Memory Use	PENDING	N/A	0d 0h 0m 13s+	0/3	Service check scheduled for Wed Aug 1 23:04:29 2001
	Physical Memory Use	PENDING	N/A	0d 0h 0m 13s+	0/3	Service check scheduled for Wed Aug 1 23:04:31 2001
	PING	PENDING	N/A	0d 0h 0m 13s+	0/3	Service check scheduled for Wed Aug 1 23:04:33 2001
nt2	PING	PENDING	N/A	0d 0h 0m 13s+	0/3	Service check scheduled for Wed Aug 1 23:04:35 2001
	Physical Memory Use	PENDING	N/A	0d 0h 0m 13s+	0/3	Service check scheduled for Wed Aug 1 23:04:37 2001
	Paged Memory Use	PENDING	N/A	0d 0h 0m 13s+	0/3	Service check scheduled for Wed Aug 1 23:04:39 2001
	Drive C: Free Space	PENDING	N/A	0d 0h 0m 13s+	0/3	Service check scheduled for Wed Aug 1 23:04:41 2001
	Average Processor Load	PENDING	N/A	0d 0h 0m 13s+	0/3	Service check scheduled for Wed Aug 1 23:04:43 2001
nt3	PING	PENDING	N/A	0d 0h 0m 13s+	0/3	Service check scheduled for Wed Aug 1 23:04:45 2001
	Physical Memory Use	PENDING	N/A	0d 0h 0m 13s+	0/3	Service check scheduled for Wed Aug 1 23:04:46 2001
	Paged Memory Use	PENDING	N/A	0d 0h 0m 13s+	0/3	Service check scheduled for Wed Aug 1 23:04:48 2001
	Drive C: Free Space	PENDING	N/A	0d 0h 0m 13s+	0/3	Service check scheduled for Wed Aug 1 23:04:50 2001
	Average Processor Load	PENDING	N/A	0d 0h 0m 13s+	0/3	Service check scheduled for Wed Aug 1 23:04:52 2001
nt4	PING	PENDING	N/A	0d 0h 0m 13s+	0/3	Service check scheduled for Wed Aug 1 23:04:54 2001
	Physical Memory Use	PENDING	N/A	0d 0h 0m 13s+	0/3	Service check scheduled for Wed Aug 1 23:04:56 2001
	Paged Memory Use	PENDING	N/A	0d 0h 0m 13s+	0/3	Service check scheduled for Wed Aug 1 23:04:58 2001
	Drive C: Free Space	PENDING	N/A	0d 0h 0m 13s+	0/3	Service check scheduled for Wed Aug 1 23:05:00 2001
	Average Processor Load	PENDING	N/A	0d 0h 0m 13s+	0/3	Service check scheduled for Wed Aug 1 23:05:02 2001

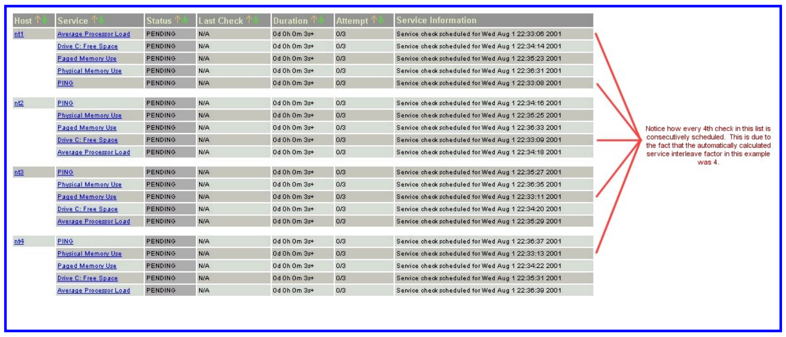
Notice how checks are scheduled consecutively, in the order they appear in the list. This is referred to as non-interleaved scheduling and is a result of setting the service interleave factor to 1.



Host	Service	Status	Last Check	Duration	Attempt	Service Information
nt1	Average Processor Load	OK	08-01-2001 23:14:41	0d 0h 0m 26s	1/3	Load ok: 1-min avg=0.2%, 5-min avg=1.1%, 15-min avg=0.8%
	Drive C: Free Space	OK	08-01-2001 23:14:43	0d 0h 0m 24s	1/3	Disk space ok - 717 MB (23%) of 2996 MB used
	Paged Memory Use	OK	08-01-2001 23:14:45	0d 0h 0m 22s	1/3	Paged memory ok - 203.4 MB (38%) of 524.8 MB used
	Physical Memory Use	OK	08-01-2001 23:14:47	0d 0h 0m 20s	1/3	Physical memory ok - 135.8 MB (47%) of 287.4 MB used
	PING	OK	08-01-2001 23:14:49	0d 0h 0m 18s	1/3	PING ok - Packet loss = 0%, RTA = 0.30 ms
nt2	PING	OK	08-01-2001 23:14:51	0d 0h 0m 16s	1/3	PING ok - Packet loss = 0%, RTA = 0.20 ms
	Physical Memory Use	OK	08-01-2001 23:14:53	0d 0h 0m 14s	1/3	Physical memory ok - 74.8 MB (29%) of 255.4 MB used
	Paged Memory Use	OK	08-01-2001 23:14:55	0d 0h 0m 12s	1/3	Paged memory ok - 55.9 MB (11%) of 507.7 MB used
	Drive C: Free Space	PENDING	N/A	0d 0h 3m 20s+	0/3	Service check scheduled for Wed Aug 1 23:14:57 2001
	Average Processor Load	PENDING	N/A	0d 0h 3m 20s+	0/3	Service check scheduled for Wed Aug 1 23:14:59 2001
nt3	PING	PENDING	N/A	0d 0h 3m 20s+	0/3	Service check scheduled for Wed Aug 1 23:15:01 2001
	Physical Memory Use	PENDING	N/A	0d 0h 3m 20s+	0/3	Service check scheduled for Wed Aug 1 23:15:02 2001
	Paged Memory Use	PENDING	N/A	0d 0h 3m 20s+	0/3	Service check scheduled for Wed Aug 1 23:15:04 2001
	Drive C: Free Space	PENDING	N/A	0d 0h 3m 20s+	0/3	Service check scheduled for Wed Aug 1 23:15:06 2001
	Average Processor Load	PENDING	N/A	0d 0h 3m 20s+	0/3	Service check scheduled for Wed Aug 1 23:15:08 2001
nt4	PING	PENDING	N/A	0d 0h 3m 20s+	0/3	Service check scheduled for Wed Aug 1 23:15:10 2001
	Physical Memory Use	PENDING	N/A	0d 0h 3m 20s+	0/3	Service check scheduled for Wed Aug 1 23:15:12 2001
	Paged Memory Use	PENDING	N/A	0d 0h 3m 20s+	0/3	Service check scheduled for Wed Aug 1 23:15:14 2001
	Drive C: Free Space	PENDING	N/A	0d 0h 3m 20s+	0/3	Service check scheduled for Wed Aug 1 23:15:16 2001
	Average Processor Load	PENDING	N/A	0d 0h 3m 20s+	0/3	Service check scheduled for Wed Aug 1 23:15:18 2001

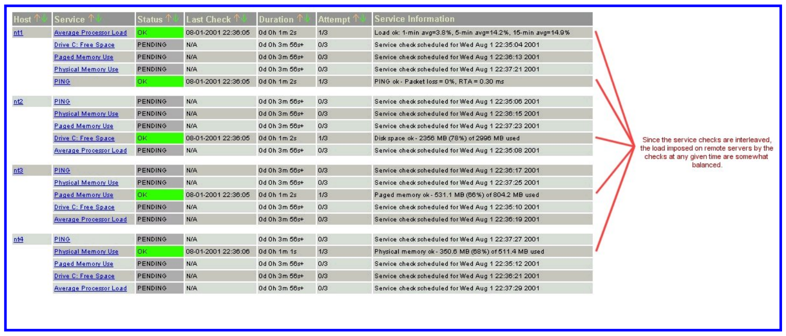
Notice how services are checked consecutively, in the order they appear in the list. Compare this to the manner in which services are checked with a service interleave factor greater than 1.

Interleaved Checks:



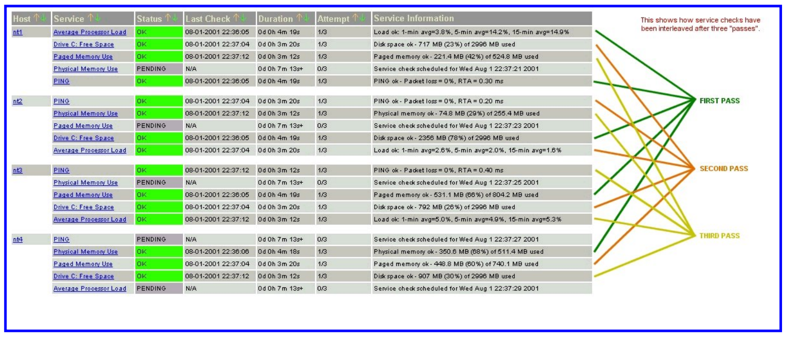
Host	Service	Status	Last Check	Duration	Attempt	Service Information
nt1	Average Processor Load	PENDING	N/A	0d 0h 0m 3s+	0/3	Service check scheduled for Wed Aug 1 22:33:06 2001
	Drive C: Free Space	PENDING	N/A	0d 0h 0m 3s+	0/3	Service check scheduled for Wed Aug 1 22:34:14 2001
	Paged Memory Use	PENDING	N/A	0d 0h 0m 3s+	0/3	Service check scheduled for Wed Aug 1 22:35:23 2001
	Physical Memory Use	PENDING	N/A	0d 0h 0m 3s+	0/3	Service check scheduled for Wed Aug 1 22:36:31 2001
	PING	PENDING	N/A	0d 0h 0m 3s+	0/3	Service check scheduled for Wed Aug 1 22:33:08 2001
nt2	PING	PENDING	N/A	0d 0h 0m 3s+	0/3	Service check scheduled for Wed Aug 1 22:34:16 2001
	Physical Memory Use	PENDING	N/A	0d 0h 0m 3s+	0/3	Service check scheduled for Wed Aug 1 22:35:25 2001
	Paged Memory Use	PENDING	N/A	0d 0h 0m 3s+	0/3	Service check scheduled for Wed Aug 1 22:36:33 2001
	Drive C: Free Space	PENDING	N/A	0d 0h 0m 3s+	0/3	Service check scheduled for Wed Aug 1 22:33:09 2001
	Average Processor Load	PENDING	N/A	0d 0h 0m 3s+	0/3	Service check scheduled for Wed Aug 1 22:34:18 2001
nt3	PING	PENDING	N/A	0d 0h 0m 3s+	0/3	Service check scheduled for Wed Aug 1 22:35:27 2001
	Physical Memory Use	PENDING	N/A	0d 0h 0m 3s+	0/3	Service check scheduled for Wed Aug 1 22:36:35 2001
	Paged Memory Use	PENDING	N/A	0d 0h 0m 3s+	0/3	Service check scheduled for Wed Aug 1 22:33:11 2001
	Drive C: Free Space	PENDING	N/A	0d 0h 0m 3s+	0/3	Service check scheduled for Wed Aug 1 22:34:20 2001
	Average Processor Load	PENDING	N/A	0d 0h 0m 3s+	0/3	Service check scheduled for Wed Aug 1 22:35:29 2001
nt4	PING	PENDING	N/A	0d 0h 0m 3s+	0/3	Service check scheduled for Wed Aug 1 22:36:37 2001
	Physical Memory Use	PENDING	N/A	0d 0h 0m 3s+	0/3	Service check scheduled for Wed Aug 1 22:33:13 2001
	Paged Memory Use	PENDING	N/A	0d 0h 0m 3s+	0/3	Service check scheduled for Wed Aug 1 22:34:22 2001
	Drive C: Free Space	PENDING	N/A	0d 0h 0m 3s+	0/3	Service check scheduled for Wed Aug 1 22:35:31 2001
	Average Processor Load	PENDING	N/A	0d 0h 0m 3s+	0/3	Service check scheduled for Wed Aug 1 22:36:39 2001

Notice how every 4th check in this list is consecutively scheduled. This is due to the fact that the automatically calculated service interleave factor in this example was 4.



Host	Service	Status	Last Check	Duration	Attempt	Service Information
nt1	Average Processor Load	OK	08-01-2001 22:36:05	0d 0h 1m 2s	1/3	Load ok: 1-min avg=3.8%, 5-min avg=14.2%, 15-min avg=14.9%
	Drive C: Free Space	PENDING	N/A	0d 0h 3m 56s+	0/3	Service check scheduled for Wed Aug 1 22:35:04 2001
	Paged Memory Use	PENDING	N/A	0d 0h 3m 56s+	0/3	Service check scheduled for Wed Aug 1 22:36:13 2001
	Physical Memory Use	PENDING	N/A	0d 0h 3m 56s+	0/3	Service check scheduled for Wed Aug 1 22:37:21 2001
	PING	OK	08-01-2001 22:36:05	0d 0h 1m 2s	1/3	PING ok - Packet loss = 0%, RTA = 0.30 ms
nt2	PING	PENDING	N/A	0d 0h 3m 56s+	0/3	Service check scheduled for Wed Aug 1 22:35:06 2001
	Physical Memory Use	PENDING	N/A	0d 0h 3m 56s+	0/3	Service check scheduled for Wed Aug 1 22:36:15 2001
	Paged Memory Use	PENDING	N/A	0d 0h 3m 56s+	0/3	Service check scheduled for Wed Aug 1 22:37:23 2001
	Drive C: Free Space	OK	08-01-2001 22:36:05	0d 0h 1m 2s	1/3	Disk space ok - 2356 MB (78%) of 2996 MB used
	Average Processor Load	PENDING	N/A	0d 0h 3m 56s+	0/3	Service check scheduled for Wed Aug 1 22:35:08 2001
nt3	PING	PENDING	N/A	0d 0h 3m 56s+	0/3	Service check scheduled for Wed Aug 1 22:36:17 2001
	Physical Memory Use	PENDING	N/A	0d 0h 3m 56s+	0/3	Service check scheduled for Wed Aug 1 22:37:25 2001
	Paged Memory Use	OK	08-01-2001 22:36:05	0d 0h 1m 2s	1/3	Paged memory ok - 531.1 MB (66%) of 804.2 MB used
	Drive C: Free Space	PENDING	N/A	0d 0h 3m 56s+	0/3	Service check scheduled for Wed Aug 1 22:35:10 2001
	Average Processor Load	PENDING	N/A	0d 0h 3m 56s+	0/3	Service check scheduled for Wed Aug 1 22:36:19 2001
nt4	PING	PENDING	N/A	0d 0h 3m 56s+	0/3	Service check scheduled for Wed Aug 1 22:37:27 2001
	Physical Memory Use	OK	08-01-2001 22:36:06	0d 0h 1m 1s	1/3	Physical memory ok - 350.6 MB (68%) of 511.4 MB used
	Paged Memory Use	PENDING	N/A	0d 0h 3m 56s+	0/3	Service check scheduled for Wed Aug 1 22:35:12 2001
	Drive C: Free Space	PENDING	N/A	0d 0h 3m 56s+	0/3	Service check scheduled for Wed Aug 1 22:36:21 2001
	Average Processor Load	PENDING	N/A	0d 0h 3m 56s+	0/3	Service check scheduled for Wed Aug 1 22:37:29 2001

Since the service checks are interleaved, the load imposed on remote servers by the checks at any given time are somewhat balanced.



Host	Service	Status	Last Check	Duration	Attempt	Service Information
nt1	Average Processor Load	OK	08-01-2001 22:36:05	0d 0h 4m 19s	1/3	Load ok: 1-min avg=3.8%, 5-min avg=14.2%, 15-min avg=14.9%
	Drive C: Free Space	OK	08-01-2001 22:37:04	0d 0h 3m 20s	1/3	Disk space ok - 717 MB (23%) of 2996 MB used
	Paged Memory Use	OK	08-01-2001 22:37:12	0d 0h 3m 12s	1/3	Paged memory ok - 221.4 MB (42%) of 524.8 MB used
	Physical Memory Use	PENDING	N/A	0d 0h 7m 13s+	0/3	Service check scheduled for Wed Aug 1 22:37:21 2001
	PING	OK	08-01-2001 22:36:05	0d 0h 4m 19s	1/3	PING ok - Packet loss = 0%, RTA = 0.30 ms
nt2	PING	OK	08-01-2001 22:37:04	0d 0h 3m 20s	1/3	PING ok - Packet loss = 0%, RTA = 0.20 ms
	Physical Memory Use	OK	08-01-2001 22:37:12	0d 0h 3m 12s	1/3	Physical memory ok - 74.8 MB (29%) of 255.4 MB used
	Paged Memory Use	PENDING	N/A	0d 0h 7m 13s+	0/3	Service check scheduled for Wed Aug 1 22:37:23 2001
	Drive C: Free Space	OK	08-01-2001 22:36:05	0d 0h 4m 19s	1/3	Disk space ok - 2356 MB (78%) of 2996 MB used
	Average Processor Load	OK	08-01-2001 22:37:04	0d 0h 3m 20s	1/3	Load ok: 1-min avg=2.6%, 5-min avg=2.0%, 15-min avg=1.6%
nt3	PING	OK	08-01-2001 22:37:12	0d 0h 3m 12s	1/3	PING ok - Packet loss = 0%, RTA = 0.40 ms
	Physical Memory Use	PENDING	N/A	0d 0h 7m 13s+	0/3	Service check scheduled for Wed Aug 1 22:37:25 2001
	Paged Memory Use	OK	08-01-2001 22:36:05	0d 0h 4m 19s	1/3	Paged memory ok - 531.1 MB (66%) of 804.2 MB used
	Drive C: Free Space	OK	08-01-2001 22:37:04	0d 0h 3m 20s	1/3	Disk space ok - 792 MB (26%) of 2996 MB used
	Average Processor Load	OK	08-01-2001 22:37:12	0d 0h 3m 12s	1/3	Load ok: 1-min avg=5.0%, 5-min avg=4.9%, 15-min avg=5.3%
nt4	PING	PENDING	N/A	0d 0h 7m 13s+	0/3	Service check scheduled for Wed Aug 1 22:37:27 2001
	Physical Memory Use	OK	08-01-2001 22:36:06	0d 0h 4m 18s	1/3	Physical memory ok - 350.6 MB (68%) of 511.4 MB used
	Paged Memory Use	OK	08-01-2001 22:37:04	0d 0h 3m 20s	1/3	Paged memory ok - 448.8 MB (60%) of 740.1 MB used
	Drive C: Free Space	OK	08-01-2001 22:37:12	0d 0h 3m 12s	1/3	Disk space ok - 907 MB (30%) of 2996 MB used
	Average Processor Load	PENDING	N/A	0d 0h 7m 13s+	0/3	Service check scheduled for Wed Aug 1 22:37:29 2001

This shows how service checks have been interleaved after three "passes".

Maximum Concurrent Service Checks

In order to prevent Nagios from consuming all of your CPU resources, you can restrict the maximum number of concurrent service checks that can be running at any given time. This is controlled by using the max_concurrent_checks option in the main config file.

The good thing about this setting is that you can regulate Nagios' CPU usage. The down side is that service checks may fall behind if this value is set too low. When it comes time to execute a service check, Nagios will make sure that no more than *x* service checks are either being executed or waiting to have their results processed (where *x* is the number of checks you specified for the *max_concurrent_checks* option). If that limit has been reached, Nagios will postpone the execution of any pending checks until some of the previous checks have completed. So how does one determine a reasonable value for the *max_concurrent_checks* option?

First off, you need to know the following things...

- The inter-check delay that Nagios uses to initially schedule service checks (use the **-s** command line argument to check this)
- The frequency (in seconds) of service reaper events, as specified by the service_reaper_frequency variable in the main config file.
- A general idea of the average time that service checks actually take to execute (most plugins timeout after 10 seconds, so the average is probably going to be lower)

Next, use the following calculation to determine a reasonable value for the maximum number of concurrent checks that are allowed...

max. concurrent checks = ceil(max(service reaper frequency , average check execution time) / inter-check delay)

The calculated number should provide a reasonable starting point for the *max_concurrent_checks* variable. You may have to increase this value a bit if service checks are still falling behind schedule or decrease it if Nagios is hogging too much CPU time.

Let's say you are monitoring 875 services, each with an average check interval of 2 minutes. That means that your inter-check delay is going to be 0.137 seconds. If you set the service reaper frequency to be 10 seconds, you can calculate a rough value for the max. number of concurrent checks as follows (I'll assume that the average execution time for service checks is less than 10 seconds) ...

max. concurrent checks = ceil(10 / 0.137)

In this case, the calculated value is going to be 73. This makes sense because (on average) Nagios are going to be executing just over 7 new service checks per second and it only processes service check results every 10 seconds. That means at given time there will be a just over 70 service checks that are either being executed or waiting to have their results processed. In this case, I would probably recommend bumping the max. concurrent checks value up to 80, since there will be delays when Nagios processes service check results and does its other work. Obviously, you're going to have test and tweak things a bit to get everything running smoothly on your system, but hopefully this provided some general guidelines...

Time Restraints

The *check_period* option determines the time period during which Nagios can run checks of the service. Regardless of what status a particular service is in, if the time that it is actually executed is not a vaid time within the time period that has been specified, the check will *not* be executed. Instead, Nagios will reschedule the service check for the next valid time in the time period. If the check can be run (e.g. the time is valid within the time period), the service check is executed.

Note: Even though a service check may not be able to be executed at a given time, Nagios may still *schedule* it to be run at that time. This is most likely to happen during the initial scheduling of services, although it may happen in other instances as well. This does *not* mean that Nagios will execute the check! When it comes time to actually *execute* a service check, Nagios will verify that the check can be run at the current time. If it cannot, Nagios will not execute the service check, but will instead just reschedule it for a later time. Don't let this one throw you confuse you! The scheduling and execution of service checks are two distinctly different (although related) things.

Normal Scheduling

In an ideal world you wouldn't have network problems. But if that were the case, you wouldn't need a network monitoring tool. Anyway, when things are running smoothly and a service is in an OK state, we'll call that "normal". Service checks are normally scheduled at the frequency specified by the *check_interval* option. That's it. Simple, huh?

Scheduling During Problems

So what happens when there are problems with a service? Well, one of the things that happens is the service check scheduling changes. If you've configured the *max_attempts* option of the service definition to be something greater than 1, Nagios will recheck the service before deciding that a real problem exists. While the service is being rechecked (up to *max_attempts* times) it is considered to be in a "soft" state (as described here) and the service checks are rescheduled at a frequency determined by the *retry_interval* option.

If Nagios rechecks the service *max_attempts* times and it is still in a non-OK state, Nagios will put the service into a "hard" state, send out notifications to contacts (if applicable), and start rescheduling future checks of the service at a frequency determined by the *check_interval* option.

As always, there are exceptions to the rules. When a service check results in a non-OK state, Nagios will check the host that the service is associated with to determine whether or not is up (see the note below for info on how this is done). If the host is not up (i.e. it is either down or unreachable), Nagios will immediately put the service into a hard non-OK state and it will reset the current attempt number to 1. Since the service is in a hard non-OK state, the service check will be rescheduled at the normal frequency specified by the *check_interval* option instead of the *retry_interval* option.

Host Checks

Unlike service checks, host checks are *not* scheduled on a regular basis. Instead they are run on demand, as Nagios sees a need. This is a common question asked by users, so it needs to be clarified.

One instance where Nagios checks the status of a host is when a service check results in a non-OK status. Nagios checks the host to decide whether or not the host is up, down, or unreachable. If the first host check returns a non-OK state, Nagios will keep pounding out checks of the host until either (a) the maximum number of host checks (specified by the *max_attempts* option in the host definition) is reached or (b) a host check results in an OK state.

Also of note - when Nagios is check the status of a host, it holds off on doing anything else (executing new service checks, processing other service check results, etc). This can slow things down a bit and cause pending service checks to be delayed for a while, but it is necessary to determine the status of the host before Nagios can take any further action on the service(s) that are having problems.

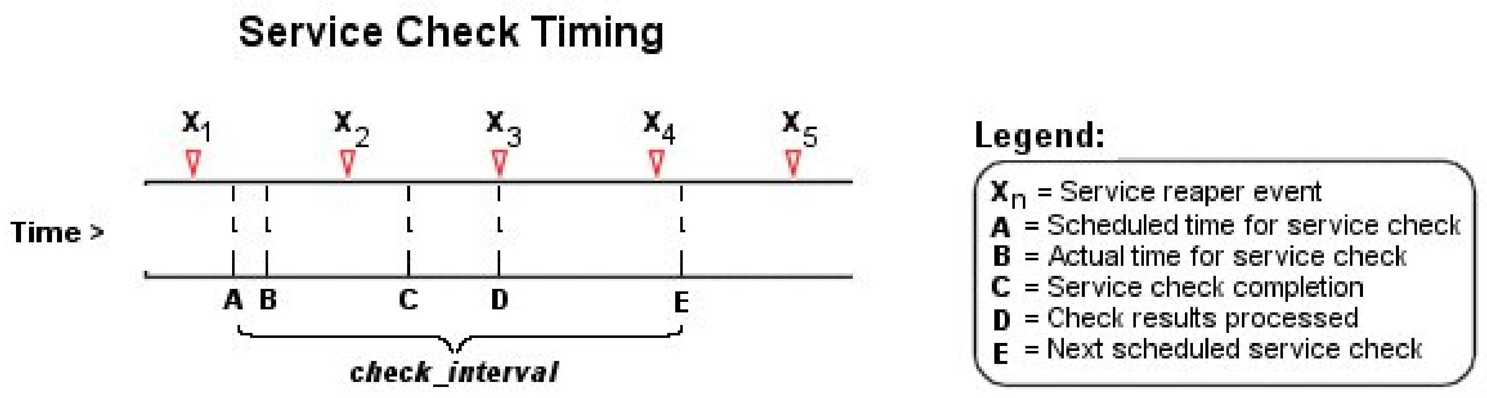
Scheduling Delays

It should be noted that service check scheduling and execution is done on a best effort basis. Individual service checks are considered to be low priority events in Nagios, so they can get delayed if high priority events need to be executed. Examples of high priority events include log file rotations, external command checks, and service reaper events. Additionally, host checks will slow down the execution and processing of service checks.

Scheduling Example

The scheduling of service checks, their execution, and the processing of their results can be a bit difficult to understand, so let's look at a simple example. Look at the diagram below - I'll refer to it as I explain how things are done.

Image 5.


Service Check Timing
X1
X2
X3
X4
X5
Time >
A
B
C
D
E
check_interval
Legend:
Xn = Service reaper event
A = Scheduled time for service check
B = Actual time for service check
C = Service check completion
D = Check results processed
E = Next scheduled service check

First off, the **X**$_n$ events are service reaper events that are scheduled at a frequency specified by the service_reaper_frequency option in the main config file. Service reaper events do the work of gathering and processing service check results. They serve as the core logic for Nagios, kicking off host checks, event handlers and notifications as necessary.

For the example here, a service has been scheduled to be executed at time **A**. However, Nagios got behind in its event queue, so the check was not actually executed until time **B**. The service check finished executing at time **C**, so the difference between points **C** and **B** is the actual amount of time that the check was running.

The results of the service check are not processed immediately after the check is done executing. Instead, the results are saved for later processing by a service reaper event. The next service reaper event occurs at time **D**, so that is approximately the time that the results are processed (the actual time may be later than **D** since other service check results may be processed before this one).

At the time that the service reaper event processes the service check results, it will reschedule the next service check and place it into Nagios' event queue. We'll assume that the service check resulted in an OK status, so the next check at time **E** is scheduled after the originally scheduled check time by a length of time specified by the *check_interval* option. Note that the service is *not* rescheduled based off the time that it was actually executed! There is one exception to this (isn't there always?) - if the time that the service check is actually executed (point **B**) occurs after the next service check time (point **E**), Nagios will compensate by adjusting the next check time. This is done to ensure that Nagios doesn't go nuts trying to keep up with service checks if it comes under heavy load. Besides, what's the point of scheduling something in the past...?

Service Definition Options That Affect Scheduling

Each service definition contains a *normal_check_interval* and *retry_check_interval* option. Hopefully this will clarify what these two options do, how they relate to the *max_check_attempts* option in the service definition, and how they affect the scheduling of the service.

First off, the *normal_check_interval* option is the interval at which the service is checked under "normal" circumstances. "Normal" circumstances mean whenever the service is in an OK state or when its in a hard non-OK state.

When a service first changes from an OK state to a non-OK state, Nagios gives you the ability to temporarily slow down or speed up the interval at which subsequent checks of that service will occur. When the service first changes state, Nagios will perform up to *max_check_attempts*-1 retries of the service check before it decides its a real problem. While the service is being retried, it is scheduled according to the *retry_check_interval* option, which might be faster or slower than the normal *normal_check_interval* option. While the service is being rechecked (up to *max_check_attempts*-1 times), the service is in a soft state. If the service is rechecked *max_check_attempts*-1 times and it is still in a non-OK state, the service turns into a hard state and is subsequently rescheduled at the normal rate specified by the *check_interval* option.

On a side note, it you specify a value of 1 for the *max_check_attempts* option, the service will not ever be checked at the interval specified by the *retry_check_interval* option. Instead, it immediately turns into a hard state and is subsequently rescheduled at the rate specified by the *normal_check_interval* option.

State Types

Introduction

The current state of services and hosts is determined by two components: the status of the service or host (i.e. OK, WARNING, UP, DOWN, etc.) and the *type* of state it is in. There are two state types in Nagios - "soft" states and "hard" states. State types are a crucial part of Nagios' monitoring logic. They are used to determine when event handlers are executed and when notifications are sent out.

Service and Host Check Retries

In order to prevent false alarms, Nagios allows you to define how many times a service or host check will be retried before the service or host is considered to have a real problem. The maximum number of retries before a service or host check is considered to have a real problem is controlled by the <*max_check)attempts*> option in the service and host definitions, respectively. Depending on what attempt a service or host check is currently on determines what type of state it is is. There are a few exceptions to this in the service monitoring logic, but we'll ignore those for now. Let's take a look at the different service state types...

Soft States

Soft states occur for services and hosts in the following situations...

- When a service or host check results in a non-OK state and it has not yet been (re)checked the number of times specified by the <*max_check_attempts*> option in the service or host definition. Let's call this a soft error state...
- When a service or host recovers from a soft error state. This is considered to be a soft recovery.

Soft State Events

What happens when a service or host is in a soft error state or experiences a soft recovery?

- The soft error or recovery is logged if you enabled the log_service_retries or log_host_retries options in the main configuration file.
- Event handlers are executed (if you defined any) to handle the soft error or recovery for the service or host. (Before any event handler is executed, the **$STATETYPE$** macro is set to "**SOFT**").
- Nagios does *not* send out notifications to any contacts because there is (or was) no "real" problem with the service or host.

As can be seen, the only important thing that really happens during a soft state is the execution of event handlers. Using event handlers can be particularly useful if you want to try and proactively fix a problem before it turns into a hard state. More information on event handlers can be found here.

Hard States

Hard states occur for *services* in the following situations (hard host states are discussed later)...

- When a service check results in a non-OK state and it has been (re)checked the number of times specified by the <*max_check_attempts*> option in the service definition. This is a hard error state.
- When a service recovers from a hard error state. This is considered to be a hard recovery.
- When a service check results in a non-OK state and its corresponding host is either DOWN or UNREACHABLE. This is an exception to the general monitoring logic, but makes perfect sense. If the host isn't up why should we try and recheck the service?

Hard states occur for *hosts* in the following situations...

- When a host check results in a non-OK state and it has been (re)checked the number of times specified by the <*max_check_attempts*> option in the host definition. This is a hard error state.
- When a host recovers from a hard error state. This is considered to be a hard recovery.

Hard State Changes

Before I discuss what happens when a host or service is in a hard state, you need to know about hard state changes. Hard state changes occur when a service or host...

- changes from a hard OK state to a hard non-OK state
- changes from a hard non-OK state to a hard OK-state
- changes from a hard non-OK state of some kind to a hard non-OK state of another kind (i.e. from a hard WARNING state to a hard UNKNOWN state)

Hard State Events

What happens when a service or host is in a hard error state or experiences a hard recovery? Well, that depends on whether or not a hard state change (as described above) has occurred.

If a hard state change has occurred *and* the service or host is in a non-OK state the following things will occur..

- The hard service or host problem is logged.
- Event handlers are executed (if you defined any) to handle the hard problem for the service or host. (Before any event handler is executed, the **$STATETYPE$** macro is set to "**HARD**").
- Contacts will be notified of the service or host problem (if the notification logic allows it).

If a hard state change has occurred *and* the service or host is in an OK state the following things will occur..

- The hard service or host recovery is logged.
- Event handlers are executed (if you defined any) to handle the hard recovery for the service or host. (Before any event handler is executed, the **$STATETYPE$** macro is set to "**HARD**").
- Contacts will be notified of the service or host recovery (if the notification logic allows it).

If a hard state change has NOT occurred *and* the service or host is in a non-OK state the following things will occur..

- Contacts will be re-notified of the service or host problem (if the notification logic allows it).

If a hard state change has NOT occurred *and* the service or host is in an OK state nothing happens. This is because the service or host is in an OK state and was the last time it was checked as well.

Time Periods

or...
"Is This a Good Time?"

Introduction

Time periods allow you to have greater control over when service checks may be run, when host and service notifications may be sent out, and when contacts may receive notifications. With this newly added power come some potential problems, as I will describe later. I was initially very hesitant to introduce time periods because of these snafus. I'll leave it up to you to decide what it right for your particular situation...

How Time Periods Work With Service Checks

Without the implementation of time periods, Nagios would monitor all services that you had defined 24 hours a day, 7 days a week. While this is fine for most services that need monitoring, it doesn't work out so well for others. For instance, do you really need to monitor printers all the time when they're really only used during normal business hours? Perhaps you have development servers which you would prefer to have up, but aren't "mission critical" and therefore don't have to be monitored for problems over the weekend. Time period definitions now allow you to have more control over when such services may be checked...

The <*check_period*> argument of each service definition allows you to specify a time period that tells Nagios when the service can be checked. When Nagios attempts to reschedule a service check, it will make sure that the next check falls within a valid time range within the defined time period. If it doesn't, Nagios will adjust the next service check time to coincide with the next "valid" time in the specified time period. This means that the service may not get checked again for another hour, day, or week, etc.

Potential Problems With Service Checks

If you use time periods which do not cover a 24x7 range, you *will* run into problems, especially if a service (or its corresponding host) is down when the check is delayed until the next valid time in the time period. Here are some of those problems...

1. Contacts will not get re-notified of problems with a service until the next service check can be run.
2. If a service recovers during a time that has been excluded from the check period, contacts will not be notified of the recovery.
3. The status of the service will appear unchanged (in the status log and CGI) until it can be checked next.
4. If all services associated with a particular host are on the same check time period, host problems or recoveries will not be recognized until one of the services can be checked (and therefore notifications may be delayed or not get sent out at all).

Limiting the service check period to anything other than a 24 hour a day, 7 days a week basis can cause a lot of problems. Well, not really problems so much as annoyances and inaccuracies... Unless you have good reason to do so, I would *strongly* suggest that you set the <*check_period*> argument of each service definition to a "24x7" type of time period.

How Time Periods Work With Contact Notifications

Probably the best use of time periods is to control when notifications can be sent out to contacts. By using the <*service_notification_period*> and <*host_notification_period*> arguments in contact definitions, you're able to essentially define an "on call" period for each contact. Note that you can specify different time periods for host and service notifications. This is helpful if you want host notifications to go out to the contact any day of the week, but only have service notifications get sent to the contact on weekdays. It should be noted that these two notification periods should cover *any time* that the contact can be notified. You can control notification times for specific services and hosts on a one-by-one basis as follows...

By setting the <*notification_period*> argument of the host definition, you can control when Nagios is allowed to send notifications out regarding problems or recoveries for that host. When a host notification is about to get sent out, Nagios will make sure that the current time is within a valid range in the <*notification_period*> time period. If it is a valid time, then Nagios will attempt to notify each contact of the host problem. Some contacts may not receive the host notification if their <*host_notification_period*> does not allow for host notifications at that time. If the time is *not* valid within the <*notification_period*> defined for the host, Nagios will not send the notification out to *any* contacts.

You can control notification times for services in a similiar manner to host notification times. By setting the <*notification_period*> argument of the service definition, you can control when Nagios is allowed to send notifications out regarding problems or recoveries for that service. When a service notification is about to get sent out, Nagios will make sure that the current time is within a valid range in the <*notification_period*> time period. If it is a valid time, then Nagios will attempt to notify each contact of the service problem. Some contacts may not receive the service notification if their <*svc_notification_period*> does not allow for service notifications at that time. If the time is *not* valid within the <*notification_period*> defined for the service, Nagios will not send the notification out to *any* contacts.

Potential Problems With Contact Notifications

There aren't really any major problems that you'll run into with using time periods to create custom contact notification times. You do, however, need to be aware that contacts may not always be notified of a service or host problem or recovery. If the time isn't right for both the host or service notification period and the contact notification period, the notification won't go through. Once you weigh the potential problems of time-restricted notifications against your needs, you should be able to come up with a configuration that works well for your situation.

Conclusion

Time periods allow you to have greater control of how Nagios performs its monitoring and notification functions, but can lead to problems. If you are unsure of what type of time periods to implement, or if you are having problems with your current implementation, I would suggest using "24x7" time periods (where all times are valid for each day of the week). Feel free to contact me if you have questions or are running into problems.

Event Handlers

Introduction

Event handlers are optional commands that are executed whenever a host or service state change occurs. An obvious use for event handlers (especially with services) is the ability for Nagios to proactively fix problems before anyone is notified. Another potential use for event handlers might be to log service or host events to an external database.

Event Handler Types

There are two main types of event handlers than can be defined - service event handlers and host event handlers. Event handler commands are (optionally) defined in each host and service definition. Because these event handlers are only associated with particular services or hosts, I will call these "local" event handlers. If a local event handler has been defined for a service or host, it will be executed when that host or service changes state.

You may also specify global event handlers that should be run for *every* host or service state change by using the global_host_event_handler and global_service_event_handler options in your main configuration file. Global event handlers are run immediately *prior* to running a local service or host event handler.

When Are Event Handler Commands Executed?

Service and host event handler commands are executed when a service or host:

- is in a "soft" error state
- initially goes into a "hard" error state
- recovers from a "soft" or "hard" error state

What are "soft" and "hard" states you ask? They are described here .

Event Handler Execution Order

Global event handlers are executed before any local event handlers that you have configured for specific hosts or services.

Writing Event Handler Commands

In most cases, event handler commands will be shell or perl scripts. At a minimum, the scripts should take the following macros as arguments:

Service event handler macros: **$SERVICESTATE$**, **$STATETYPE$**, **$SERVICEATTEMPT$**
Host event handler macros: **$HOSTSTATE$**, **$STATETYPE$**, **$HOSTATTEMPT$**

The scripts should examine the values of the arguments passed in and take any necessary action based upon those values. The best way to understand how event handlers should work is to see and example. Lucky for you, one is provided below. There are also some sample event handler scripts included in the **eventhandlers/** subdirectory of the Nagios distribution. Some of these sample scripts demonstrate the use of external commands to implement redundant monitoring hosts.

Permissions For Event Handler Commands

Any event handler commands you configure will execute with the same permissions as the user under which Nagios is running on your machine. This presents a problem with scripts that attempt to restart system services, as root privileges are generally required to do these sorts of tasks.

Ideally you should evaluate the types of event handlers you will be implementing and grant just enough permissions to the Nagios user for executing the necessary system commands. You might want to try using sudo to accomplish this. Implementation of this is your job, so read the docs and decide if its what you need.

Debugging Event Handler Commands

When you are debugging event handler commands, I would highly recommend that you enable logging of service retries, host retries, and event handler commands. All of these logging options are configured in the main configuration file. Enabling logging for these options will allow you to see exactly when and why event handler commands are being executed.

When you're done debugging your event handler commands you'll probably want to disable logging of service and host retries. They can fill up your log file fast, but if you have enabled log rotation you might not care.

Service Event Handler Example

The example below assumes that you are monitoring the HTTP server on the local machine and have specified **restart-httpd** as the event handler command for the HTTP service definition. Also, I will be assuming that you have set the <max_check_attempts> option for the service to be a value of 4 or greater (i.e. the service is checked 4 times before it is considered to have a real problem). An example service definition (w/ only the fields we discuss) might look like this...

```
define service{
        host_name                       somehost
        service_description             HTTP
        max_check_attempts              4
        event_handler                   restart-httpd
        ...other service variables...
        }
```

Once the service has been defined with an event handler, we must define that event handler as a command. Notice the macros in the command line that I am passing to the event handler - these are important!

```
define command{
        command_name    restart-httpd
        command_line    /usr/local/nagios/libexec/eventhandlers/restart-httpd  $SERVICESTATE$
$STATETYPE$ $SERVICEATTEMPT$
        }
```

Now, let's actually write the event handler script (this is the **/usr/local/nagios/libexec/eventhandlers/restart-httpd** file).

```
#!/bin/sh
#
# Event handler script for restarting the web server on the local machine
#
# Note: This script will only restart the web server if the service is
#       retried 3 times (in a "soft" state) or if the web service somehow
#       manages to fall into a "hard" error state.
#


# What state is the HTTP service in?
case "$1" in
OK)
        # The service just came back up, so don't do anything...
        ;;
WARNING)
        # We don't really care about warning states, since the service is probably still running...
        ;;
UNKNOWN)
        # We don't know what might be causing an unknown error, so don't do anything...
        ;;
```

```
CRITICAL)
        # Aha!  The HTTP service appears to have a problem - perhaps we should restart the server...

        # Is this a "soft" or a "hard" state?
        case "$2" in

        # We're in a "soft" state, meaning that Nagios is in the middle of retrying the
        # check before it turns into a "hard" state and contacts get notified...
        SOFT)

                # What check attempt are we on?  We don't want to restart the web server on the
first
                # check, because it may just be a fluke!
                case "$3" in

                # Wait until the check has been tried 3 times before restarting the web server.
                # If the check fails on the 4th time (after we restart the web server), the state
                # type will turn to "hard" and contacts will be notified of the problem.
                # Hopefully this will restart the web server successfully, so the 4th check will
                # result in a "soft" recovery.  If that happens no one gets notified because we
                # fixed the problem!
                3)
                        echo -n "Restarting HTTP service (3rd soft critical state)..."
                        # Call the init script to restart the HTTPD server
                        /etc/rc.d/init.d/httpd restart
                        ;;
                        esac
                ;;

        # The HTTP service somehow managed to turn into a hard error without getting fixed.
        # It should have been restarted by the code above, but for some reason it didn't.
        # Let's give it one last try, shall we?
        # Note: Contacts have already been notified of a problem with the service at this
        # point (unless you disabled notifications for this service)
        HARD)
                echo -n "Restarting HTTP service..."
                # Call the init script to restart the HTTPD server
                /etc/rc.d/init.d/httpd restart
                ;;
        esac
        ;;
esac
exit 0
```

The sample script provided above will attempt to restart the web server on the local machine in two different instances - after the HTTP service is being retried for the 3rd time (in an "soft" error state) and after the service falls into a "hard" state. The "hard" state situation shouldn't really occur, since the script should restart the service when its still in a "soft" state (i.e. the 3rd check retry), but its left as a fallback anyway.

It should be noted that the service event handler will only be execute the first time that the service falls into a "hard" state. This will prevent Nagios from continuously executing the script to restart the web server when it is in a "hard" state.

External Commands

Introduction

Nagios can process commands from external applications (including CGIs - see the command CGI for an example) and alter various aspects of its monitoring functions based on the commands it receives.

Enabling External Commands

By default, Nagios *does not* check for or process any external commands. If you want to enable external command processing, you'll have to do the following...

- Enable external command checking with the check_external_commands option
- Set the frequency of command checks with the command_check_interval option
- Specify the location of the command file with the command_file option. Its best to put the external command file in its own directory (i.e. */usr/local/nagios/var/rw*).
- Setup proper permissions on the directory containing the external command file. Details on how to do this can be found here.

When Does Nagios Check For External Commands?

- At regular intervals specified by the command_check_interval option in the main configuration file
- Immediately after event handlers are executed. This is in addtion to the regular cycle of external command checks and is done to provide immediate action if an event handler submits commands to Nagios.

Using External Commands

External commands can be used to accomplish a variety of things while Nagios is running. Example of what can be done include temporarily disabling notifications for services and hosts, temporarily disabling service checks, forcing immediate service checks, adding comments to hosts and services, etc.

External Command Examples

Some example scripts that can be used to issue commands to Nagios can be found in the *eventhandlers/* subdirectory of the Nagios distribution. You may have to modify the scripts to accomodate for differences in system command syntaxes, file and directory locations, etc.

Command Format

External commands that are written to the command file have the following format...

[*time*] *command_id*;*command_arguments*

...where *time* is the time (in *time_t* format) that the external application or CGI committed the external command to the command file. Some of the commands that are available are described in the table below, along with their *command_id* and a description of their *command_arguments*.

Implemented Commands

This is a description of the some of the external commands which have been implemented in Nagios. Note that all time arguments should be specified in *time_t* format (seconds since the UNIX epoch).

Command ID	**Command Arguments**	**Command Description**
ADD_HOST_COMMENT	<host_name>;<persistent>;<author>;<comment>	This command is used to associate a comment with the specified host. The *author* argument generally contains the name of the person who entered the comment. The actual comment should not contain any semi-colons. The persistent flag determines whether or not the comment will survive program restarts (1=save comment across program restarts, 0=delete comment on restart).
ADD_SVC_COMMENT	<host_name>;<service_description>;<persistent>;<author>;<comment>	This command is used to associate a comment with the specified host. Note that both the host name and service description are required. The *author* argument generally contains the name of the person who entered the comment. The actual comment should not contain any semi-colons. The persistent flag determines whether or not the comment will survive program restarts (1=save comment across program restarts, 0=delete comment on restart).
DEL_HOST_COMMENT	<comment_id>	This is used to delete a comment having a ID matching *comment_id* for the specified host.
DEL_ALL_HOST_COMMENTS	<host_name>	This is used to delete all comments associated with the specified host.
DEL_SVC_COMMENT	<comment_id>	This is used to delete a comment having a ID matching *comment_id* for the specified service.
DEL_ALL_SVC_COMMENTS	<host_name>;<service_description>	This is used to delete all comments associated with the specified service. Note that both the host name and service description are required.
DELAY_HOST_NOTIFICATION	<host_name>;<next_notification_time>	This will delay the next notification about this host until the time specified by the *next_notification_time* argument. This will have no effect if the host state changes before the next notification is scheduled to be sent out.
DELAY_SVC_NOTIFICATION	<host_name>;<service_description>;<next_notification_time>	This will delay the next notification about this service until the time specified by the *next_notification_time* argument. Note that both the host name and service description are required. This will have no effect if the service state changes before the next notification is scheduled to be sent out. This *does not* delay notifications about the host.

SCHEDULE_SVC_CHECK	<host_name>;<service_description>;<next_check_time>	This will reschedule the next check of the specified service for the time specified by the *next_check_time* argument. Note that both the host name and service description are required.
SCHEDULE_HOST_SVC_CHECKS	<host_name><next_check_time>	This will reschedule the next check of all services on the specified host for the time specified by the *next_check_time* argument.
ENABLE_SVC_CHECK	<host_name>;<service_description>	This will re-enable checks of the specified service. Note that both the host name and service description are required.
DISABLE_SVC_CHECK	<host_name>;<service_description>	This will temporarily disable checks of the specified service. Service checks are automatically re-enabled when Nagios restarts. Issuing this command will have the side effect of temporarily preventing notifications from being sent out for the service. It *does not* prevent notifications about the host from being sent out.
ENABLE_SVC_NOTIFICATIONS	<host_name>;<service_description>	This is used to re-enable notifications for the specified service. Note that both the host name and service description are required.
DISABLE_SVC_NOTIFICATIONS	<host_name>;<service_description>	This is used to temporarily disable notifications from being sent out about the specified service. Notifications are automatically re-enabled when Nagios restarts. Note that both the host name and service description are required. This *does not* disable notifications for the host.
ENABLE_HOST_SVC_NOTIFICATIONS	<host_name>	This is used to re-enable notifications for all services on the specified host. This *does not* enable notifications for the host.
DISABLE_HOST_SVC_NOTIFICATIONS	<host_name>	This is used to temporarily disable notifications for all services on the specified host. This *does not* disable notifications for the host.
ENABLE_HOST_SVC_CHECKS	<host_name>	This will re-enable checks of all services on the specified host. If one or more services were in a non-OK state when they were disabled, contacts may receive notifications if the service(s) recover after the checks are re-enabled.
DISABLE_HOST_SVC_CHECKS	<host_name>	This will temporarily disable checks of all services on the specified host. Service checks are automatically re-enabled when Nagios restarts. Issuing this command will have the side effect of temporarily preventing notifications from being sent out for any of the affected services. It *does not* prevent notifications about the host from being sent out.

ENABLE_HOST_NOTIFICATIONS	<host_name>	This will temporarily disable notifications for this host. Note that this *does not* enable notifications for the services associated with this host.
DISABLE_HOST_NOTIFICATIONS	<host_name>	This will temporarily disable notifications for this host. Notifications are automatically re-enabled when Nagios restarts. Note that this *does not* disable notifications for the services associated with this host.
ENABLE_ALL_NOTIFICATIONS_BEYOND_HOST	<host_name>	This will enable notifications for all hosts and services "beyond" the host specified by the *host_name* argument (from the view of Nagios). This command is most often used in conjunction with redundant monitoring hosts.
DISABLE_ALL_NOTIFICATIONS_BEYOND_HOST	<host_name>	This will temporarily disable notifications for all hosts and services "beyond" the host specified by the *host_name* argument (from the view of Nagios). Notifications are automatically re-enabled when Nagios restarts. This command is most often used in conjunction with redundant monitoring hosts.
ENABLE_NOTIFICATIONS	<execution_time>	This will enable host and service notifications on a program-wide basis at the time specified by the *execution time* argument.
DISABLE_NOTIFICATIONS	<execution_time>	This will disable host and service notifications on a program-wide basis at the time specified by the *execution time* argument.
SHUTDOWN_PROGRAM	<execution_time>	This will cause Nagios to shutdown at the time specified by the *execution_time* argument. Note: Nagios cannot be restarted via the web interface once it has been shutdown.
RESTART_PROGRAM	<execution_time>	This will cause Nagios to flush all configuration state information, re-read all the config files, and restart monitoring at the time specified by the *execution_time* argument
PROCESS_SERVICE_CHECK_RESULT	<host_name>;<service_description>;<return_code>;<plugin_output>	This command is used to submit check results for a particular service to Nagios. These "passive" checks are acted upon in the same manner as normal "active" checks. More information on passive service checks can be found here.
SAVE_STATE_INFORMATION	<execution_time>	This will force Nagios to dump current state information for all services and hosts to the file specified by the state_retention_file variable. You must enable the retain_state_information option for this to work.

READ_STATE_INFORMATION	<execution_time>	This will force Nagios to read previously saved state information for all services and hosts from the file specified by the state_retention_file variable. You must enable the retain_state_information option for this to work.
START_EXECUTING_SVC_CHECKS		This is used to resume the execution of service checks. The execution of service checks may have been stopped at an earlier time by either receiving a *STOP_EXECUTING_SVC_CHECKS* command, or by setting the execute_service_checks option in the main config file to 0. Most often used when implementing redundant monitoring hosts.
STOP_EXECUTING_SVC_CHECKS		This is used to stop the execution of service checks. When service checks are not being executed, Nagios will not keep requeuing checks for a later time, but will not actually execute any checks. This essentially puts Nagios into a "sleep" mode, as far as monitoring is concerned. Most often used when implementing redundant monitoring hosts.
START_ACCEPTING_PASSIVE_SVC_CHECKS		This is used to resume the acceptance of passive service checks for all services. The acceptance of passive service checks may have been stopped at an earlier time by either receiving a *STOP_ACCEPTING_PASSIVE_SVC_CHECKS* command, or by setting the accept_passive_service_checks option in the main config file to 0. If passive checks have been disabled for specific services using the *DISABLE_PASSIVE_SVC_CHECKS* command, passive checks will *not* be accepted for those services, but will for all others.
STOP_ACCEPTING_PASSIVE_SVC_CHECKS		This is used to disable the acceptance of passive service checks for all services.

ENABLE_PASSIVE_SVC_CHECKS	<host_name>;<service_description>	This is used to resume the acceptance of passive service checks for a specific service. The acceptance of passive checks may have been disabled for a service at an earlier time by receiving a *DISABLE_PASSIVE_SVC_CHECKS* command. If passive checks have been disabled for all services either by using the *STOP_ACCEPTING_PASSIVE_SVC_CHECKS* command or by setting the accept_passive_service_checks option in the main config file to 0, passive checks will *not* be accepted for this service.
DISABLE_PASSIVE_SVC_CHECKS	<host_name>;<service_description>	This is used to disable the acceptance of passive service checks for a specific service.

Indirect Host and Service Checks

Introduction

Chances are, many of the services that you're going to be monitoring on your network can be checked directly by using a plugin on the host that runs Nagios. Examples of services that can be checked directly include availability of web, email, and FTP servers. These services can be checked directly by a plugin from the Nagios host because they are publicly accessible resources. However, there are a number of things you may be interested in monitoring that are not as publicly accessible as other services. These "private" resources/services include things like disk usage, processor load, etc. on remote machines. Private resources like these cannot be checked without the use of an intermediary agent. Service checks which require an intermediary agent of some kind to actually perform the check are called *indirect* checks.

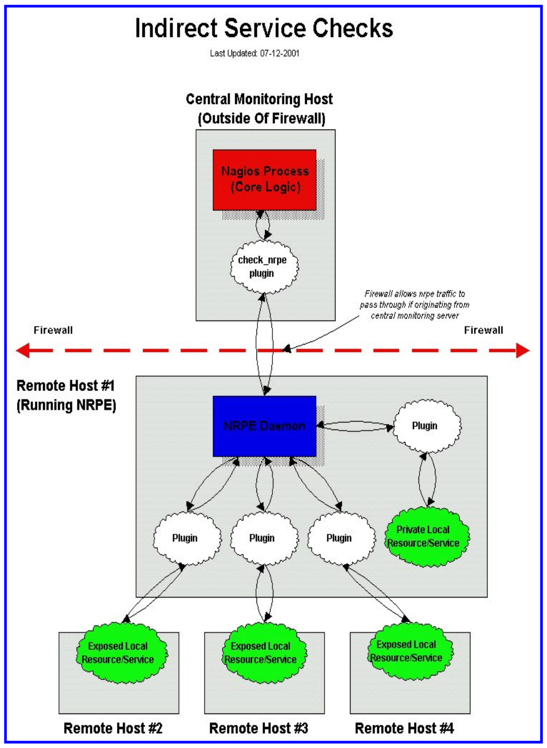
Indirect checks are useful for:

- Monitoring "local" resources (such as disk usage, processer load, etc.) on remote hosts
- Monitoring services and hosts behind firewalls
- Obtaining more realistic results from checks of time-sensitive services between remote hosts (i.e. ping response times between two remote hosts)

There are several methods for performing indirect active checks (passive checks are not discussed here), but I will only talk about how they can be done by using the nrpe addon.

Indirect Service Checks

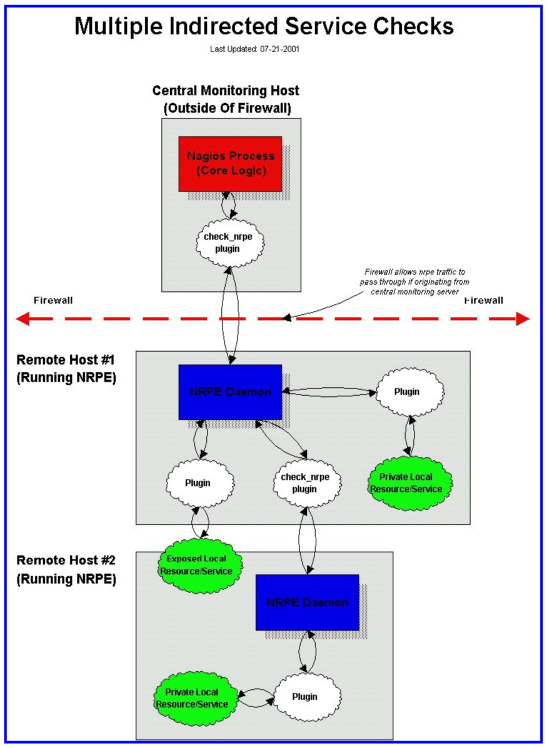
The diagram below shows how indirect service checks work. Click the image for a larger version...


Indirect Service Checks
Last Updated: 07-12-2001
Central Monitoring Host (Outside Of Firewall)
Nagios Process (Core Logic)
check_nrpe plugin
Firewall allows nrpe traffic to pass through if originating from central monitoring server
Firewall
Firewall
Remote Host #1 (Running NRPE)
NRPE Daemon
Plugin
Plugin
Plugin
Plugin
Private Local Resource/Service
Exposed Local Resource/Service
Exposed Local Resource/Service
Exposed Local Resource/Service
Remote Host #2
Remote Host #3
Remote Host #4

Multiple Indirected Service Checks

If you are monitoring servers that lie behind a firewall (and the host running Nagios is outside that firewall), checking services on those machines can prove to be a bit of a pain. Chances are that you are blocking most incoming traffic that would normally be required to perform the monitoring. One solution for performing active checks (passive checks could also be used) on the hosts behind the firewall would be to poke a tiny hold in the firewall filters that allow the Nagios host to make calls to the *nrpe* daemon on one host inside the firewall. The host inside the firewall could then be used as an intermediary in performing checks on the other servers inside the firewall.

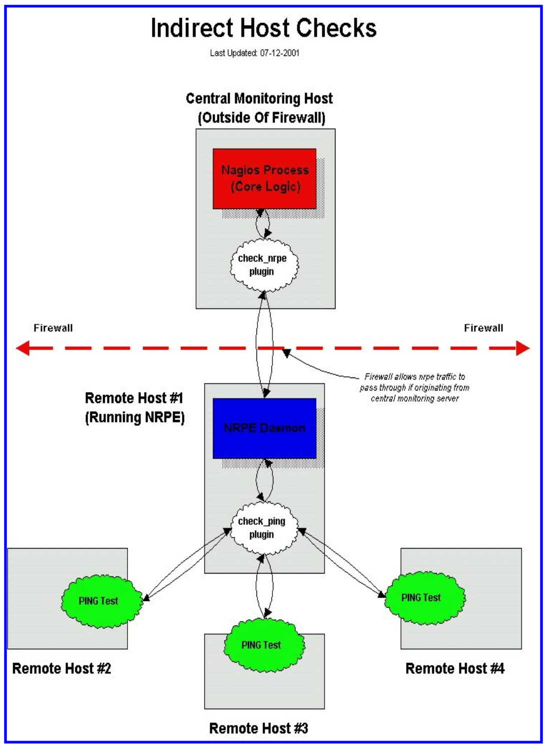
The diagram below show how multiple indirect service checks work. Notice how the *nrpe* daemon is running on hosts #1 and #2. The copy that runs on host #2 is used to allow the *nrpe* agent on host #1 to perform a check of a "private" service on host #2. "Private" services are things like process load, disk usage, etc. that are not directly exposed like SMTP, FTP, and web services. Click on the diagram for a larger image...


Multiple Indirected Service Checks
Last Updated: 07-21-2001
Central Monitoring Host (Outside Of Firewall)
Nagios Process (Core Logic)
check_nrpe plugin
Firewall allows nrpe traffic to pass through if originating from central monitoring server
Firewall
Firewall
Remote Host #1 (Running NRPE)
NRPE Daemon
Plugin
Plugin
check_nrpe plugin
Private Local Resource/Service
Remote Host #2 (Running NRPE)
Exposed Local Resource/Service
NRPE Daemon
Private Local Resource/Service
Plugin

Indirect Host Checks

Indirect host checks work on the same principle as indirect service checks. Basically, the plugin used in the host check command asks an intermediary agent (i.e. a daemon running on a remote host) to perform the host check for it. Indirect host checks are useful when the remote hosts being monitored are located behind a firewall and you want to restrict inbound monitoring traffic to a particular machine. That machine (remote host #1 in the diagram below) performs will perform the host check and return the results back to the top level *check_nrpe* plugin (on the central server). It should be noted that with this setup comes potential problems. If remote host #1 goes down, the *check_nrpe* plugin will not be able to contact the *nrpe* daemon and Nagios will believe that remote hosts #2, #3, and #4 are down, even though this may not be the case. If host #1 is your firewall machine, then the problem isn't really an issue because Nagios will detect that it is down and mark hosts #2, #3, and #4 as being unreachable.

The diagram below shows how an indirect host check can be performed by using the *nrpe* daemon and *check_nrpe* plugin. Click the image for a larger version.


Indirect Host Checks
Last Updated: 07-12-2001
Central Monitoring Host
(Outside Of Firewall)
Nagios Process
(Core Logic)
check_nrpe
plugin
Firewall
Firewall
Firewall allows nrpe traffic to pass through if originating from central monitoring server
Remote Host #1
(Running NRPE)
NRPE Daemon
check_ping
plugin
PING Test
PING Test
PING Test
Remote Host #2
Remote Host #3
Remote Host #4

Passive Service Checks

Introduction

On of the features of Nagios is that is can process service check results that are submitted by external applications. Service checks which are performed and submitted to Nagios by external apps are called *passive* checks. Passive checks can be contrasted with *active* checks, which are service checks that have been initiated by Nagios.

Why The Need For Passive Checks?

Passive checks are useful for monitoring services that are:

- located behind a firewall, and can therefore not be checked actively from the host running Nagios
- asynchronous in nature and can therefore not be actively checked in a reliable manner (e.g. SNMP traps, security alerts, etc.)

How Do Passive Checks Work?

The only real difference between active and passive checks is that active checks are initiated by Nagios, while passive checks are performed by external applications. Once an external application has performed a service check (either actively or by having received an synchronous event like an SNMP trap or security alert), it submits the results of the service "check" to Nagios through the external command file.

The next time Nagios processes the contents of the external command file, it will place the results of all passive service checks into a queue for later processing. The same queue that is used for storing results from active checks is also used to store the results from passive checks.

Nagios will periodically execute a service reaper event and scan the service check result queue. Each service check result, regardless of whether the check was active or passive, is processed in the same manner. The service check logic is exactly the same for both types of checks. This provides a seamless method for handling both active and passive service check results.

How Do External Apps Submit Service Check Results?

External applications can submit service check results to Nagios by writing a PROCESS_SERVICE_CHECK_RESULT external command to the external command file.

The format of the command is as follows:

[<timestamp>] PROCESS_SERVICE_CHECK_RESULT;<host_name>;<description>;<return_code>;<plugin_output>

where...

- *timestamp* is the time in time_t format (seconds since the UNIX epoch) that the service check was perfomed (or submitted). Please note the single space after the right bracket.
- *host_name* is the short name of the host associated with the service in the service definition
- *description* is the description of the service as specified in the service definition
- *return_code* is the return code of the check (0=OK, 1=WARNING, 2=CRITICAL, 3=UNKNOWN)
- *plugin_output* is the text output of the service check (i.e. the plugin output)

Note that in order to submit service checks to Nagios, a service must have already been defined in the object configuration file! Nagios will ignore all check results for services that had not been configured before it was last (re)started.

If you only want passive results to be provided for a specific service (i.e. active checks should not be performed), simply set the *active_checks_enabled* member of the service definition to 0. This will prevent Nagios from ever actively performing a check of the service. Make sure that the *passive_checks_enabled* member of the service definition is set to 1. If it isn't, Nagios won't process passive checks for the service!

An example shell script of how to submit passive service check results to Nagios can be found in the documentation on volatile services.

Submitting Passive Service Check Results From Remote Hosts

If an application that resides on the same host as Nagios is sending passive service check results, it can simply write the results directly to the external command file as outlined above. However, applications on remote hosts can't do this so easily. In order to allow remote hosts to send passive service check results to the host that runs Nagios, I've developed the nsca addon. The addon consists of a daemon that runs on the Nagios hosts and a client that is executed from remote hosts. The daemon will listen for connections from remote clients, perform some basic validation on the results being submitted, and then write the check results directly into the external command file (as described above). More information on the nsca addon can be found here...

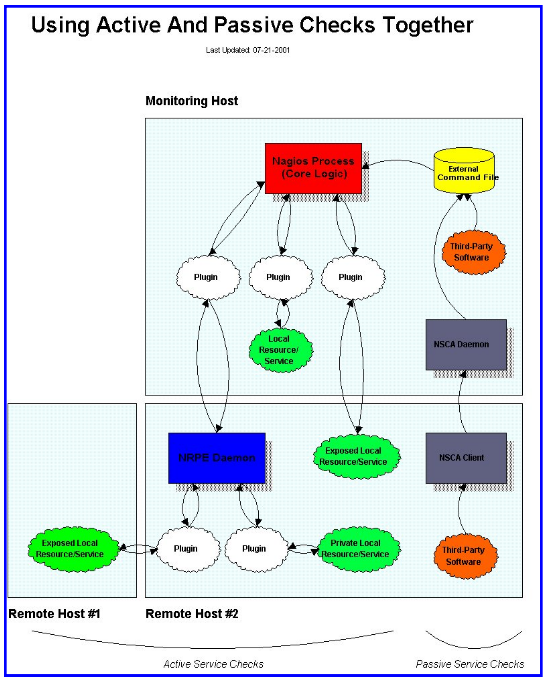
Using Both Active And Passive Service Checks

Unless you're implementing a distributed monitoring environment with the central server accepting only passive service checks (and not performing any active checks), you'll probably be using both types of checks in your setup. As mentioned before, active checks are more suited for services that lend themselves to periodic checks (availability of an FTP or web server, etc), whereas passive checks are better off at handling asynchronous events that occur at variable intervals (security alerts, etc.).

The image below gives a visual representation of how active and passive service checks can both be used to monitor network resources (click on the image for a larger version).

The orange bubbles on the right side of the image are third-party applications that submit passive check results to Nagios' external command file. One of the applications resides on the same host as Nagios, so it can write directly to the command file. The other application resides on a remote host and makes used of the nsca client program and daemon to transfer the passive check results to Nagios.

The items on the left side of the image represent active service checks that Nagios is performing. I've shown how the checks can be made for local resources (disk usage, etc.), "exposed" resources on remote hosts (web server, FTP server, etc.), and "private" resources on remote hosts (remote host disk usage, processor load, etc.). In this example, the private resources on the remote hosts are actually checked by making use of the nrpe addon, which facilitates the execution of plugins on remote hosts.



Volatile Services

Introduction

Nagios has the ability to distinguish between "normal" services and "volatile" services. The *is_volatile* option in each service definition allows you to specify whether a specific service is volatile or not. For most people, the majority of all monitored services will be non-volatile (i.e. "normal"). However, volatile services can be very useful when used properly...

What Are They Useful For?

Volatile services are useful for monitoring...

- things that automatically reset themselves to an "OK" state each time they are checked
- events such as security alerts which require attention every time there is a problem (and not just the first time)

What's So Special About Volatile Services?

Volatile services differ from "normal" services in three important ways. *Each time* they are checked when they are in a hard non-OK state, and the check returns a non-OK state (i.e. no state change has occurred)...

- the non-OK service state is logged
- contacts are notified about the problem (if that's what should be done)
- the event handler for the service is run (if one has been defined)

These events normally only occur for services when they are in a non-OK state and a hard state change has just occurred. In other words, they only happen the first time that a service goes into a non-OK state. If future checks of the service result in the same non-OK state, no hard state change occurs and none of the events mentioned take place again.

The Power Of Two

If you combine the features of volatile services and passive service checks, you can do some very useful things. Examples of this include handling SNMP traps, security alerts, etc.

How about an example... Let's say you're running Psionic Software's PortSentry product (which is free, by the way) to detect port scans on your machine and automatically firewall potential intruders. If you want to let Nagios know about port scans, you could do the following..

In Nagios:

- Configure a service called *Port Scans* and associate it with the host that PortSentry is running on.
- Set the *max_check_attempts* option in the service definition to 1. This will tell Nagios to immediate force the service into a hard state when a non-OK state is reported.
- Either set the *active_checks_enabled* option to 0 or set the *check_time* option in the service definition to a timeperiod that contains *no* valid time ranges. Doing either of these will prevent Nagios from ever actively checking the service. Even though the service check will get scheduled, it will never actually be checked.

In PortSentry:

- Edit your PortSentry configuration file (portsentry.conf), define a command for the **KILL_RUN_CMD** directive as follows:

  KILL_RUN_CMD="/usr/local/Nagios/libexec/eventhandlers/submit_check_result <*host_name*> 'Port Scans' 2 'Port scan from host $TARGET$ on port $PORT$. Host has been firewalled.'"

  Make sure to replace <*host_name*> with the short name of the host that the service is associated with.

Create a shell script in the */usr/local/nagios/libexec/eventhandlers* directory named *submit_check_result*. The contents of the shell script should be something similiar to the following...

```
#!/bin/sh

# Write a command to the Nagios command file to cause
# it to process a service check result

echocmd="/bin/echo"

CommandFile="/usr/local/nagios/var/rw/nagios.cmd"

# get the current date/time in seconds since UNIX epoch
datetime=`date +%s`

# create the command line to add to the command file
cmdline="[$datetime] PROCESS_SERVICE_CHECK_RESULT;$1;$2;$3;$4"

# append the command to the end of the command file
`$echocmd $cmdline >> $CommandFile`
```

Note that if you are running PortSentry as root, you will have to make additions to the script to reset file ownership and permissions so that Nagios and the CGIs can read/modify the command file. Details on permissions/ownership of the command file can be found here.

So what happens when PortSentry detects a port scan on the machine?

- It blocks the host (this is a function of the PortSentry software)
- It executes the *submit_check_result* shell script to send the security alert info to Nagios
- Nagios reads the command file, recognized the port scan entry as a passive service check
- Nagios processes the results of the service by logging the CRITICAL state, sending notifications to

contacts (if configured to do so), and executes the event handler for the *Port Scans* service (if one is defined)

Service Result Freshness Checks

Introduction

Nagios supports a feature that does "freshness" checking on the results of service checks. This feature is useful when you want to ensure that passive checks are being received as frequently as you want. Although freshness checking can be used in a number of situations, it is primarily useful when attempting to configure a distributed monitoring environment.

The purpose of "freshness" checking is to ensure that service checks are being provided passively by external applications on a regular basis. If the results of a particular service check (for which freshness checking has been enabled) is determined to be "stale", Nagios will force an active check of that service.

Configuring Freshness Checking

Before you configure per-service freshness threshold, you must enable freshness checking using the check_service_freshness and freshness_check_interval directives in the main config file.

So how do you go about enabling freshness checking for a particular service? Well, at the moment you can only enable freshness checking of services if you are using template-based object configuration file(s). The older object configuration files formats have not been expanded to support freshness checking.

Assuming you're using the template-based object configuration file(s), you need to configure service definitions as follows.

- The **check_freshness** option in the service definition should be set to 1. This enables "freshness" checking for the service.
- The **freshness_threshold** option in the service definition should be set to a value (usually in seconds, unless you changed the interval_length directive) which reflects how "fresh" the results for the service should be.
- The **check_command** option in the service definition should reflect valid command that should be used to actively check the service when it is detected as being "stale".

How The Freshness Threshold Works

Nagios periodically checks the "freshness" of the results for all services that have freshness checking enabled. The *freshness_threshold* option in each service definition is used to determine how "fresh" the results for each service should be. For example, if you set the *freshness_threshold* option to 5 for one of your services and your interval_length directive is set to 60 seconds, Nagios will consider that service to be "stale" if its results are older than 5 minutes. If you do not specify a value for the *freshness_threshold* option (or you set it to zero), Nagios will automatically calculate a "freshness" threshold to use by looking at either the *normal_check_interval* or *retry_check_interval* options (depending on what type of state the service is currently in).

What Happens When A Service Check Result Becomes "Stale"

If the check results of a service are found to be "stale" (as described above), Nagios will force an active check of the service by executing the command specified by the *check_command* option in the service definition. It is important to note that an active service check which is being forced because the service was detected as being "stale" gets executed *even if active service checks are disabled on a program-wide or service-specific basis*.

Working With Passive-Only Checks

As I mentioned earlier, freshness checking is of most use when you are dealing with services that get their results from passive checks. More often than not (as in the case with distributed monitoring setups), these services may not be getting *all* of their results from passive checks - no results are obtained from active checks.

An example of a passive-only service might be one that reports the status of your nightly backup jobs. Perhaps you have a external script that submit the results of the backup job to Nagios once the backup is completed. In this case, all of the checks/results for the service are provided by an external application using passive checks. In order to ensure that the status of the backup job gets reported every day, you may want to enable freshness checking for the service. If the external script doesn't submit the results of the backup job, you can have Nagios fake a critical result by doing something like this...

Here's what the definition for the service might look like (some required options are omitted)...

```
define service{
        host_name                backup-server
        service_description      ArcServe Backup Job
        active_checks_enabled    0                        ; active checks are NOT enabled
        passive_checks_enabled   1                        ; passive checks are enabled (this is how
results are reported)
        check_freshness          1
        freshness_threshold      93600                    ; 26 hour threshold, since backups may not
always finish at the same time
        check_command            no-backup-report         ; this command is run only if the service
results are "stale"
        ...other options...
        }
```

Notice that active checks are disabled for the service. This is because the results for the service are only made by an external application using passive checks. Freshness checking is enabled and the freshness threshold has been set to 26 hours. This is a bit longer than 24 hours because backup jobs sometimes run late from day to day (depending on how much data there is to backup, how much network traffic is present, etc.). The *no-backup-report* command is executed only if the results of the service are determined to be "stale". The definition of the *no-backup-report* command might look like this...

```
define command{
        command_name    no-backup-report
        command_line    /usr/local/nagios/libexec/nobackupreport.sh
        }
```

The **nobackupreport.sh** script in your */usr/local/nagios/libexec* directory might look something like this:

```
#!/bin/sh

/bin/echo "CRITICAL: Results of backup job were not reported!"

exit 2
```

If Nagios detects that the service results are stale, it will run the **no-backup-report** command as an active service check (even though active checks are disabled for this specific service - remember that this is a special case). This causes the */usr/local/nagios/libexec/nobackupreport.sh* script to be executed, which returns a critical state. The service go into to a critical state (if it isn't already there) and someone will probably get notified of the problem.

Distributed Monitoring

Introduction

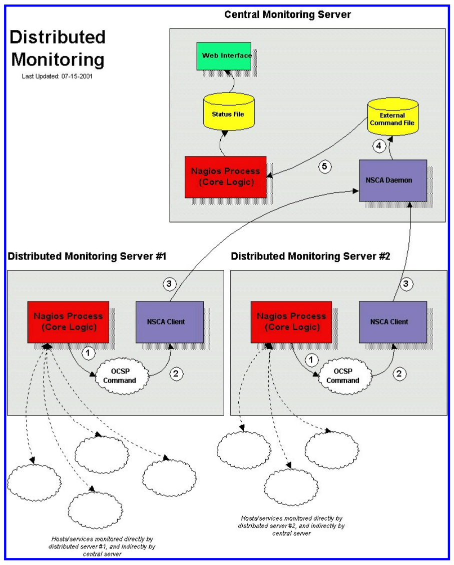
Nagios can be configured to support distributed monitoring of network services and resources. I'll try to briefly explan how this can be accomplished...

Goals

The goal in the distributed monitoring environment that I will describe is to offload the overhead (CPU usage, etc.) of performing service checks from a "central" server onto one or more "distributed" servers. Most small to medium sized shops will not have a real need for setting up such an environment. However, when you want to start monitoring hundreds or even thousands of *hosts* (and several times that many services) using Nagios, this becomes quite important.

Reference Diagram

The diagram below should help give you a general idea of how distributed monitoring works with Nagios. I'll be referring to the items shown in the diagram as I explain things...


Central Monitoring Server
Distributed Monitoring
Last Updated: 07-15-2001
Web Interface
Status File
External Command File
Nagios Process (Core Logic)
NSCA Daemon
1
2
3
4
5
Distributed Monitoring Server #1
Distributed Monitoring Server #2
NSCA Client
OCSP Command
Hosts/services monitored directly by distributed server #1, and indirectly by central server
Hosts/services monitored directly by distributed server #2, and indirectly by central server

Central Server vs. Distributed Servers

When setting up a distributed monitoring environment with Nagios, there are differences in the way the central and distributed servers are configured. I'll show you how to configure both types of servers and explain what effects the changes being made have on the overall monitoring. For starters, lets describe the purpose of the different types of servers...

The function of a *distributed server* is to actively perform checks all the services you define for a "cluster" of hosts. I use the term "cluster" loosely - it basically just mean an arbitrary group of hosts on your network. Depending on your network layout, you may have several cluters at one physical location, or each cluster may be separated by a WAN, its own firewall, etc. The important thing to remember to that for each cluster of hosts (however you define that), there is one distributed server that runs Nagios and monitors the services on the hosts in the cluster. A distributed server is usually a bare-bones installation of Nagios. It doesn't have to have the web interface installed, send out notifications, run event handler scripts, or do anything other than execute service checks if you don't want it to. More detailed information on configuring a distributed server comes later...

The purpose of the *central server* is to simply listen for service check results from one or more distributed servers. Even though services are occassionally actively checked from the central server, the active checks are only performed in dire circumstances, so lets just say that the central server only accepts passive check for now. Since the central server is obtaining passive service check results from one or more distributed servers, it serves as the focal point for all monitoring logic (i.e. it sends out notifications, runs event handler scripts, determines host states, has the web interface installed, etc).

Obtaining Service Check Information From Distributed Monitors

Okay, before we go jumping into configuration detail we need to know how to send the service check results from the distributed servers to the central server. I've already discussed how to submit passive check results to Nagios from same host that Nagios is running on (as described in the documentation on passive checks), but I haven't given any info on how to submit passive check results from other hosts.

In order to facilitate the submission of passive check results to a remote host, I've written the nsca addon. The addon consists of two pieces. The first is a client program (send_nsca) which is run from a remote host and is used to send the service check results to another server. The second piece is the nsca daemon (nsca) which either runs as a standalone daemon or under inetd and listens for connections from client programs. Upon receiving service check information from a client, the daemon will sumbit the check information to Nagios (on the central server) by inserting a *PROCESS_SVC_CHECK_RESULT* command into the external command file, along with the check results. The next time Nagios checks for external commands, it will find the passive service check information that was sent from the distributed server and process it. Easy, huh?

Distributed Server Configuration

So how exactly is Nagios configured on a distributed server? Basically, its just a bare-bones installation. You don't need to install the web interface or have notifications sent out from the server, as this will all be handled by the central server.

Key configuration changes:

- Only those services and hosts which are being monitored directly by the distributed server are defined in the object configuration file.
- The distributed server has its enable_notifications directive set to 0. This will prevent any notifications from being sent out by the server.
- The distributed server is configured to obsess over services.
- The distributed server has an ocsp command defined (as described below).

In order to make everything come together and work properly, we want the distributed server to report the results of *all* service checks to Nagios. We could use event handlers to report *changes* in the state of a service, but that just doesn't cut it. In order to force the distributed server to report all service check results, you must enabled the obsess_over_services option in the main configuration file and provide a ocsp_command to be run after every service check. We will use the ocsp command to send the results of all service checks to the central server, making use of the send_nsca client and nsca daemon (as described above) to handle the tranmission.

In order to accomplish this, you'll need to define an ocsp command like this:

ocsp_command=submit_check_result

The command definition for the *submit_check_result* command looks something like this:

```
define command{
        command_name    submit_check_result
        command_line    /usr/local/nagios/libexec/eventhandlers/submit_check_result $HOSTNAME$
'$SERVICEDESC$' $SERVICESTATE$ '$OUTPUT$'
        }
```

The *submit_check_result* shell scripts looks something like this (replace *central_server* with the IP address of the central server):

```
        #!/bin/sh

        # Arguments:
        #  $1 = host_name (Short name of host that the service is
        #       associated with)
        #  $2 = svc_description (Description of the service)
        #  $3 = state_string (A string representing the status of
        #       the given service - "OK", "WARNING", "CRITICAL"
        #       or "UNKNOWN")
        #  $4 = plugin_output (A text string that should be used
        #       as the plugin output for the service checks)
        #

        # Convert the state string to the corresponding return code
        return_code=-1

        case "$3" in
            OK)
                        return_code=0
                    ;;
                WARNING)
                    return_code=1
                        ;;
                CRITICAL)
                    return_code=2
                        ;;
                UNKNOWN)
                    return_code=-1
                        ;;
        esac

        # pipe the service check info into the send_nsca program, which
        # in turn transmits the data to the nsca daemon on the central
        # monitoring server

        /bin/echo -e "$1\t$2\t$return_code\t$4\n" | /usr/local/nagios/bin/send_nsca central_server -
c /usr/local/nagios/etc/send_nsca.cfg
```

The script above assumes that you have the send_nsca program and it configuration file (send_nsca.cfg) located in the */usr/local/nagios/bin/* and */usr/local/nagios/etc/* directories, respectively.

That's it! We've sucessfully configured a remote host running Nagios to act as a distributed monitoring server. Let's go over exactly what happens with the distributed server and how it sends service check results to Nagios (the steps outlined below correspond to the numbers in the reference diagram above):

1. After the distributed server finishes executing a service check, it executes the command you defined by the ocsp_command variable. In our example, this is the */usr/local/nagios/libexec/eventhandlers/submit_check_result* script. Note that the definition for the *submit_check_result* command passed four pieces of information to the script: the name of the host the service is associated with, the service description, the return code from the service check, and the plugin output from the service check.
2. The *submit_check_result* script pipes the service check information (host name, description, return code, and output) to the *send_nsca* client program.
3. The *send_nsca* program transmits the service check information to the *nsca* daemon on the central monitoring server.
4. The *nsca* daemon on the central server takes the service check information and writes it to the external command file for later pickup by Nagios.
5. The Nagios process on the central server reads the external command file and processes the passive service check information that originated from the distributed monitoring server.

Central Server Configuration

We've looked at hot distributed monitoring servers should be configured, so let's turn to the central server. For all intensive purposes, the central is configured as you would normally configure a standalone server. It is setup as follows:

- The central server has the web interface installed (optional, but recommended)
- The central server has its enable_notifications directive set to 1. This will enable notifications. (optional, but recommended)

- The central server has active service checks disabled (optional, but recommended - see notes below)
- The central server has external command checks enabled (required)
- The central server has passive service checks enabled (required)

There are three other very important things that you need to keep in mind when configuring the central server:

- The central server must have service definitions for *all services* that are being monitored by all the distributed servers. Nagios will ignore passive check results if they do not correspond to a service that has been defined.
- If you're only using the central server to process services whose results are going to be provided by distributed hosts, you can simply disable all active service checks on a program-wide basis by setting the execute_service_checks directive to 0. If you're using the central server to actively monitor a few services on its own (without the aid of distributed servers), the *enable_active_checks* option of the defintions for service being monitored by distributed servers should be set to 0. This will prevent Nagios from actively checking those services.

It is important that you either disable all service checks on a program-wide basis or disable the *enable_active_checks* option in the definitions for each service that is monitored by a distributed server. This will ensure that active service checks are never executed under normal circumstances. The services will keep getting rescheduled at their normal check intervals (3 minutes, 5 minutes, etc...), but the won't actually be executed. This rescheduling loop will just continue all the while Nagios is running. I'll explain why this is done in a bit...

That's it! Easy, huh?

Problems With Passive Checks

For all intensive purposes we can say that the central server is relying solely on passive checks for monitoring. The main problem with relying completely on passive checks for monitoring is the fact that Nagios must rely on something else to provide the monitoring data. What if the remote host that is sending in passive check results goes down or becomes unreachable? If Nagios isn't actively checking the services on the host, how will it know that there is a problem?

Fortunately, there is a way we can handle these types of problems...

Freshness Checking

Nagios supports a feature that does "freshness" checking on the results of service checks. More information freshness checking can be found here. This features gives some protection against situations where remote hosts may stop sending passive service checks into the central monitoring server. The purpose of "freshness" checking is to ensure that service checks are either being provided passively by distributed servers on a regular basis or performed actively by the central server if the need arises. If the service check results provided by the distributed servers get "stale", Nagios can be configured to force active checks of the service from the central monitoring host.

So how do you do this? On the central monitoring server you need to configure services that are being monitoring by distributed servers as follows...

- The *check_freshness* option in the service definitions should be set to 1. This enables "freshness" checking for the services.
- The *freshness_threshold* option in the service definitions should be set to a value which reflects how "fresh" the results for the services (provided by the distributed servers) should be.
- The *check_command* option in the service definitions should reflect valid commands that can be used to actively check the service from the central monitoring server.

Nagios periodically checks the "freshness" of the results for all services that have freshness checking enabled. The *freshness_threshold* option in each service definition is used to determine how "fresh" the results for each service should be. For example, if you set this value to 5 for one of your services and your interval_length directive is set to 60 seconds, Nagios will consider the service results to be "stale" if they're older than 5 minutes. If you do not specify a value for the *freshness_threshold* option, Nagios will automatically calculate a "freshness" threshold by looking at either the *normal_check_interval* or *retry_check_interval* options (depending on what type of state the service is in). If the service results are found to be "stale", Nagios will run the service check command specified by the *check_command* option in the service definition, thereby actively checking the service.

Remember that you have to specify a *check_command* option in the service definitions that can be used to actively check the status of the service from the central monitoring server. Under normal circumstances, this check command is never executed (because active checks were disabled on a program-wide basis or for the specific services). When freshness checking is enabled, Nagios will run this command to actively check the status of the service *even if active checks are disabled on a program-wide or service-specific basis*.

If you are unable to define commands to actively check a service from the central monitoring host (or if turns out to be a major pain), you could simply define all your services with the *check_command* option set to run a dummy script that returns a critical status. Here's an example... Let's assume you define a command called 'service-is-stale' and use that command name in the *check_command* option of your services. Here's what the definition would look like...

```
define command{
        command_name    service-is-stale
        command_line    /usr/local/nagios/libexec/staleservice.sh
        }
```

The **staleservice.sh** script in your */usr/local/nagios/libexec* directory might look something like this:

```
#!/bin/sh

/bin/echo "CRITICAL: Service results are stale!"

exit 2
```

When Nagios detects that the service results are stale and runs the **service-is-stale** command, the */usr/local/nagios/libexec/staleservice.sh* script is executed and the service will go into a critical state. This would likely cause notifications to be sent out, so you'll know that there's a problem.

Performing Host Checks

At this point you know how to obtain service check results passivly from distributed servers. This means that the central server is not actively checking services on its own. But what about host checks? You still need to do them, so how?

Since host checks usually compromise a small part of monitoring activity (they aren't done unless absolutely necessary), I'd recommend that you perform host checks actively from the central server. That means that you define host checks on the central server the same way that you do on the distributed servers (and the same way you would in a normal, non-distributed setup).

There are ways to obtain host checks passively, but implementing them is beyond the scope of what I care to write about at this time. :-)

Redundant and Failover Network Monitoring

Introduction

This section describes a few scenarios for implementing redundant monitoring hosts an various types of network layouts. With redundant hosts, you can maintain the ability to monitor your network when the primary host that runs Nagios fails or when portions of your network become unreachable.

Note: If you are just learning how to use Nagios, I would suggest not trying to implement redudancy until you have becoming familiar with the prerequisites I've laid out. Redundancy is a relatively complicated issue to understand, and even more difficult to implement properly.

Index

Prerequisites

Before you can even think about implementing redundancy with Nagios, you need to be familiar with the following...

- Implementing event handlers for hosts and services
- Issuing external commands to Nagios via shell scripts
- Executing plugins on remote hosts using either the nrpe addon or some other method
- Checking the status of the Nagios process with the check_nagios plugin

Sample Scripts

All of the sample scripts that I use in this documentation can be found in the *eventhandlers/* subdirectory of the Nagios distribution. You'll probably need to modify them to work on your system...

Scenario 1 - Redundant Monitoring

Introduction

This is an easy (and naive) method of implementing redundant monitoring hosts on your network and it will only protect against a limited number of failures. More complex setups are necessary in order to provide smarter redundancy, better redundancy across different network segments, etc.

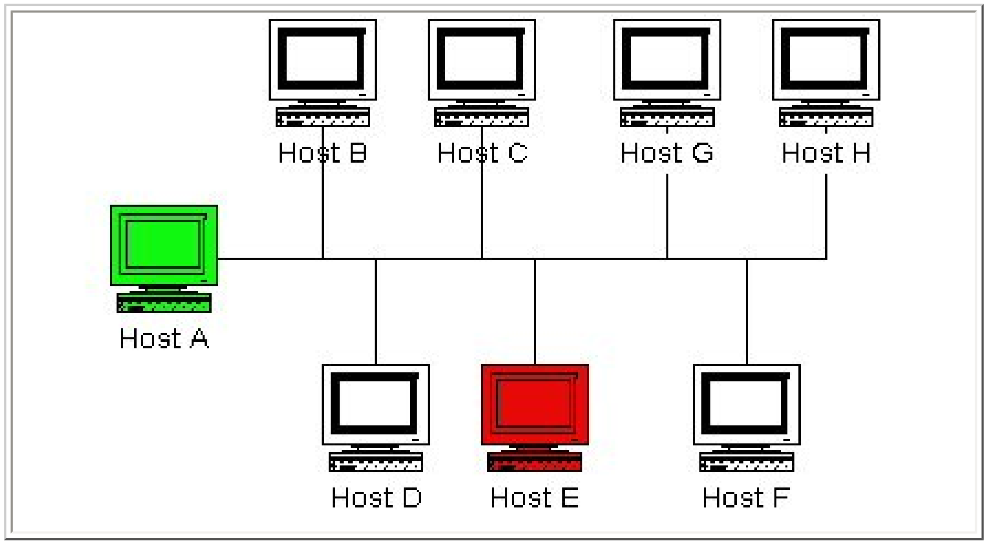
Goals

The goal of this type of redundancy implementation is simple. Both the "master" and "slave" hosts monitor the same hosts and service on the network. Under normal circumstances only the "master" host will be sending out notifications to contacts about problems. We want the "slave" host running Nagios to take over the job of notifying contacts about problems if:

1. The "master" host that runs Nagios is down or..
2. The Nagios process on the "master" host stops running for some reason

Network Layout Diagram

The diagram below shows a very simple network setup. For this scenario I will be assuming that hosts A and E are both running Nagios and are monitoring all the hosts shown. Host A will be considered the "master" host and host E will be considered the "slave" host.



Initial Program Settings

The slave host (host E) has its initial enable_notifications directive disabled, thereby preventing it from sending out any host or service notifications. You also want to make sure that the slave host has its check_external_commands directive enabled. That was easy enough...

Initial Configuration

Next we need to consider the differences between the object configuration file(s) on the master and slave hosts...

I will assume that you have the master host (host A) setup to monitor services on all hosts shown in the diagram above. The slave host (host E) should be setup to monitor the same services and hosts, with the following additions in the configuration file...

- The host definition for host A (in the host E configuration file) should have a host event handler defined. Lets say the name of the host event handler is handle-master-host-event.
- The configuration file on host E should have a service defined to check the status of the Nagios process on host A. Lets assume that you define this service check to run the *check_nagios* plugin on host A. This can be done by using one of the methods described in this FAQ.
- The service definition for the Nagios process check on host A should have an event handler defined. Lets say the name of the service event handler is handle-master-proc-event.

It is important to note that host A (the master host) has no knowledge of host E (the slave host). In this scenario it simply doesn't need to. Of course you may be monitoring services on host E from host A, but that has nothing to do with the implementation of redundancy...

Event Handler Command Definitions

We need to stop for a minute and describe what the command definitions for the event handlers on the slave host look like. Here is an example...

```
define command{
        command_name    handle-master-host-event
        command_line    /usr/local/nagios/libexec/eventhandlers/handle-master-host-event $HOSTSTATE$
$STATETYPE$
        }
```

```
define command{
        command_name    handle-master-proc-event
        command_line    /usr/local/nagios/libexec/eventhandlers/handle-master-proc-event
$SERVICESTATE$ $STATETYPE$
        }
```

This assumes that you have placed the event handler scripts in the */usr/local/nagios/libexec/eventhandlers* directory. You may place them anywhere you wish, but you'll need to modify the examples I've given here.

Event Handler Scripts

Okay, now lets take a look at what the event handler scripts look like...

Host Event Handler (handle-master-host-event):

```
#!/bin/sh

# Only take action on hard host states...
case "$2" in
HARD)
        case "$1" in
        DOWN)
                # The master host has gone down!
                # We should now become the master host and take
                # over the responsibilities of monitoring the
                # network, so enable notifications...
                /usr/local/nagios/libexec/eventhandlers/enable_notifications
                ;;
        UP)
                # The master host has recovered!
                # We should go back to being the slave host and
                # let the master host do the monitoring, so
                # disable notifications...
                /usr/local/nagios/libexec/eventhandlers/disable_notifications
                ;;
        esac
        ;;
esac
exit 0
```

Service Event Handler (handle-master-proc-event):

```
#!/bin/sh

# Only take action on hard service states...
case "$2" in
HARD)
        case "$1" in
        CRITICAL)
                # The master Nagios process is not running!
                # We should now become the master host and
                # take over the responsibility of monitoring
                # the network, so enable notifications...
                /usr/local/nagios/libexec/eventhandlers/enable_notifications
                ;;
        WARNING)
        UNKNOWN)
```

```
                # The master Nagios process may or may not
                # be running.. We won't do anything here, but
                # to be on the safe side you may decide you
                # want the slave host to become the master in
                # these situations...
                ;;
        OK)
                # The master Nagios process running again!
                # We should go back to being the slave host,
                # so disable notifications...
                /usr/local/nagios/libexec/eventhandlers/disable_notifications
                ;;
        esac
        ;;
esac
exit 0
```

<u>What This Does For Us</u>

The slave host (host E) initially has notifications disabled, so it won't send out any host or service notifications while the Nagios process on the master host (host A) is still running.

The Nagios process on the slave host (host E) becomes the master host when...

- The master host (host A) goes down and the *handle-master-host-event* host event handler is executed.
- The Nagios process on the master host (host A) stops running and the *handle-master-proc-event* service event handler is executed.

When the Nagios process on the slave host (host E) has notifications enabled, it will be able to send out notifications about any service or host problems or recoveries. At this point host E has effectively taken over the responsibility of notifying contacts of host and service problems!

The Nagios process on host E returns to being the slave host when...

- Host A recovers and the *handle-master-host-event* host event handler is executed.
- The Nagios process on host A recovers and the *handle-master-proc-event* service event handler is executed.

When the Nagios process on host E has notifications disabled, it will not send out notifications about any service or host problems or recoveries. At this point host E has handed over the responsibilities of notifying contacts of problems to the Nagios process on host A. Everything is now as it was when we first started!

<u>Time Lags</u>

Redundancy in Nagios is by no means perfect. One of the more obvious problems is the lag time between the master host failing and the slave host taking over. This is affected by the following...

- The time between a failure of the master host and the first time the slave host detects a problem
- The time needed to verify that the master host really does have a problem (using service or host check retries on the slave host)
- The time between the execution of the event handler and the next time that Nagios checks for external commands

You can minimize this lag by...

- Ensuring that the Nagios process on host E (re)checks one or more services at a high frequency. This is done by using the *check_interval* and *retry_interval* arguments in each service definition.
- Ensuring that the number of host rechecks for host A (on host E) allow for fast detection of host problems. This is done by using the *max_check_attempts* argument in the host definition.
- Increase the frequency of external command checks on host E. This is done by modifying the command_check_interval option

in the main configuration file.

When Nagios recovers on the host A, there is also some lag time before host E returns to being a slave host. This is affected by the following...

- The time between a recovery of host A and the time the Nagios process on host E detects the recovery
- The time between the execution of the event handler on host B and the next time the Nagios process on host E checks for external commands

The exact lag times between the transfer of monitoring responsibilities will vary depending on how many services you have defined, the interval at which services are checked, and a lot of pure chance. At any rate, its definitely better than nothing.

Special Cases

Here is one thing you should be aware of... If host A goes down, host E will have notifications enabled and take over the responsibilities of notifying contacts of problems. When host A recovers, host E will have notifications disabled. If - when host A recovers - the Nagios process on host A does not start up properly, there will be a period of time when neither host is notifying contacts of problems! Fortunately, the service check logic in Nagios accounts for this. The next time the Nagios process on host E checks the status of the Nagios process on host A, it will find that it is not running. Host E will then have notifications enabled again and take over all responsibilities of notifying contacts of problems.

The exact amount of time that neither host is monitoring the network is hard to determine. Obviously, this period can be minimized by increasing the frequency of service checks (on host E) of the Nagios process on host A. The rest is up to pure chance, but the total "blackout" time shouldn't be too bad.

Scenario 2 - Failover Monitoring

Introduction

Failover monitoring is similiar to, but slightly different than redundant monitoring (as discussed above in scenario 1).

Goals

The basic goal of failover monitoring is to have the Nagios process on the slave host sit idle while the Nagios process on the master host is running. If the process on the master host stops running (or if the host goes down), the Nagios process on the slave host starts monitoring everything.

While the method described in scenario 1 will allow you to continue receive notifications if the master monitoring hosts goes down, it does have some pitfalls. The biggest problem is that the slave host is monitoring the same hosts and servers as the master *at the same time as the master*! This can cause problems with excessive traffic and load on the machines being monitored if you have a lot of services defined. Here's how you can get around that problem...

Initial Program Settings

Disable active service checks and notifications on the slave host using the execute_service_checks and enable_notifications directives. This will prevent the slave host from monitoring hosts and services and sending out notifications while the Nagios process on the master host is still up and running. Make sure you also have the check_external_commands directive enabled on the slave host.

Master Process Check

Set up a cron job on the slave host that periodically (say every minute) runs a script that checks the staus of the Nagios process on the master host (using the *check_nrpe* plugin on the slave host and the nrpe daemon and *check_nagios* plugin on the master host). The script should check the return code of the *check_nrpe plugin* . If it returns a non-OK state, the script should send the appropriate commands to the external command file to enable both notifications and active service checks. If the plugin returns an OK state, the script should send commands to the external command file to disable both notifications and active checks.

Bye doing this you end up with only one process monitoring hosts and services at a time, which is much more efficient that monitoring everything twice.

Also of note, you *don't* need to define host and service handlers as mentioned in scenario 1 because things are handled differently.

Additional Issues

At this point, you have implemented a very basic failover monitoring setup. However, there is one more thing you should consider doing to make things work smoother.

The big problem with the way things have been setup thus far is the fact that the slave host doesn't have the current status of any services or hosts at the time it takes over the job of monitoring. One way to solve this problem is to enable the ocsp command on the master host and have it send all service check results to the slave host using the nsca addon. The slave host will then have up-to-date status information for all services at the time it takes over the job of monitoring things. Since active service checks are not enabled on the slave host, it will not actively run any service checks. However, it will execute host checks if necessary. This means that both the master and slave hosts will be executing host checks as needed, which is not really a big deal since the majority of monitoring deals with service checks.

That's pretty much it as far as setup goes.

Detection and Handling of State Flapping

Introduction

Nagios supports optional detection of hosts and services that are "flapping". Flapping occurs when a service or host changes state too frequently, resulting in a storm of problem and recovery notifications. Flapping can be indicative of configuration problems (i.e. thresholds set too low) or real network problems.

Before I go any futher, let me say that flapping detection has been a little difficult to implement. How exactly does one determine what "too frequently" means in regards to state changes for a particular host or service? When I first started looking into flap detection I tried to find some information on how flapping could/should be detected. After I couldn't find any, I decided to settle with what seemed to be a reasonable solution. The methods by which Nagios detects service and host state flapping are described below...

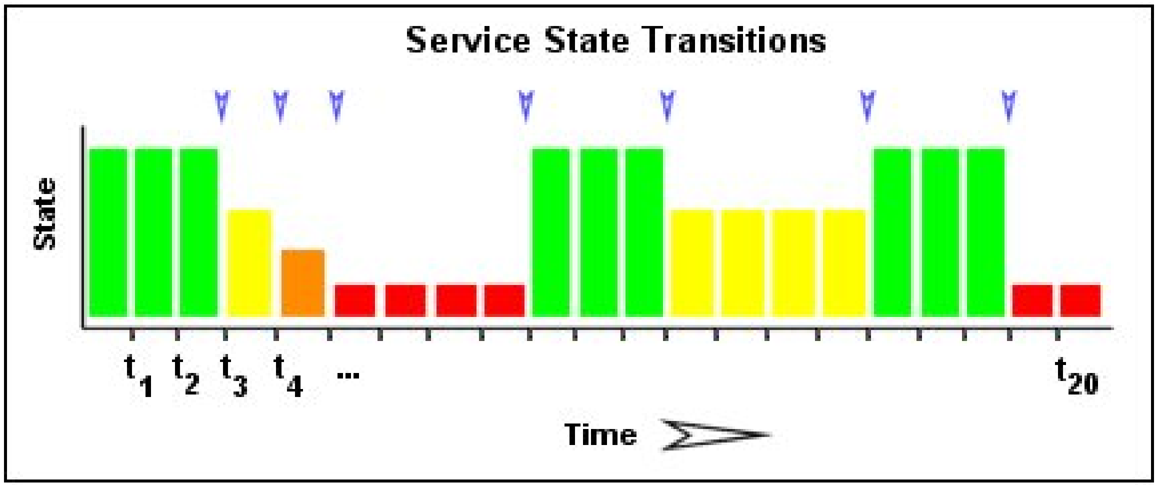
Service Flap Detection

Whenever a service check is performed that results in a hard state or a soft recovery state, Nagios checks to see if the services has started or stopped flapping. It does this by storing the results of the last 21 checks of the service in an array. Older check results in the array are overwritten by newer check results.

The contents of the historical state array are examined (in order from oldest result to newest result) to determine the total percentage of change in state that has occurred during the last 21 service checks. A state change occurs when an archived state is different from the archived state that immediately precedes it in the array. Since we keep the results of the last 21 service checks in the array, there is a possibility of having 20 state changes.

Image 1 below shows a chronological array of service states. OK states are shown in green, WARNING states in yellow, CRITICAL states in red, and UNKNOWN states in orange. Blue arrows have been placed over periods of time where state changes occur.

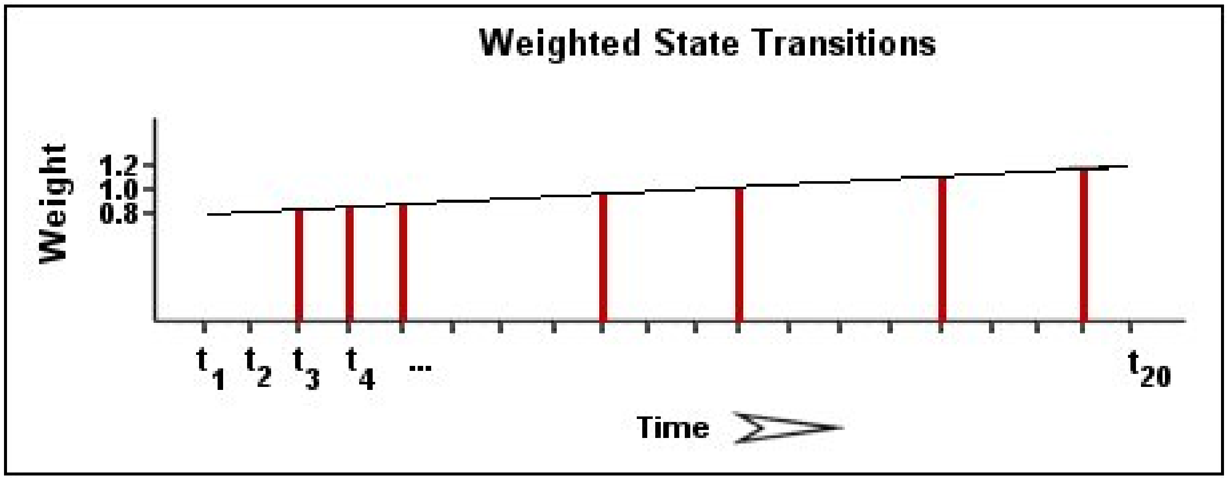
Image 1.



Services that rarely change between states will have a lower total percentage of change than those that do change between states a lot. Since flapping is associated with frequent state changes, we can use the calculated amount of change in state over a period of time (in this case, the last 21 service checks) to determine whether or not a service is flapping. That's not quite good enough though...

It stands to reason that newer state changes should carry more weight than older state changes, so we really need to recalculate the total percent change in state for the service on some sort of curve... To make things simple, I've decided to make the relationship between time and weight linear for calculation of percent state change. The flap detection routines are currently designed to make the newest possible state change carry 50% more weight than the oldest possible state change. Image 2 shows how more recent state changes are given more weight than older state changes when calculating the overall or total percent state change for a particular service. If you really want to see exactly how the weighted calculation is done, look at the code in *base/flapping.c*...

Image 2.



Let's look at a quick example of how flap detection is done. Image 1 above depicts the array of historical service check results for a particular service. The oldest result is on the left and the newest result is on the right. We see that in the example below there were a total of 7 state changes (at t_3, t_4, t_5, t_9, t_{12}, t_{16}, and t_{19}). Without any weighting of the state changes over time, this would give us a total state change of 35% (7 state changes out of a possible 20 state changes). When the individual state changes are weighted relative to the time at which they occurred, the resulting total percent state change for the service is less than 35%. This makes sense since most of the state changes occurred earlier rather than later. Let's just say that the weighted percent of state change turned out to be 31%...

So what significance does the 31% state change have? Well, if the service was previously *not* flapping and 31% is *equal to or greater than* the value specified by the high_service_flap_threshold option in the service definition, Nagios considers the service to have just started flapping. If the service *was* previously flapping and 31% is *less than or equal to* the value specified by the low_service_flap_threshold value in the service definition, Nagios considers the service to have just stopped flapping. If either of those two conditions are not met, Nagios does nothing else with the service, since it is either not currently flapping or it is still flapping...

Host Flap Detection

Host flap detection works in a similiar manner to service flap detection, with one important difference: Nagios will attempt to check to see if a host is flapping whenever the status of the host is checked *and* whenever a service associated with that host is checked. Why is this done? Well, with services we know that the minimum amount of time between consecutive flap detection routines is going to be equal to the service check interval. With hosts, we don't have a check interval, since hosts are not monitored on a regular basis - they are only checked as necessary. A host will be checked for flapping if its state has changed since the last time the flap detection was performed for that host *or* if its state has not changed but at least *x* amount of time has passed since the flap detection was performed. The *x* amount of time is equal to the average check interval of all services associated with the host. That's the best method I could come up with for determining how often flap detection could be performed on a host...

Just as with services, Nagios stores the results of the last 21 of these host checks in an array for the flap detection logic. State changes are weighted based on the time at which they occurred, and the total percent change in state is calculated in the same manner that it is in the service flapping logic.

If a host was previously *not* flapping and its total computed state change percentage is *equal to or greater than* the value specified by the high_host_flap_threshold option, Nagios considers the host to have just started flapping. If the host *was* previously flapping and its total computed state change percentage is *less than or equal to* the value specified by the low_host_flap_threshold value, Nagios considers the host to have just stopped flapping. If either of those two conditions are not met, Nagios does nothing else with the host, since it is either not currently flapping or it is still flapping...

Host- and Service-Specific Flap Detection Thresholds

If you're using the template-based object definition files, you can specify host- and service-specific flap detection thresholds by adding **low_flap_threshold** and **high_flap_threshold** directives to individual host and service definitions. If these directives are *not* present in the host or service definitions, the global host and service flap detection thresholds will be used.

On a similiar note, you can also enable/disable flap detection for specific hosts and services by using the **enable_flap_detection** directive in each object definition. Note that flap detection must be enabled on a program-wide basis (using the enable_flap_detection directive in the main config file) in order for any host or service to have flap detection enabled.

Flap Handling

When a service or host is first detected as flapping, Nagios will do three things:

1. Log a message indicating that the service or host is flapping
2. Add a non-persistent comment to the host or service indicating that it is flapping
3. Supress notifications for the service or host (this is one of the filters in the notification logic)

When a service or host stops flapping, Nagios will do the following:

1. Log a message indicating that the service or host has stopped flapping
2. Delete the comment that was originally added to the service or host when it started flapping

3. Remove the block on notifications for the service or host (notifications will still be bound to the normal notification logic)

Service Check Parallelization

Introduction

One of the features of Nagios is its ability to execute service checks in parallel. This documentation will attempt to explain in detail what that means and how it affects services that you have defined.

How The Parallelization Works

Before I can explain how the service check parallelization works, you first have to understand a bit about how Nagios schedules events. All internal events in Nagios (i.e. log file rotations, external command checks, service checks, etc.) are placed in an event queue. Each item in the event queue has a time at which it is scheduled to be executed. Nagios does its best to ensure that all events get executed when they should, although events may fall behing schedule if Nagios is busy doing other things.

Service checks are one type of event that get scheduled in Nagios' event queue. When it comes time for a service check to be executed, Nagios will kick off another process (using a call to fork()) to go out and run the service check (i.e. a plugin of some sort). Nagios does *not*, however, wait for the service check to finish! Instead, Nagios will immediately go back to servicing other events that reside in the event queue...

So what happens when the service check finishes executing? Well, the process that was started by Nagios to run the service check sends a message back to Nagios containing the results of the service check. It is then up to Nagios to check for and process the results of that service check when it gets a chance.

In order for Nagios to actually do any monitoring, it much process the results of service checks that have finished executing. This is done via a service check "reaper" process. Service "reapers" are another type of event that get scheduled in Nagios' event queue. The frequency of these "reaper" events is determined by the service_reaper_frequency option in the main configuration file. When a "reaper" event is executed, it will check for any messages that contain the result of service checks that have finished executing. These service check results are then handled by the core service monitoring logic. From there Nagios determines whether or not hosts should be checked, notifications should be sent out, etc. When the service check results have been processed, Nagios will reschedule the next check of the service and place it in the event queue for later execution. That completes the service check/monitoring cycle!

For those of you who really want to know, but haven't looked at the code, Nagios uses message queues to handle communication between Nagios and the process that actually runs the service check...

Potential Gotchas...

You should realize that there are potential drawbacks to having service checks parallelized. Since more than one service check may be running at the same time, they have may interfere with one another. You'll have to evaluate what types of service checks you're running and take appropriate steps to guard against any unfriendly outcomes. This is particularly important if you have more than one service check that accesses any hardware (like a modem). Also, if two or more service checks connect to daemon on a remote host to check some information, make sure that daemon can handle multiple simultaneous connections.

Fortunately, there are some things you can do to protect against problems with having some types of service checks "collide"...

1. The easiest thing you can do to prevent service check collisions to to use the service_interleave_factor variable. Interleaving services will help to reduce the load imposed upon remote hosts by service checks. Set the variable to use "smart" interleave factor calculation and then adjust it manually if you find it necessary to do so.
2. The second thing you can do is to set the *max_check_attempts* argument in each service definition to something greater than one. If the service check does happen to collide with another running check, Nagios will retry the service check *max_check_attempts-1* times before notifying anyone of a problem.
3. You could try is to implement some kind of "back-off and retry" logic in the actual service check code, although you may find it difficult or too time-consuming
4. If all else fails you can effectively prevent service checks from being parallelized by setting the max_concurrent_checks option to 1. This will allow only one service to be checked at a time, so it isn't a spectacular solution. If there is enough demand, I will add an option to the service definitions which will allow you to specify on a per-service basis whether or not a service check can be parallelized. If there isn't enough demand, I won't...

One other thing to note is the effect that parallelization of service checks can have on system resources on the machine that runs Nagios. Running a lot of service checks in parallel can be taxing on the CPU and memory. The inter_check_delay_method will attempt to minimize the load imposed on your machine by spreading the checks out evenly over time (if you use the "smart" method), but it isn't a surefire solution. In order to have some control over how many service checks can be run at any given time, use the max_concurrent_checks variable. You'll have to tweak this value based on the total number of services you check, the system resources you have available (CPU speed, memory, etc.), and other processes which are running on your machine. For more information on how to tweak the *max_concurrent_checks* variable for your setup, read the documentation on check scheduling.

What Isn't Parallelized

It is important to remember that only the *execution* of service checks has been parallelized. There is good reason for this - other things cannot be parallelized in a very safe or sane manner. In particular, event handlers, contact notifications, processing of service checks, and host checks are *not* parallelized. Here's why...

Event handlers are not parallelized because of what they are designed to do. Much of the power of event handlers comes from the ability to do proactive problem resultion. An example of this is restarting the web server when the HTTP service on the local machine is detected as being down. In order to prevent more than one event handler from trying to "fix" problems in parallel (without any knowledge of what each other is doing), I have decided to not parallelize them.

Contact notifications are not parallelized because of potential notification methods you may be using. If, for example, a contact notification uses a modem to dial out and send a message to your pager, it requires exclusive access to the modem while the notification is in progress. If two or more such notifications were being executed in parallel, all but one would fail because the others could not get access to the modem. There are ways to get around this, like providing some kind of "back-off and retry" method in the notification script, but I've decided not to rely on users having implemented this type of feature in their scripts. One quick note - if you have service checks which use a modem, make sure that any notification scripts that dial out have some method of retrying access to the modem. This is necessary because a service check may be running at the same time a notification is!

Processing of service check results has not been parallelized. This has been done to prevent situations where multiple notifications about host problems or recoveries may be sent out if a host goes down, becomes unreachable, or recovers.

Notification Escalations

Introduction

Nagios supports *optional* escalation of contact notifications for hosts and services. I'll explain quickly how they work, although they should be fairly self-explanatory...

Service Notification Escalations

Escalation of service notifications is accomplished by defining service escalation definitions in your object configuration file. Service escalation definitions are used to escalate notifications for a particular service.

Host Notification Escalations

Escalation of host notifications is accomplished by defining either host or hostgroup escalation definitions in your object configuration file. Host escalation definitions are used to escalate notifications for specific hosts, while hostgroup escalation definitions are used to escalate notifications for all hosts in a particular hostgroup. The examples I provide below all use service escalation definitions, but host and hostgroup escalations work the same way (except for the fact that they are used for host notifications and not service notifications).

When Are Notifications Escalated?

Notifications are escalated *if and only if* one or more escalation definitions matches the current notification that is being sent out. If a host or service notification *does not* have any valid escalation definitions that applies to it, the contact group(s) specified in either the host group or service definition will be used for the notification. Look at the example below:

```
define serviceescalation{
        host_name               webserver
        service_description     HTTP
        first_notification      3
        last_notification       5
        notification_interval   90
        contact_groups          nt-admins,managers
        }

define serviceescalation{
        host_name               webserver
        service_description     HTTP
        first_notification      6
        last_notification       10
        notification_interval   60
        contact_groups          nt-admins,managers,everyone
        }
```

Notice that there are "holes" in the notification escalation definitions. In particular, notifications 1 and 2 are not handled by the escalations, nor are any notifications beyond 10. For the first and second notification, as well as all notifications beyond the tenth one, the *default* contact groups specified in the service definition are used. For all the examples I'll be using, I'll be assuming that the default contact groups for the service definition is called *nt-admins*.

Contact Groups

When defining notification escalations, it is important to keep in mind that any contact groups that were members of "lower" escalations (i.e. those with lower notification number ranges) should also be included in "higher" escalation definitions. This should be done to ensure that anyone who gets notified of a problem *continues* to get notified as the problem is escalated. Example:

```
define serviceescalation{
        host_name               webserver
        service_description     HTTP
        first_notification      3
        last_notification       5
        notification_interval   90
        contact_groups          nt-admins,managers
        }

define serviceescalation{
        host_name               webserver
        service_description     HTTP
        first_notification      6
        last_notification       0
        notification_interval   60
        contact_groups          nt-admins,managers,everyone
        }
```

The first (or "lowest") escalation level includes both the *nt-admins* and *managers* contact groups. The last (or "highest") escalation level includes the *nt-admins*, *managers*, and *everyone* contact groups. Notice that the *nt-admins* contact group is included in both escalation definitions. This is done so that they continue to get paged if there are still problems after the first two service notifications are sent out. The *managers* contact group first appears in the "lower" escalation definition - they are first notified when the third problem notification gets sent out. We want the *managers* group to continue to be notified if the problem continues past five notifications, so they are also included in the "higher" escalation definition.

Overlapping Escalation Ranges

Notification escalation definitions can have notification ranges that overlap. Take the following example:

```
define serviceescalation{
        host_name               webserver
        service_description     HTTP
        first_notification      3
        last_notification       5
        notification_interval   20
        contact_groups          nt-admins,managers
        }

define serviceescalation{
        host_name               webserver
        service_description     HTTP
        first_notification      4
        last_notification       0
        notification_interval   30
        contact_groups          on-call-support
        }
```

In the example above:

- The *nt-admins* and *managers* contact groups get notified on the third notification
- All three contact groups get notified on the fourth and fifth notifications
- Only the *on-call-support* contact group gets notified on the sixth (or higher) notification

Recovery Notifications

Recovery notifications are slightly different than problem notifications when it comes to escalations. Take the following example:

```
define serviceescalation{
        host_name               webserver
        service_description     HTTP
        first_notification      3
        last_notification       5
        notification_interval   20
        contact_groups          nt-admins,managers
        }

define serviceescalation{
        host_name               webserver
        service_description     HTTP
        first_notification      4
        last_notification       0
        notification_interval   30
        contact_groups          on-call-support
        }
```

If, after three problem notifications, a recovery notification is sent out for the service, who gets notified? The recovery is actually the fourth notification that gets sent out. However, the escalation code is smart enough to realize that only those people who were notified about the problem on the third notification should be notified about the recovery. In this case, the *nt-admins* and *managers* contact groups would be notified of the recovery.

Notification Intervals

You can change the frequency at which escalated notifications are sent out for a particular host or service by using the *notification_interval* option of the hostgroup or service escalation definition. Example:

```
define serviceescalation{
        host_name               webserver
        service_description     HTTP
        first_notification      3
        last_notification       5
        notification_interval   45
        contact_groups          nt-admins,managers
        }

define serviceescalation{
        host_name               webserver
        service_description     HTTP
        first_notification      6
        last_notification       0
        notification_interval   60
        contact_groups          nt-admins,managers,everyone
        }
```

In this example we see that the default notification interval for the services is 240 minutes (this is the value in the service definition). When the service notification is escalated on the 3rd, 4th, and 5th notifications, an interval of 45 minutes will be used between notifications. On the 6th and subsequent notifications, the notification interval will be 60 minutes, as specified in the second escalation definition.

Since it is possible to have overlapping escalation definitions for a particular hostgroup or service, and the fact that a host can be a member of multiple hostgroups, Nagios has to make a decision on what to do as far as the notification interval is concerned when escalation definitions overlap. In any case where there are multiple valid escalation definitions for a particular notification, Nagios will choose the smallest notification interval. Take the following example:

```
define serviceescalation{
        host_name               webserver
        service_description     HTTP
        first_notification      3
        last_notification       5
        notification_interval   45
        contact_groups          nt-admins,managers
        }

define serviceescalation{
        host_name               webserver
        service_description     HTTP
        first_notification      4
        last_notification       0
        notification_interval   60
        contact_groups          nt-admins,managers,everyone
        }
```

We see that the two escalation definitions overlap on the 4th and 5th notifications. For these notifications, Nagios will use a notification interval of 45 minutes, since it is the smallest interval present in any valid escalation definitions for those notifications.

One last note about notification intervals deals with intervals of 0. An interval of 0 means that Nagios should only sent a notification out for the first valid notification during that escalation definition. All subsequent notifications for the hostgroup or service will be suppressed. Take this example:

```
define serviceescalation{
        host_name               webserver
        service_description     HTTP
        first_notification      3
        last_notification       5
        notification_interval   45
        contact_groups          nt-admins,managers
        }

define serviceescalation{
        host_name               webserver
        service_description     HTTP
        first_notification      4
        last_notification       6
        notification_interval   0
        contact_groups          nt-admins,managers,everyone
        }

define serviceescalation{
        host_name               webserver
        service_description     HTTP
        first_notification      7
        last_notification       0
        notification_interval   30
        contact_groups          nt-admins,managers
        }
```

In the example above, the maximum number of problem notifications that could be sent out about the service would be four. This is because the notification interval of 0 in the second escalation definition indicates that only one notification should be sent out (starting with and including the 4th notification) and all subsequent notifications should be repressed. Because of this, the third service escalation definition has no effect whatsoever, as there will never be more than four notifications.

Monitoring Service and Host Clusters

Introduction

Several people have asked how to go about monitoring clusters of hosts or services, so I decided to write up a little documentation on how to do this. Its fairly straightforward, so hopefully you find things easy to understand...

First off, we need to define what we mean by a "cluster". The simplest way to understand this is with an example. Let's say that your organization has five hosts which provide redundant DNS services to your organization. If one of them fails, its not a major catastrophe because the remaining servers will continue to provide name resolution services. If you're concerned with monitoring the availability of DNS service to your organization, you will want to monitor five DNS servers. This is what I consider to be a *service* cluster. The service cluster consists of five separate DNS services that you are monitoring. Although you do want to monitor each individual service, your main concern is with the overall status of the DNS service cluster, rather than the availability of any one particular service.

If your organization has a group of hosts that provide a high-availability (clustering) solution, I would consider those to be a *host* cluster. If one particular host fails, another will step in to take over all the duties of the failed server. As a side note, check out the High-Availability Linux Project for information on providing host and service redundancy with Linux.

Plan of Attack

There are several ways you could potentially monitor service or host clusters. I'll describe the method that I believe to be the easiest. Monitoring service or host clusters involves two things:

- Monitoring individual cluster elements
- Monitoring the cluster as a collective entity

Monitoring individual host or service cluster elements is easier than you think. In fact, you're probably already doing it. For service clusters, just make sure that you are monitoring each service element of the cluster. If you've got a cluster of five DNS servers, make sure you have five separate service definitions (probably using the *check_dns* plugin). For host clusters, make sure you have configured appropriate host definitions for each member of the cluster (you'll also have to define at least one service to be monitored for each of the hosts). **Important:** You're going to want to disable notifications for the individual cluster elements (host or service definitions). Even though no notifications will be sent about the individual elements, you'll still get a visual display of the individual host or service status in the status CGI. This will be useful for pinpointing the source of problems within the cluster in the future.

Monitoring the overall cluster can be done by using the previously cached results of cluster elements. Although you could re-check all elements of the cluster to determine the cluster's status, why waste bandwidth and resources when you already have the results cached? Where are the results cached? Cached results for cluster elements can be found in the status file (assuming you are monitoring each element). The *check_cluster* plugin is designed specifically for checking cached host and service states in the status file. **Important:** Although you didn't enable notifications for individual elements of the cluster, you will want them enabled for the overall cluster status check.

Using the *check_cluster* Plugin

The check_cluster plugin is designed to check the overall status of a host or service cluster. It works by checking the cached status information of individual host or service cluster elements in the status file.

More to come... The check_cluster plugin can temporarily be obtained from http://www.nagios.org/download/alpha.

Monitoring Service Clusters

First off, you're going to have to define a new service for monitoring the cluster. This service will perform the check of the overall status of the cluster. You are probably going to want to have notifications enabled for this service so you know when there are problems that need to be looked at. You probably don't care so much about the status of any one of the services that are members of the cluster, so you can disable notifications in those those service definitions.

Okay, let's assume that you have a *check_service_cluster* command defined as follows:

```
define command{
        command_name    check_service_cluster
        command_line    /usr/local/nagios/libexec/check_cluster --service
/usr/local/nagios/var/status.log $ARG1$ $ARG2$ < $ARG3$
        }
```

Let's say you have five services that are members of the service cluster. If you want Nagios to generate a warning alert if two or more services in the cluster and in a non-ok state or a critical alert if three or more are in a non-ok state, the <*check_command*> argument of the service you define to monitor the cluster looks something like this:

```
check_service_cluster!2!3!/usr/local/nagios/etc/servicecluster.cfg
```

The $ARG3$ macro will be replaced with */usr/local/nagios/etc/servicecluster.cfg* when the check is made. Since this is the file from which the *check_cluster* plugin will read the names of cluster members, you'll need to create that file and add the services that are members (one per line). The format of a service entry is the short name of the host the service is associated with, followed by a semi-colon, and then the service description. An example of the file contents would be as follows:

```
host1;DNS Service
host2;DNS Service
host3;DNS Service
host4;DNS Service
host5;DNS Service
host6;DNS Service
```

Monitoring Host Clusters

Monitoring host clusters is very similiar to monitoring service clusters. Obviously, the main difference is that the cluster members are hosts and not services. In order to monitor the status of a host cluster, you must define a service that uses the *check_cluster* plugin. The service should *not* be associated with any of the hosts in the cluster, as this will cause problems with notifications for the cluster if that host goes down. A good idea might be to associate the service with the host that Nagios is running on. After all, if the host that Nagios is running on goes down, then Nagios isn't running anymore, so there isn't anything you can do as far as monitoring (unless you've setup redundant monitoring hosts)...

Anyway, let's assume that you have a *check_host_cluster* command defined as follows:

```
define command{
        command_name    check_host_cluster
        command_line    /usr/local/nagios/libexec/check_cluster --host $ARG1$ $ARG2$
/usr/local/nagios/var/status.log < $ARG3$
        }
```

Let's say you have six hosts in the host cluster. If you want Nagios to generate a warning alert if two or more hosts in the cluster are not up or a critical alert if four or more hosts are not up, the <*check_command*> argument of the service you define to monitor the cluster looks something like this:

check_host_cluster!2!4!/usr/local/nagios/etc/hostcluster.cfg

The $ARG3$ macro will be replaced with */usr/local/nagios/etc/hostcluster.cfg* when the check is made. Since this is the file from which the *check_cluster* plugin will read the names of cluster members, you'll need to create that file and add the short names of all hosts (as they were defined in your host definitions) that are members (one per line). An example of the file contents would be as follows:

```
host1
host2
host3
host4
host5
host6
```

That's it! Nagios will periodically check the status of the host cluster and send notifications to you when its status is degraded (assuming you've enabled notification for the service). Note that for thehost definitions of each cluster member, you will most likely want to disable notifications when the host goes down . Remeber that you don't care as much about the status of any individual host as you do the overall status of the cluster. Depending on your network layout and what you're trying to accomplish, you may wish to leave notifications for unreachable states enabled for the host definitions.

Host and Service Dependencies

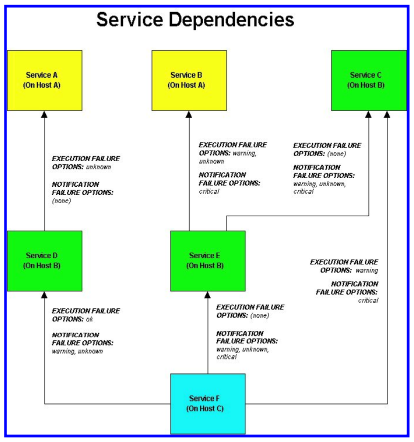
Introduction

Service and host dependencies are an *advanced* feature that allow you to control the behavior of hosts and services based on the status of one or more other hosts or services. I'll explain how dependencies work, along with the differences between host and service dependencies.

Service Dependencies Overview

The image below shows an example logical layout of service dependencies. There are a few things you should notice:

1. A service can be dependent on one or more other services
2. A service can be dependent on services which are not associated with the same host
3. Service dependencies are not inherited
4. Service dependencies can be used to cause service execution and service notifications to be suppressed under different circumstances (OK, WARNING, UNKNOWN, and/or CRITICAL states)


Service Dependencies
Service A (On Host A)
Service B (On Host A)
Service C (On Host B)
EXECUTION FAILURE OPTIONS: unknown
NOTIFICATION FAILURE OPTIONS: (none)
EXECUTION FAILURE OPTIONS: warning, unknown
NOTIFICATION FAILURE OPTIONS: critical
EXECUTION FAILURE OPTIONS: (none)
NOTIFICATION FAILURE OPTIONS: warning, unknown, critical
Service D (On Host B)
Service E (On Host B)
EXECUTION FAILURE OPTIONS: warning
NOTIFICATION FAILURE OPTIONS: critical
EXECUTION FAILURE OPTIONS: ok
NOTIFICATION FAILURE OPTIONS: warning, unknown
EXECUTION FAILURE OPTIONS: (none)
NOTIFICATION FAILURE OPTIONS: warning, unknown, critical
Service F (On Host C)

Defining Service Dependencies

First, the basics. You create service dependencies by adding service dependency definitions in your object config file(s). In each definition you specify the *dependent* service, the service you are *depending on*, and the criteria (if any) that cause the execution and notification dependencies to fail (these are described later).

You can create several dependencies for a given service, but you must add a separate service dependency definition for each dependency you create.

In the image above, the dependency definitions for *Service F* on *Host C* would be defined as follows:

```
define servicedependency{
        host_name                       Host B
        service_description             Service D
        dependent_host_name             Host C
        dependent_service_description   Service F
        execution_failure_criteria      o
        notification_failure_criteria   n
        }

define servicedependency{
        host_name                       Host B
        service_description             Service E
        dependent_host_name             Host C
        dependent_service_description   Service F
        execution_failure_criteria      n
        notification_failure_criteria   w,u,c
        }

define servicedependency{
        host_name                       Host B
        service_description             Service C
        dependent_host_name             Host C
        dependent_service_description   Service F
        execution_failure_criteria      w
        notification_failure_criteria   c
        }
```

How Service Dependencies Are Tested

Before Nagios executes a service check or sends notifications out for a service, it will check to see if the service has any dependencies. If it doesn't have any dependencies, the check is executed or the notification is sent out as it normally would be. If the service *does* have one or more dependencies, Nagios will check each dependency entry as follows:

1. Nagios gets the current status* of the service that is being *depended upon*.
2. Nagios compares the current status of the service that is being *depended upon* against either the execution or notification failure options in the dependency definition (whichever one is relevant at the time).
3. If the current status of the service that is being *depended upon* matches one of the failure options, the dependency is said to have failed and Nagios will break out of the dependency check loop.
4. If the current state of the service that is being *depended upon* does not match any of the failure options for the dependency entry, the dependency is said to have passed and Nagios will go on and check the next dependency entry.

This cycle continues until either all dependencies for the service have been checked or until one dependency check fails.

*One important thing to note is that by default, Nagios will use the most current hard state of the service(s) that is/are being depended upon when it does the dependeny checks. If you want Nagios to use the most current state of the services (regardless of whether its a soft or hard state), enable the soft_service_dependencies option.

Service Execution Dependencies

If *all* of the execution dependency tests for the service *passed*, Nagios will execute the check of the service as it normally would. If even just one of the execution dependencies for a service fails, Nagios will temporarily prevent the execution of checks for that (dependent) service. At some point in the future the execution dependency tests for the service may all pass. If this happens, Nagios will start checking the service again as it normally would. More information on the check scheduling logic can be found here.

In the example above, **Service E** would have failed execution dependencies if **Service B** is in a WARNING or UNKNOWN state. If this was the case, the service check would not be performed and the check would be scheduled for (potential) execution at a later time.

Service Notification Dependencies

If *all* of the notification dependency tests for the service *passed*, Nagios will send notifications out for the service as it normally would. If even just one of the notification dependencies for a service fails, Nagios will temporarily repress notifications for that (dependent) service. At some point in the future the notification dependency tests for the service may all pass. If this happens, Nagios will start sending out notifications again as it normally would for the service. More information on the notification logic can be found here.

In the example above, **Service F** would have failed notification dependencies if **Service C** is in a CRITICAL state, *and/or* **Service D** is in a WARNING or UNKNOWN state, *and/or* if **Service E** is in a WARNING, UNKNOWN, or CRITICAL state. If this were the case, notifications for the service would not be sent out.

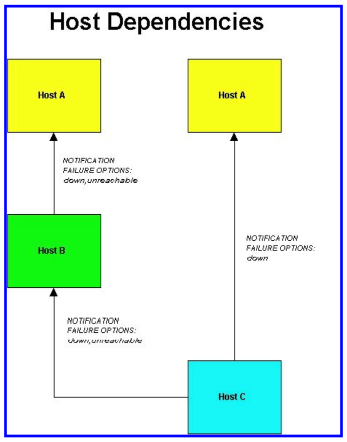
Service Dependency Inheritance

As mentioned before, service dependencies are *not* inherited. In the example above you can see that Service F is dependent on Service E. However, it does not automatically inherit Service E's dependencies on Service B and Service C. In order to make Service F dependent on Service C we had to add another service dependency definition. There is no dependency definition for Service B, so Service F is *not* dependent on Service B. In some cases the lack of inheritance means you're going to have to add some additional dependency definitions in your config file, but I think it makes things much more flexible. For instance, in the example above we might have good reason for not making Service F dependent on Service B. If dependencies were automatically inherited, this would not be possible.

Host Dependencies

As you'd probably expect, host dependencies work in a similiar fashion to service dependencies. The big difference is that they're for hosts, not services. Another difference is that host dependencies only work for suppressing host notifications, not host checks.

The image below shows an example of the logical layout of host dependencies.


Host Dependencies
Host A
Host A
NOTIFICATION FAILURE OPTIONS: down,unreachable
Host B
NOTIFICATION FAILURE OPTIONS: down
NOTIFICATION FAILURE OPTIONS: down,unreachable
Host C

In the image above, the dependency definitions for *Host C* would be defined as follows:

```
define hostdependency{
        host_name                       Host A
        dependent_host_name             Host C
        notification_failure_criteria   d
        }

define hostdependency{
        host_name                       Host B
        dependent_host_name             Host C
        notification_failure_criteria   d,u
        }
```

As with service dependencies, host dependencies are not inherited. In the example image you can see that Host C does not inherit the host dependencies of Host B. In order for Host C to be dependent on Host A, a new host dependency definition must be defined.

Host notification dependencies work in a similiar manner to service dependencies. If *all* of the notification dependency tests for the host *pass*, Nagios will send notifications out for the host as it normally would. If even just one of the notification dependencies for a host fails, Nagios will temporarily repress notifications for that (dependent) host. At some point in the future the notification dependency tests for the host may all pass. If this happens, Nagios will start sending out notifications again as it normally would for the host. More information on the notification logic can be found here.

State Stalking

Introduction

State "stalking" is a feature which is probably not going to used by most users. When enabled, it allows you to log changes in service and host checks even if the state of the host or service does not change. When stalking is enabled for a particular host or service, Nagios will watch that service very carefully and log any changes it sees. As you'll see, it can be very helpful to you in later analysis of the log files.

How Does It Work?

Under normal circumstances, the result of a host or service check is only logged if the host or service has changed state since it was last checked. There are a few exceptions to this, but for the most part, that's the rule.

If you enable stalking for one or more states of a particular host or service, Nagios will log the results of the host or service check if the output from the check differs from the output from the previous check. Take the following example of eight consecutive checks of a service:

Service Check #:	Service State:	Service Check Output:
x	OK	RAID array optimal
x+1	OK	RAID array optimal
x+2	WARNING	RAID array degraded (1 drive bad, 1 hot spare rebuilding)
x+3	CRITICAL	RAID array degraded (2 drives bad, 1 host spare online, 1 hot spare rebuilding)
x+4	CRIICAL	RAID array degraded (3 drives bad, 2 hot spares online)
x+5	CRITICAL	RAID array failed
x+6	CRITICAL	RAID array failed
x+7	CRITICAL	RAID array failed

Given this sequence of checks, you would normally only see two log entries for this catastrophe. The first one would occur at service check x+2 when the service changed from an OK state to a WARNING state. The second log entry would occur at service check x+3 when the service changed from a WARNING state to a CRITICAL state.

For whatever reason, you may like to have the complete history of this catasrophe in your log files. Perhaps to help explain to your manager how quickly the situation got out of control, perhaps just to laugh at over a couple of drinks at the local pub, whatever...

Well, if you had enabled stalking of this service for CRITICAL states, you would have events at x+4 and x+5 logged in addition to the events at x+2 and x+3. Why is this? With state stalking enabled, Nagios would have examined the output from each service check to see if it differed from the output of the previous check. If the output differed and the state of the service didn't change between the two checks, the result of the newer service check would get logged.

A similiar example of stalking might be on a service that checks your web server. If the check_http plugin first returns a WARNING state because of a 404 error and on subsequent checks returns a WARNING state because of a particular pattern not being found, you might want to know that. If you didn't enable state stalking for WARNING states of the service, only the first WARNING state event (the 404 error) would be logged and you wouldn't have any idea (looking back in the archived logs) that future problems were not due to a 404, but rather a missing pattern in the returned web page.

Should I Enable Stalking?

First, you must decide if you have a real need to analyze archived log data to find the exact cause of a problem. You may decide you need this feature for some hosts or services, but not for all. You may also find that you only have a need to enable stalking for some host or service states, rather than all of them. For example, you may decide to enable stalking for WARNING and CRITICAL states of a service, but not for OK and UNKNOWN states.

The decision to to enable state stalking for a particular host or service will also depend on the plugin that you use to check that host or service. If the plugin always returns the same text output for a particular state, there is no reason to enable stalking for that state.

How Do I Enable Stalking?

You can enable state stalking for hosts and services by using the *stalking_options* directive in host and service definitions. This directive is currently only supported in the template-based config file format.

Caveats

You should be aware that there are some potential pitfalls with enabling stalking. These all relate to the reporting functions found in various CGIs (histogram, alert summary, etc.). Because state stalking will cause additional alert entries to be logged, the data produced by the reports will show evidence of inflated numbers of alerts.

As a general rule, I would suggest that you *not* enable stalking for hosts and services without thinking things through. Still, its there if you need and want it.

Performance Data

Introduction

Nagios is designed to allow plugins to return optional performance data in addition to normal status data, as well as allow you to pass that performance data to external applications for processing. A description of the different types of performance data, as well as information on how to go about processing that data is described below...

Types of Performance Data

There are two basic categories of performance data that can be obtained from Nagios:

1. **Check performance data**
2. **Plugin performance data**

Check performance data is internal data that relates to the actual execution of a host or service check. This might include things like service check latency (i.e. how "late" was the service check from its scheduled execution time) and the number of seconds a host or service check took to execute. This type of performance data is available for all checks that are performed. The $EXECUTIONTIME$ macro can be used to determine the number of seconds a host or service check was running and the $LATENCY$ macro can be used to determine how "late" a service check was (host checks have zero latency, as they are executed on an as-needed basis, rather than at regularly scheduled intervals).

Plugin performance data is external data specific to the plugin used to perform the host or service check. Plugin-specific data can include things like percent packet loss, free disk space, processor load, number of current users, etc. - basically any type of metric that the plugin is measuring when it executes. Plugin-specific performance data is optional and may not be supported by all plugins. As of this writing, no plugins return performance data, although they mostly likely will in the near future. Plugin-specific performance data (if available) can be obtained by using the $PERFDATA$ macro. See below for more information on how plugins can return performance data to Nagios for inclusion in the $PERFDATA$ macro.

Performance Data Support For Plugins

Normally plugins return a single line of text that indicates the status of some type of measurable data. For example, the check_ping plugin might return a line of text like the following:

```
PING ok - Packet loss = 0%, RTA = 0.80 ms
```

With this type of output, the entire line of text is available in the $OUTPUT$ macro.

In order to facilitate the passing of plugin-specific performance data to Nagios, the plugin specification has been expanded. If a plugin wishes to pass performance data back to Nagios, it does so by sending the normal text string that it usually would, followed by a pipe character (|), and then a string containing one or more performance data metrics. Let's take the check_ping plugin as an example and assume that it has been enhanced to return percent packet loss and average round trip time as performance data metrics. A sample plugin output might look like this:

```
PING ok - Packet loss = 0%, RTA = 0.80 ms | percent_packet_loss=0, rta=0.80
```

When Nagios seems this format of plugin output it will split the output into two parts: everything before the pipe character is considered to be the "normal" plugin output and everything after the pipe character is considered to be the plugin-specific performance data. The "normal" output gets stored in the $OUTPUT$ macro, while the optional performance data gets stored in the $PERFDATA$ macro. In the example above, the $OUTPUT$ macro would contain "*PING ok - Packet loss = 0%, RTA = 0.80 ms*" (without quotes) and the $PERFDATA$ macro would contain "*percent_packet_loss=0, rta=0.80*" (without quotes).

Enabling Performance Data Processing

If you want to process the performance data that is available from Nagios and the plugins, you'll need to do three things.

First, you'll have to enable the process_performance_data option in the main config file.

Second, you'll have to compile Nagios with the proper type of performance data processing. There are currently two options for this:

- **Default method** - Nagios will launch a command you define in order to process the data. This method is the most flexible, but consumes more system resources as it requires Nagios to fork a new system process in order to handle the performance data.
- **File-based method** - Performance data is dumped directly into one or more files in a manner of your choosing. You simply define a template to be used in writing the data and Nagios will dump performance data to the files in that format. This is less flexible that the default method, but requires far less system resources and is much faster.

Lastly, you'll need to add any necessary directives and command definitions to your config files to start using performance data. The exact items you'll need to add depend on what type of performance data processing you've compiled Nagios with. Follow the link to appropriate option mentioned above to find out what you need to do.

Post-Processing Options

I'm assuming that you're going to want to do some post-processing of the performance data that you get out of Nagios. If not, why are you enabling performance data processing in the first place?

What you do with the performance data once its out of Nagios is completely up to you. If you are simply writing performance data to text files, you could setup an occassional cron job to process the entries in those files, squash them using rrdtool, dump them into a database, produce graphs, whatever...

Scheduled Downtime

Introduction

Nagios allows you to schedule periods of planned downtime for hosts and service that you're monitoring. This is useful in the event that you actually know you're going to be taking a server down for an upgrade, etc. When a host a service is in a period of scheduled downtime, notifications for that host or service will be suppressed.

Downtime File

Scheduled host and service downtime is stored in the file you specify by the downtime_file directive in your main configuration file.

Downtime Retention

Scheduled host and service downtime is automatically preserved across program restarts. When Nagios starts up, it will scan the downtime file, delete any old or invalid entries, and schedule downtime for all valid host and service entries.

Scheduling Downtime

You can schedule downtime for hosts and service through the extinfo CGI (either when viewing host or service information). Click in the "Schedule downtime for this host/service" link to actually schedule the downtime.

Once you schedule downtime for a host or service, Nagios will add a comment to that host/service indicating that it is scheduled for downtime during the period of time you indicated. When that period of downtime passes, Nagios will automatically delete the comment that it added. Nice, huh?

Types of Scheduled Downtime

There are two types of scheduled downtime - "fixed" and "flexible". When you schedule downtime for a host or service through the web interface you'll be asked if the downtime is fixed or not. Here's an explanation of how "fixed" and "flexible" downtime differs:

"Fixed" downtime starts and stops at the exact start and end times that you specify when you schedule it. Okay, that was easy enough...

"Flexible" downtime is intended for times when you know that a host or service is going to be down for X minutes (or hours), but you don't know exactly when that'll start. When you schedule flexible downtime, Nagios will start the scheduled downtime sometime between the start and end times you specified. The downtime will last for as long as the duration you specified when you scheduled the downtime. This assumes that the host or service for which you scheduled flexible downtime either goes down (or becomes unreachable) or goes into a non-OK state sometime between the start and end times you specified. The time at which a host or service transitions to a problem state determines the time at which Nagios actually starts the downtime. The downtime will then last for the duration you specified, even if the host or service recovers before the downtime expires. This is done for a very good reason. As we all know, you can think you've got a problem fixed (and restart a server) ten times before it actually works right. Smart, eh?

How Scheduled Downtime Affects Notifications

When a host or service is in a period of scheduled downtime, Nagios will not allow notifications to be sent out for the host or service. suppression of notifications is accomplished by adding an additional filter to the notification logic. You will *not* see an icon in the CGIs indicating that notifications for that host/service are disabled. When the scheduled downtime has passed, Nagios will allow notifications to be sent out for the host or service as it normally would.

Overlapping Scheduled Downtime

I like to refer to this as the "Oh crap, its not working" syndrome. You know what I'm talking about. You take a server down to perform a "routine" hardware upgrade, only to later realize that the OS drivers aren't working, the RAID array blew up, or the drive imaging failed and left your original disks useless to the world. Moral of the story is that any routine work on a server is quite likely to take three or four times as long as you had originally planned...

Let's take the following scenario:

1. You schedule downtime for host A from 7:30pm-9:30pm on a Monday
2. You bring the server down about 7:45pm Monday evening to start a hard drive upgrade
3. After wasting an hour and a half battling with SCSI errors and driver incompatabilities, you finally get the machine to boot up
4. At 9:15 you realize that one of your partitions is either hosed or doesn't seem to exist anywhere on the drive
5. Knowing you're in for a long night, you go back and schedule additional downtime for host A from 9:20pm Monday evening to 1:30am Tuesday Morning.

If you schedule overlapping periods of downtime for a host or service (in this case the periods were 7:40pm-9:30pm and 9:20pm-1:30am), Nagios will wait until the last period of scheduled downtime is over before it allows notifications to be sent out for that host or service. In this example notifications would be suppressed for host A until 1:30am Tuesday morning.

Database Support

(MySQL and PostgreSQL)

Index

Introduction

This will explain how to optionally compile both the core program and the CGIs so that they *natively* support storage of various types of data in one or more databases. Currently only MySQL and PostgreSQL databases are supported, although more may be supported in the future.

Out With The Old...

Okay, before we go ahead and get into the details of the database integration stuff, you need to understand something. The default method for storing status data, comments, etc. in Nagios is (and probably will continue to be) in plain old text files. The standard files used by the default external data routines include the status file, downtime file, comment file, and the state_retention file. With the default install, extended host and service information is not stored in its own file, but in extended host and service information definitions in the CGI configuration file.

Assuming you plan on using a database to store some or all external data, a few things are obviously going to change. Data will no longer be stored in text files, but rather in one or more databases. Since I don't feel like rewriting a lot of documentation, you're going to have to make a mental transition. You'll need to realize that status information is no longer stored in the status log, but rather in a few tables in a database somewhere. Same thing applies for other types of external data (downtime data, comments, retention information, and extended host information).

Getting Started

First off, I assume you've got a MySQL or PostgreSQL database server up and running on your network somewhere and you've got the appropriate client libraries installed on the same machine where you're going to compile and run Nagios. I'm also assumimg you're familiar with creating databases and tables and managing accounts and security in the particular database system(s) you're going to use. If you're not, go out and learn before you attempt to compile Nagios with database support.

Very Important Note: Once you (re)run the configure script to add support for database storage (as will be described below), make sure you recompile *both* the core program and *all* the CGIs (using the **make all** command)!!

Compiling With MySQL Support

In order to support storage of various types of data in MySQL, you're going to have to supply one or more options to the configure script.

You have a few options here. First, you need to decide what data you want to keep in MySQL and what (if any) you want to leave in the older format (text files). Use the table below to determine what options you'll need to supply to the configure script once you determine your needs. **Note:** MySQL support for storage of object data (service, host, and command definitions, etc) is not yet supported.

Data Type	Configure Script Option	Comments
All types	--with-mysql-xdata	This will compile in MySQL support for all types of external data (downtime data, comment data, status data, retention data, and extended data). Support for object data (service and host definitions, etc.) is as of yet non-existent.
Comment data	--with-mysql-downtime	This will compile in MySQL support for downtime data (it will replace the standard downtime file)
Comment data	--with-mysql-comments	This will compile in MySQL support for comment data (it will replace the standard comment file)
Status data	--with-mysql-status	This will compile in MySQL support for status data (it will replace the standard status log)

Retention data	--with-mysql-retention	This will compile in MySQL support for retention data (it will replace the standard state_retention file)
Extended data	--with-mysql-extinfo	This will compile in MySQL support for extended data (it will replace the standard hostextinfo[] and serviceextinfo[] definitions in the CGI config file)

Compiling With PostgreSQL Support

In order to support storage of various types of data in PostgreSQL, you're going to have to supply one or more options to the configure script.

You have a few options here. First, you need to decide what data you want to keep in PostgreSQL and what (if any) you want to leave in the older format (text files) or possibly in MySQL. Use the table below to determine what options you'll need to supply to the configure script once you determine your needs. **Note:** PostgreSQL support for storage of object data (service, host, and command definitions, etc) is not yet supported.

Data Type	Configure Script Option	Comments
All types	--with-pgsql-xdata	This will compile in PostgreSQL support for all types of external data (downtime data, comment data, status data, retention data, and extended data). Support for object data (service and host definitions, etc.) is as of yet non-existent.
Comment data	--with-pgsql-downtime	This will compile in PostgreSQL support for downtime data (it will replace the standard downtime file)
Comment data	--with-pgsql-comments	This will compile in PostgreSQL support for comment data (it will replace the standard comment file)
Status data	--with-pgsql-status	This will compile in PostgreSQL support for status data (it will replace the standard status log)
Retention data	--with-pgsql-retention	This will compile in PostgreSQL support for retention data (it will replace the standard state_retention file)
Extended data	--with-pgsql-extinfo	This will compile in PostgreSQL support for extended data (it will replace the standard hostextinfo[] and serviceextinfo[] definitions in the CGI config file)

Configuration Directives

Once you decide what types of external data you want to store in one or more databases, you'll have to add some configuration directives to the resource file and/or the CGI config file. Here we go...

Configuration Directives For Downtime Data: (*--with-mysql-downtime* or *--with-pgsql-downtime* options):

In the CGI config file, you need to add the following directives (the downtime_file directive in the main configuration file is no longer used)...

```
xdddb_host=database_host
xdddb_port=database_port
xdddb_username=database_user
xdddb_password=database_password
xdddb_database=database_name
```

These are fairly self-explanatory. They are used by the CGIs to identify the address of your database server (and the port it is running on), the name of the database in which the downtime entries should be stored, and the username/password that should be used when connecting to the databse server. Nagios will assume that two tables (as defined here) exist in this database for storage of downtime data. **Note:** The CGIs only need read access to the downtime data, so this user should only have SELECT privileges on the comment tables.

In a resource file, you need to add the following directives...

```
xdddb_host=database_host
xdddb_port=database_port
xdddb_username=database_user
xdddb_password=database_password
xdddb_database=database_name
xdddb_optimize_data=[0/1]
```

There directives are identical to the ones you added to the CGI config file, except these are used by the Nagios process. The database user you specify here needs to have SELECT, INSERT, UPDATE, and DELETE privileges on the downtime tables. The CGIs do not attempt to read the contents of any resource files, so you can set restrictive permissions on them and make sure that no one other than the Nagios process can read them. The *xdddb_optimize_data* option will force Nagios to optimize data in the downtime tables when it starts/restarts. If you're using PostgreSQL DB support for downtime data, this means that a VACUUM is run on the downtime tables.

Configuration Directives For Comment Data: (*--with-mysql-comments* or *--with-pgsql-comments* options):

In the CGI config file, you need to add the following directives (the comment_file directive in the main configuration file is no longer used)...

```
xcddb_host=database_host
xcddb_port=database_port
xcddb_username=database_user
xcddb_password=database_password
xcddb_database=database_name
```

These are fairly self-explanatory. They are used by the CGIs to identify the address of your database server (and the port it is running on), the name of the database in which the comments should be stored, and the username/password that should be used when connecting to the databse server. Nagios will assume that two tables (as defined here) exist in this database for storage of comment data. **Note:** The CGIs only need read access to the comments, so this user should only have SELECT privileges on the comment tables.

In a resource file, you need to add the following directives...

xcddb_host=*database_host*
xcddb_port=*database_port*
xcddb_username=*database_user*
xcddb_password=*database_password*
xcddb_database=*database_name*
xcddb_optimize_data=*[0/1]*

There directives are identical to the ones you added to the CGI config file, except these are used by the Nagios process. The database user you specify here needs to have SELECT, INSERT, UPDATE, and DELETE privileges on the comment tables. The CGIs do not attempt to read the contents of any resource files, so you can set restrictive permissions on them and make sure that no one other than the Nagios process can read them. The *xcddb_optimize_data* option will force Nagios to optimize data in the comment tables when it starts/restarts. If you're using PostgreSQL DB support for comments, this means that a VACUUM is run on the comment data tables.

Configuration Directives For Status Data: (*--with-mysql-status* or *--with-pgsql-status* options):

In the CGI config file, you need to add the following directives (the status_file directive in the main configuration file is no longer used)...

xsddb_host=*database_host*
xsddb_port=*database_port*
xsddb_username=*database_user*
xsddb_password=*database_password*
xsddb_database=*database_name*

These are fairly self-explanatory. They are used by the CGIs to identify the address of your database server (and the port it is running on), the name of the database in which the status data should be stored, and the username/password that should be used when connecting to the database. Nagios will assume that three tables (as defined here) exist in this database for storage of status data. **Note:** The CGIs only need read access to the status data, so the database user you specify here should only have SELECT privileges on the status tables.

In a resource file, you need to add the following directives...

xsddb_host=*database_host*
xsddb_port=*database_port*
xsddb_username=*database_user*
xsddb_password=*database_password*
xsddb_database=*database_name*
xsddb_optimize_data=*[0/1]*
xsddb_optimize_interval=*seconds*

These directives are used by the Nagios process instead of the CGIs. The only difference between these directives and those found in the CGI config file is the fact that the database user you specify here needs to have SELECT, INSERT, UPDATE, and DELETE privileges on the status tables. The CGIs do not attempt to read the contents of any resource files, so you can set restrictive permissions on them and make sure that no one other than the Nagios process can read them. The *xsddb_optimize_data* option will force Nagios to periodically optimize data in the status tables. The frequency of optimization is determined by the number of seconds specified by the *xsddb_optimize_interval* option. If you're using PostgreSQL DB support for status data, this means that a VACUUM is run on the status data tables.

Configuration Directives For Retention Data: (*--with-mysql-retention* or *--with-pgsql-retention* options):

In a resource file, you need to add the following directives (the state_retention_file directive in the main config file is no longer used)...

xrddb_host=*database_host*
xrddb_port=*database_port*
xrddb_username=*database_user*
xrddb_password=*database_password*
xrddb_database=*database_name*
xrddb_optimize_data=*[0/1]*

These are fairly self-explanatory. They are used by the Nagios process to identify the address of your database server (and the port it is running on), the name of the database in which the retention data should be stored, and the username/password that should be used when connecting to the database. Nagios will assume that three tables (as defined here) exist in this database for storage of retention data. The user you specify here needs to have SELECT, INSERT, UPDATE, and DELETE privileges on the retention tables. The CGIs do not attempt to read the contents of any resource files, so you can set restrictive permissions on them and make sure that no one other than the Nagios process can read them. The *xrddb_optimize_data* option will force Nagios to optimize data in the retention tables when it starts/restarts. If you're using PostgreSQL DB support for retention data, this means that a VACUUM is run on the retention data tables.

Configuration Directives For Extended Data: (*--with-mysql-extinfo* or *--with-pgsql-extinfo* options):

In the CGI config file, you need to add the following directives (the hostextino[] and serviceextinfo[] directives in the CGI config file are no longer used)...

xeddb_host=*database_host*
xeddb_port=*database_port*
xeddb_username=*database_user*
xeddb_password=*database_password*
xeddb_database=*database_name*

These are fairly self-explanatory. They are used by the CGIs to identify the address of your database server (and the port it is running on), the name of the database in which the extended data is stored, and the username/password that should be used when connecting to the database. Nagios will assume that two tables (as defined here) exists in this database for storage of extended data. The user you specify here should only have SELECT privileges on the extended info tables.

Table Definitions

In order to read from or write to a database, you first have to create it and setup some tables to hold your data. Note: If you are storing more than one type of external data in databases, you could create separate databases for each type of data (comments, status info, etc.) You could also keep everything in a single database (different data is kept in different tables). In your database(s) you're going to have to setup the appropriate table(s) so Nagios can actually read/write data.

Important: Scripts for creating tables for all types of external data for both MySQL and PostgreSQL databases can be found in the *contrib/database/* directory of the distribution.

Downtime Data Tables:

There are two tables (named **hostdowntime** and **servicedowntime**) you need to create in order to store downtime entries in a database. One of the tables is used to store host downtime and the other for service downtime. The CGIs only need SELECT rights on these tables, while the main Nagios process needs SELECT, INSERT, UPDATE, and DELETE privileges.

Comment Data Tables:

There are two tables (named **hostcomments** and **servicecomments**) you need to create in order to store comments in a database. One of the tables is used to store host comments and the other for service comments. The CGIs only need SELECT rights on these tables, while the main Nagios process needs SELECT, INSERT, UPDATE, and DELETE privileges.

Status Data Tables:

There are three tables (named **programstatus**, **hoststatus**, and **servicestatus**) you need to create in order to store status data in a database. One of the tables is used to store program status data, one for host status data, and another for service status data. The CGIs only need SELECT rights on these tables, while the main process needs SELECT, INSERT, UPDATE, and DELETE privileges.

Retention Data Tables:

There are three tables (named **programretention**, **hostretention**, and **serviceretention**) you need to create in order to store retention data in a database. One is used to store program data, one for host data, and another for service data. The main process needs SELECT, INSERT, UPDATE, and DELETE privileges on these tables. The CGIs *do not* access these tables at all.

Extended Data Tables:

There are two tables (named **hostextinfo** and **serviceextinfo**) you need to create in order to store extended data in a database. One table is used to store extended host information and the other for extended service information (used by the CGIs). The CGIs need SELECT privileges on these tables. The main Nagios process *does not* access these tables at all.

Using The Embedded Perl Interpreter

Introduction

Stephen Davies has contributed code that allows you to compile Nagios with an embedded Perl interpreter. This may be of interest to you if you rely heavily on plugins written in Perl.

Stanley Hopcroft has worked with the embedded Perl interpreter quite a bit and has commented on the advantages/disadvanges of using it. He has also given several helpful hints on creating Perl plugins that work properly with the embedded interpreter. The majority of this documentation comes from his comments.

It should be noted that "ePN", as used in this documentation, refers to embedded Perl Nagios, or if you prefer, Nagios compiled with an embedded Perl interpreter.

Advantages

Some advantages of ePN (embedded Perl Nagios) include:

- Nagios will spend much less time running your Perl plugins because it no longer forks to execute the plugin (each time loading the Perl interpreter). Instead, it executes your plugin by making a library call.
- It greatly reduces the system impact of Perl plugins and/or allows you to run more checks with Perl plugin than you otherwise would be able to. In other words, you have less incentive to write plugins in other languages such as C/C++, or Expect/TCL, that are generally recognised to have development times at least an order of magnitude slower than Perl (although they do run about ten times faster also - TCL being an exception).
- If you are not a C programmer, then you can still get a huge amount of mileage out of Nagios by letting Perl do all the heavy lifting without having Nagios slow right down. Note however, that the ePN will not speed up your plugin (apart from eliminating the interpreter load time). If you want fast plugins then consider Perl XSUBs (XS), or C *after* you are sure that your Perl is tuned and that you have a suitable algorithm (Benchmark.pm is *invaluable* for comparing the performance of Perl language elements).
- Using the ePN is an excellentt opportunity to learn more about Perl.

Disadvantages

The disadvantages of ePN (embedded Perl Nagios) are much the same as Apache mod_perl (i.e. Apache with an embedded interpreter) compared to a plain Apache:

- A Perl program that works *fine* with plain Nagios may *not* work with the ePN. You may have to modify your plugins to get them to work.
- Perl plugins are harder to debug under an ePN than under a plain Nagios.
- Your ePN will have a larger SIZE (memory footprint) than a plain Nagios.
- Some Perl constructs cannot be used or may behave differently than what you would expect.

- You may have to be aware of 'more than one way to do it' and choose a way that seems less attractive or obvious.
- You will need greater Perl knowledge (but nothing very esoteric or stuff about Perl internals - unless your plugin uses XSUBS).

Target Audience

- Average Perl developers; those with an appreciation of the languages powerful features without knowledge of internals or an in depth knowledge of those features.
- Those with a utilitarian appreciation rather than a great depth of understanding.
- If you are happy with Perl objects, name management, data structures, and the debugger, that's probably sufficient.

Things you should do when developing a Perl Plugin (ePN or not)

- Always always generate some output
- Use 'use utils' and import the stuff it exports ($TIMEOUT %ERRORS &print_revision &support)
- Have a look at how the standard Perl plugins do their stuff e.g.
  - Always exit with $ERRORS{CRITICAL}, $ERRORS{OK}, etc.
  - Use getopt to read command line arguments
  - Manage timeouts
  - Call print_usage (supplied by you) when there are no command line arguments
  - Use standard switch names (eg H 'host', V 'version')

Things you must do to develop a Perl plugin for ePN

1. <DATA> can not be used; use here documents instead e.g.

```
my $data = <<DATA;
portmapper 100000
portmap 100000
sunrpc 100000
rpcbind 100000
rstatd 100001
rstat 100001
rup 100001
..
DATA

%prognum = map { my($a, $b) = split; ($a, $b) } split(/\n/, $data) ;
```

2. BEGIN blocks will not work as you expect. May be best to avoid.
3. Ensure that it is squeaky clean at compile time i.e.

- use strict
- use perl -w (other switches [T notably] may not help)
- use perl -c

4. Avoid lexical variables (my) with global scope as a means of passing __variable__ data into subroutines. In fact this is __fatal__ if the subroutine is called by the plugin more than once when the check is run. Such subroutines act as 'closures' that lock the global lexicals first value into subsequent calls of the subroutine. If however, your global is read-only (a complicated structure for example) this is not a problem. What Bekman recommends you do instead, is any of the following:

   - make the subroutine anonymous and call it via a code ref e.g.

```
turn this                        into

my $x = 1 ;                      my $x = 1 ;
sub a { .. Process $x ... }      $a_cr = sub { ... Process $x ... } ;
.                                .
.                                .
a ;                              &$a_cr ;
$x = 2                           $x = 2 ;
a ;                              &$a_cr ;

# anon closures __always__ rebind the current lexical value
```

   - put the global lexical and the subroutine using it in their own package (as an object or a module)
   - pass info to subs as references or aliases (\$lex_var or $_[n])
   - replace lexicals with package globals and exclude them from 'use strict' objections with 'use vars qw(global1 global2 ..)'

5. Be aware of where you can get more information.

   Useful information can be had from the usual suspects (the O'Reilly books, plus Damien Conways "Object Oriented Perl") but for the really useful stuff in the right context start at Stas Bekman's mod_perl guide at http://perl.apache.org/guide/.

   This wonderful book sized document has nothing whatsoever about Nagios, but all about writing Perl programs for the embedded Perl interpreter in Apache (ie Doug MacEacherns mod_perl).

   The perlembed manpage is essential for context and encouragement.

   On the basis that Lincoln Stein and Doug MacEachern know a thing or two about Perl and embedding Perl, their book 'Writing Apache Modules with Perl and C' is almost certainly worth looking at.

6. Be aware that your plugin may return strange values with an ePN and that this is likely to be caused by the problem in item #4 above

7. Be prepared to debug via:

   - having a test ePN and

- adding print statements to your plugin to display variable values to STDERR (can't use STDOUT)
- adding print statements to p1.pl to display what ePN thinks your plugin is before it tries to run it (vi)
- running the ePN in foreground mode (probably in conjunction with the former recommendations)
- use the 'Deparse' module on your plugin to see how the parser has optimised it and what the interpreter will actually get. (see 'Constants in Perl' by Sean M. Burke, The Perl Journal, Fall 2001)

  perl -MO::Deparse <your_program>

8. Be aware of what ePN is transforming your plugin too, and if all else fails try and debug the transformed version.

As you can see below p1.pl rewrites your plugin as a subroutine called 'hndlr' in the package named 'Embed::<something_related_to_your_plugin_file_name>'.

Your plugin may be expecting command line arguments in @ARGV so pl.pl also assigns @_ to @ARGV.

This in turn gets 'eval' ed and if the eval raises an error (any parse error and run error), the plugin gets chucked out.

The following output shows how a test ePN transformed the *check_rpc* plugin before attempting to execute it. Most of the code from the actual plugin is not shown, as we are interested in only the transformations that the ePN has made to the plugin). For clarity, transformations are shown in red:

```
                package main;
                use subs 'CORE::GLOBAL::exit';
                sub CORE::GLOBAL::exit { die "ExitTrap: $_[0]
(Embed::check_5frpc)"; }
                package Embed::check_5frpc; sub hndlr { shift(@_);
@ARGV=@_;
#! /usr/bin/perl -w
#
# check_rpc plugin for netsaint
#
# usage:
#    check_rpc host service
#
# Check if an rpc serice is registered and running
# using rpcinfo - $proto $host $prognum 2>&1 |";
#
# Use these hosts.cfg entries as examples
#
# command[check_nfs]=/some/path/libexec/check_rpc $HOSTADDRESS$ nfs
# service[check_nfs]=NFS;24x7;3;5;5;unix-admin;60;24x7;1;1;1;;check_rpc
#
# initial version: 3 May 2000 by Truongchinh Nguyen and Karl DeBisschop
# current status: $Revision: 1.12 $
#
# Copyright Notice: GPL
#
... rest of plugin code goes here (it was removed for brevity) ...
```

```
        }
```

9. Don't use 'use diagnostics' in a plugin run by your production ePN. I think it causes__all__ the Perl plugins to return CRITICAL.

10. Consider using a mini embedded Perl C program to check your plugin. This is not sufficient to guarantee your plugin will perform Ok with an ePN but if the plugin fails this test it will ceratinly fail with your ePN. [A sample mini ePN is included in the *contrib/* directory of the Nagios distribution for use in testing Perl plugins. Change to the contrib/ directory and type 'make mini_epn' to compile it. It must be executed from the same directory that the p1.pl file resides in (this file is distributed with Nagios).]

Compiling Nagios With The Embedded Perl Interpreter

Okay, you can breathe again now. So do you *still* want to compile Nagios with the embedded Perl interpreter? ;-)

If you want to compile Nagios with the embedded Perl interpreter you need to rerun the configure script with the addition of the *--enable-embedded-perl* option. If you want the embedded interpreter to cache internally compiled scripts, add the *--with-perlcache* option as well. Example:

```
./configure --enable-embedded-perl --with-perlcache ...other options...
```

Once you've rerun the configure script with the new options, make sure to recompile Nagios. You can check to make sure that Nagios has been compile with the embedded Perl interpreter by executing it with the *-m* command-line argument. Output from executing the command will look something like this (notice that the embedded perl interpreter is listed in the options section):

```
[nagios@firestore ]# ./nagios -m

Nagios 1.0a0
Copyright (c) 1999-2001 Ethan Galstad (nagios@nagios.org)
Last Modified: 07-03-2001
License: GPL

External Data I/O
-----------------
Object Data:      DEFAULT
Status Data:      DEFAULT
Retention Data:   DEFAULT
Comment Data:     DEFAULT
Downtime Data:    DEFAULT
Performance Data: DEFAULT


Options
-------
* Embedded Perl compiler (With caching)
```

Object Inheritence Using Template-Based Config Data

Introduction

One of my primary motivations for adding support for template-based configuration data was its ability to easily allow object definitions to inherit various properties from other object definitions. Object property inheritence is accomplished through recursion when Nagios processes your configuration files.

This documentation attempts to explain recursion and inheritence in object definitions that is available when you compile Nagios with support for template-based object configuration data. It should be noted that the recursion and inheritence principles described here also apply to template-based extended information data.

If you are still confused about how recursion and inheritence work after reading this, take a look at the sample template-base config files provided in the distribution. If that still doesn't help, drop an email message with a *detailed* description of your problem to the *nagios-users* mailing list.

Basics

There are three variables affecting recursion and inheritence that are present in all object definitions. They are as follows...

```
define someobjecttype{

        object-specific variables ...

        name                template_name
        use                 name_of_template_to_use
        register            [0/1]
        }
```

The first variable is *name*. Its just a "template" name that can be referenced in other object definitions so they can inherit the objects properties/variables. Template names must be unique amongst objects of the same type, so you can't have two or more host definitions that have "hosttemplate" as their template name.

The second variable is *use*. This is where you specify the name of the template object that you want to inherit properties/variables from. The name you specify for this variable must be defined as another object's template named (using the *name* variable).

The third variable is *register*. This variable is used to indicate whether or not the object definition should be "registered" with Nagios. By default, all object definitions are registered. If you are using a partial object definition as a template, you would want to prevent it from being registered (an example of this is provided later). Values are as follows: 0 = do NOT register object definition, 1 = register object definition (this is the default).

Local Variables vs. Inherited Variables

One important thing to understand with inheritance is that "local" object variables always take precedence over variables defined in the template object. Take a look at the following example of two host definitions (not all required variables have been supplied):

```
define host{
        host_name               bighost1
        check_command           check-host-alive
        notification_options    d,u,r
        max_check_attempts      5
        name                    hosttemplate1
        }

define host{
        host_name               bighost2
        max_check_attempts      3
        use                     hosttemplate1
        }
```

You'll note that the definition for host *bighost1* has been defined as having *hosttemplate1* as its template name. The definition for host *bighost2* is using the definition of *bighost1* as its template object. Once Nagios processes this data, the resulting definition of host *bighost2* would be equivalent to this definition:

```
define host{
        host_name               bighost2
        check_command           check-host-alive
        notification_options    d,u,r
        max_check_attempts      3
        }
```

You can see that the *check_command* and *notification_options* variables were inherited from the template object (where host *bighost1* was defined). However, the *host_name* and *max_check_attempts* variables were not inherited from the template object because they were defined locally. Remember, locally defined variables override variables that would normally be inherited from a template object. That should be a fairly easy concept to understand.

Inheritence Chaining

Objects can inherit properties/variables from multiple levels of template objects. Take the following example:

```
define host{
        host_name               bighost1
        check_command           check-host-alive
        notification_options    d,u,r
        max_check_attempts      5
        name                    hosttemplate1
        }
define host{
        host_name                bighost2
        max_check_attempts       3
        use                      hosttemplate1
        name                     hosttemplate2
        }

define host{
        host_name                bighost3
        use                      hosttemplate2
        }
```

You'll notice that the definition of host *bighost3* inherits variables from the definition of host *bighost2*, which in turn inherits variables from the definition of host *bighost1*. Once Nagios processes this configuration data, the resulting host definitions are equivalent to the following:

```
        define host{
                host_name                bighost1
                check_command            check-host-alive
                notification_options     d,u,r
                max_check_attempts       5
                }

        define host{
                host_name                bighost2
                check_command            check-host-alive
                notification_options     d,u,r
                max_check_attempts       3
                }

        define host{
                host_name                bighost3
                check_command            check-host-alive
                notification_options     d,u,r
                max_check_attempts       3
                }
```

There is no inherent limit on how "deep" inheritance can go, but you'll probably want to limit yourself to at most a few levels in order to maintain sanity.

Using Incomplete Object Definitions as Templates

It is possible to use imcomplete object definitions as templates for use by other object definitions. By "incomplete" definition, I mean that all required variables in the object have not been supplied in the object definition. It may sound odd to use incomplete definitions as templates, but it is in fact recommended that you use them. Why? Well, they can serve as a set of defaults for use in all other object definitions. Take the following example:

```
        define host{
                check_command           check-host-alive
                notification_options    d,u,r
                max_check_attempts      5
                name                    generichosttemplate
                register                        0
                }

        define host{
                host_name               bighost1
                address                 192.168.1.3
                use                     generichosthosttemplate
                }

        define host{
                host_name               bighost2
                address                 192.168.1.4
                use                     generichosthosttemplate
                }
```

Notice that the first host definition is incomplete because it is missing the required *host_name* variable. We don't need to supply a host name because we just want to use this definition as a generic host template. In order to prevent this definition from being registered with Nagios as a normal host, we set the *register* variable to 0.

The definitions of hosts *bighost1* and *bighost2* inherit their values from the generic host definition. The only variable we've chosed to override is the *address* variable. This means that both hosts will have the exact same properties, except for their *host_name* and *address* variables. Once Nagios processes the config data in the example, the resulting host definitions would be equivalent to specifying the following:

```
        define host{
                host_name               bighost1
                address                 192.168.1.3
                check_command           check-host-alive
                notification_options    d,u,r
                max_check_attempts      5
                }

        define host{
                host_name               bighost2
                address                 192.168.1.4
                check_command           check-host-alive
                notification_options    d,u,r
                max_check_attempts      5
                }
```

At the very least, using a template definition for default variables will save you a lot of typing. It'll also save you a lot of headaches later if you want to change the default values of variables for a large number of hosts.

Time-Saving Tricks For Template-Based Object Definitions

or...
"How To Preserve Your Sanity"

Introduction

This documentation attempts to explain how you can exploit the (somewhat) hidden features template-based object definitions to save your sanity. How so, you ask? Several types of objects allow you to specify multiple host names and/or hostgroup names in definitions, allowing you to "copy" the object defintion to multiple hosts or services. I'll cover each type of object that supports these features seperately. For starters, the object types which support this time-saving feature are as follows:

- Services
- Service escalations
- Service dependencies
- Host escalations
- Host dependencies
- Hostgroups
- Hostgroup escalations

Object types that are not listed above (i.e. timeperiods, commands, etc.) do not support the features I'm about to describe.

Services

Multiple Hosts: If you want to create identical services that are assigned to multiple hosts, you can specify multiple hosts in the *host_name* directive as follows:

```
define service{
        host_name               HOST1,HOST2,HOST3,...,HOSTN
        service_description     SOMESERVICE
        other service directives ...
        }
```

The definition above would create a service called *SOMESERVICE* on hosts *HOST1* through *HOSTN*. All the instances of the *SOMESERVICE* service would be identical (i.e. have the same check command, max check attempts, notification period, etc.).

All Hosts In Multiple Hostgroups: If you want to create identical services that are assigned to all hosts in one or more hostgroups, you can do so by creating a single service definition. How? The *hostgroup_name* directive allows you to specify the name of one or more hostgroups that the service should be created for:

```
define service{
        hostgroup_name          HOSTGROUP1,HOSTGROUP2,...,HOSTGROUPN
        service_description     SOMESERVICE
        other service directives ...
        }
```

The definition above would create a service called *SOMESERVICE* on all hosts that are members of hostgroups *HOSTGROUP1* through *HOSTGROUPN*. All the instances of the *SOMESERVICE* service would be identical (i.e. have the same check command, max check attempts, notification period, etc.).

All Hosts: If you want to create identical services that are assigned to all hosts that are defined in your configuration files, you can use a wildcard in the *host_name* directive as follows:

```
define service{
        host_name               *
        service_description     SOMESERVICE
        other service directives ...
        }
```

The definition above would create a service called *SOMESERVICE* on **all hosts** that are defined in your configuration files. All the instances of the *SOMESERVICE* service would be identical (i.e. have the same check command, max check attempts, notification period, etc.).

Service Escalations

Multiple Hosts: If you want to create service escalations for services of the same name/description that are assigned to multiple hosts, you can specify multiple hosts in the *host_name* directive as follows:

```
define serviceescalation{
        host_name               HOST1,HOST2,HOST3,...,HOSTN
        service_description     SOMESERVICE
        other escalation directives ...
        }
```

The definition above would create a service escalation for services called *SOMESERVICE* on hosts *HOST1* through *HOSTN*. All the instances of the service escalation would be identical (i.e. have the same contact groups, notification interval, etc.).

All Hosts In Multiple Hostgroups: If you want to create service escalations for services of the same name/description that are assigned to all hosts in in one or more hostgroups, you can do use the *hostgroup_name* directive as follows:

```
define serviceescalation{
        hostgroup_name          HOSTGROUP1,HOSTGROUP2,...,HOSTGROUPN
        service_description     SOMESERVICE
        other escalation directives ...
        }
```

The definition above would create a service escalation for services called *SOMESERVICE* on all hosts that are members of hostgroups *HOSTGROUP1* through *HOSTGROUPN*. All the instances of the service escalation would be identical (i.e. have the same contact groups, notification interval, etc.).

All Hosts: If you want to create identical service escalations for services of the same name/description that are assigned to all hosts that are defined in your configuration files, you can use a wildcard in the *host_name* directive as follows:

```
        define serviceescalation{
                host_name                *
                service_description      SOMESERVICE
                other escalation directives ...
                }
```

The definition above would create a service escalation for all services called *SOMESERVICE* on **all hosts** that are defined in your configuration files. All the instances of the service escalation would be identical (i.e. have the same contact groups, notification interval, etc.).

Multiple Services: If you want to create service escalations for all services assigned to a particular host, you can use a wildcard in the *service_description* directive as follows:

```
        define serviceescalation{
                host_name                HOST1
                service_description      *
                other escalation directives ...
                }
```

The definition above would create a service escalation for **all** services on host *HOST1*. All the instances of the service escalation would be identical (i.e. have the same contact groups, notification interval, etc.).

If you feel like being particularly adventurous, you can specify a wildcard in both the *host_name* and *service_description* directives. Doing so would create a service escalation for **all services** that you've defined in your configuration files.

Service Dependencies

Multiple Hosts: If you want to create service dependencies for services of the same name/description that are assigned to multiple hosts, you can specify multiple hosts in the *host_name* and or *dependent_host_name* directives as follows:

```
        define servicedependency{
                host_name                        HOST1,HOST2
                service_description              SERVICE1
                dependent_host_name              HOST3,HOST4
                dependent_service_description    SERVICE2
                other dependency directives ...
                }
```

In the example above, service *SERVICE2* on hosts *HOST3* and *HOST4* would be dependent on service *SERVICE1* on hosts *HOST1* and *HOST2*. All the instances of the service dependencies would be identical except for the host names (i.e. have the same notification failure criteria, etc.).

All Hosts In Multiple Hostgroups: If you want to create service dependencies for services of the same name/description that are assigned to all hosts in in one or more hostgroups, you can do use the *hostgroup_name* and/or *dependent_hostgroup_name* directives as follows:

```
        define servicedependency{
                hostgroup_name                  HOSTGROUP1,HOSTGROUP2
                service_description             SERVICE1
                dependent_hostgroup_name        HOSTGROUP3,HOSTGROUP4
                dependent_service_description   SERVICE2
                other dependency directives ...
                }
```

In the example above, service *SERVICE2* on all hosts in hostgroups *HOSTGROUP3* and *HOSTGROUP4* would be dependent on service *SERVICE1* on all hosts in hostgroups *HOSTGROUP1* and *HOSTGROUP2*. Assuming there were five hosts in each of the hostgroups, this definition would be equivalent to creating 100 single service dependency definitions! All the instances of the service dependency would be identical except for the host names (i.e. have the same notification failure criteria, etc.).

Multiple Services: If you want to create service dependencies for all services assigned to a particular host, you can use a wildcard in the *service_description* and/or *dependent_service_description* directives as follows:

```
        define servicedependency{
                host_name                       HOST1
                service_description             *
                dependent_host_name             HOST2
                dependent_service_description   *
                other dependency directives ...
                }
```

In the example above, **all services** on host *HOST2* would be dependent on **all services** on host *HOST1*. All the instances of the service dependencies would be identical (i.e. have the same notification failure criteria, etc.).

Host Escalations

Multiple Hosts: If you want to create host escalations for multiple hosts, you can specify multiple hosts in the *host_name* directive as follows:

```
        define hostescalation{
                host_name               HOST1,HOST2,HOST3,...,HOSTN
```

```
                    other escalation directives ...
                    }
```

The definition above would create a host escalation for hosts *HOST1* through *HOSTN*. All the instances of the host escalation would be identical (i.e. have the same contact groups, notification interval, etc.).

All Hosts In Multiple Hostgroups: If you want to create host escalations for all hosts in in one or more hostgroups, you can do use the *hostgroup_name* directive as follows:

```
          define hostescalation{
                    hostgroup_name          HOSTGROUP1,HOSTGROUP2,...,HOSTGROUPN
                    other escalation directives ...
                    }
```

The definition above would create a host escalation on all hosts that are members of hostgroups *HOSTGROUP1* through *HOSTGROUPN*. All the instances of the host escalation would be identical (i.e. have the same contact groups, notification interval, etc.).

All Hosts: If you want to create identical host escalations for all hosts that are defined in your configuration files, you can use a wildcard in the *host_name* directive as follows:

```
          define hostescalation{
                    host_name               *
                    other escalation directives ...
                    }
```

The definition above would create a hosts escalation for **all hosts** that are defined in your configuration files. All the instances of the host escalation would be identical (i.e. have the same contact groups, notification interval, etc.).

Host Dependencies

Multiple Hosts: If you want to create host dependencies for multiple hosts, you can specify multiple hosts in the *host_name* and/or *dependent_host_name* directives as follows:

```
          define hostdependency{
                    host_name               HOST1,HOST2
                    dependent_host_name     HOST3,HOST4,HOST5
                    other dependency directives ...
                    }
```

The definition above would be equivalent to creating six seperate host dependencies. In the example above, hosts *HOST3*, *HOST4* and *HOST5* would be dependent upon both *HOST1* and *HOST2*. All the instances of the host dependencies would be identical except for the host names (i.e. have the same notification failure criteria, etc.).

All Hosts In Multiple Hostgroups: If you want to create host escalations for all hosts in in one or more hostgroups, you can do use the *hostgroup_name* and /or *dependent_hostgroup_name* directives as follows:

```
        define hostdependency{
                hostgroup_name                  HOSTGROUP1,HOSTGROUP2
                dependent_hostgroup_name        HOSTGROUP3,HOSTGROUP4
                other dependency directives ...
                }
```

In the example above, all hosts in hostgroups *HOSTGROUP3* and *HOSTGROUP4* would be dependent on all hosts in hostgroups *HOSTGROUP1* and *HOSTGROUP2*. All the instances of the host dependencies would be identical except for host names (i.e. have the same notification failure criteria, etc.).

Hostgroups

All Hosts: If you want to create a hostgroup that has all hosts that are defined in your configuration files as members, you can use a wildcard in the *members* directive as follows:

```
        define hostgroup{
                hostgroup_name          HOSTGROUP1
                members                 *
                other hostgroup directives ...
                }
```

The definition above would create a hostgroup called *HOSTGROUP1* that has all **all hosts** that are defined in your configuration files as members.

Hostgroup Escalations

Multiple Hostgroups: If you want to create identical hostgroup escalations that are assigned to multiple hostgroups, you can specify multiple hostgroups in the *hostgroup_name* directive as follows:

```
        define hostgroupescalation{
                hostgroup_name          HOSTGROUP1,HOSTGROUP2,...,HOSTGROUPN
                other escalation directives ...
                }
```

The definition above would create a seperate hostgroup escalation for hostgroups *HOSTGROUP1* through *HOSTGROUPN*. All the instances of the hostgroup escalation would be identical (i.e. contact groups, notification interval, etc.).

Portsentry Integration

Introduction

This example explains how to easily generate alerts in Nagios for port scan that are detected by Psionic Software's Portsentry software. These directions assume that the host which you are generating alerts for (i.e. the host you are running Portsentry on) is not the same host on which Nagios is running. If you want to generate alerts on the same host that Nagios is running you will need to make a few modifications to the examples I provide. Also, I am assuming that you having installed the nsca daemon on your monitoring server and the nsca client (*send_nsca*) on the machine that you are running Portsentry on.

Defining The Service

First off you're going to have to define a service in your object configuration file for the port scan alerts. Assuming that the host that the alerts are originating from is called **firestorm**, a sample service definition might look something like this:

```
define service{
        host_name                       firestorm
        service_description             Port Scans
        is_volatile                     1
        active_checks_enabled           0
        passive_checks_enabled          1
        max_check_attempts              1
        contact_groups                  security-admins
        notification_interval           120
        notification_period             24x7
        notification_options            w,u,c,r
        check_command                   check_none
        }
```

Important things to note are the fact that this service has the *volatile* option enabled. We want this option enabled because we want a notification to be generated for every alert that comes in. Also of note is the fact that active checks are disabled for the service. The command used in the *check_command* option is not really used at all - its just there to keep Nagios from complaining. Passive checks are enabled however, as all port scan alert information will have to be sent in passively by the *nsca client* from the **firestorm** host.

Configuring Portsentry

In order to get Portsentry to send an alert to your monitoring box when it detects a port scan, you'll need to define a command for the *KILL_RUN_CMD* option in the Portsentry config file (*portsentry.conf*). It should look something like the following:

```
KILL_RUN_CMD="/usr/local/nagios/libexec/eventhandlers/handle_port_scan $TARGET$ $PORT$"
```

This line assumes that there is a script called *handle_port_scan* in the */usr/local/nagios/libexec/eventhandlers/* directory on **firestorm**. The directory and script name can be changed to whatever you want.

Writing The Script

The last thing you need to do is write the *handle_port_scan* script on **firestorm** that will send the alert back to the monitoring host. It might look something like this:

```
#!/bin/sh

# Arguments:
#       $1 = target
#       $2 = port
# Submit port scan to Nagios
/usr/local/nagios/libexec/eventhandlers/submit_check_result firestorm "Port Scans" 2 "Port scan from
$1 on port $2.  Host has been firewalled."
```

Notice that the *handle_port_scan* script calls the *submit_check_result* to actually send the alert back to the monitoring host. Assuming your monitoring host is called **monitor**, the *submit check_result* script might look like this (you'll have to modify this to specify the proper location of the *send_nsca* program on **firestorm**):

```
#!/bin/sh

# Arguments
#	$1 = name of host in service definition
#	$2 = name/description of service in service definition
#	$3 = return code
#	$4 = output

/bin/echo -e "$1\t$2\t$3\t$4\n" | /usr/local/nagios/bin/send_nsca monitor -c
/usr/local/nagios/etc/send_nsca.cfg
```

Finishing Up

You've now configured everything you need to, so all you have to do is restart the *portsentry* process on **firestorm** and restart Nagios on your monitoring server. That's it! When the Portsentry software on **firestorm** detects a port scan, you should be getting alerts in Nagios. The plugin output for the alert will look something like the following:

```
Port scan from 24.24.137.131 on port 21. Host has been firewalled.
```

UCD-SNMP (NET-SNMP) Integration

Note: Nagios is not designed to be a replacement for a full-blown SNMP management application like HP OpenView or OpenNMS. However, you can set things up so that SNMP traps received by a host on your network can generate alerts in Nagios. Here's how...

Introduction

This example explains how to easily generate alerts in Nagios for SNMP traps that are received by the UCD-SNMP *snmptrapd* daemon. These directions assume that the host which is receiving SNMP traps is not the same host on which Nagios is running. If your monitoring box is the same box that is receiving SNMP traps you will need to make a few modifications to the examples I provide. Also, I am assuming that you having installed the nsca daemon on your monitoring server and the nsca client (*send_nsca*) on the machine that is receiving SNMP traps.

For the purposes of this example, I will be describing how I setup Nagios to generate alerts from SNMP traps received by the ArcServe backup jobs running on my Novell servers. I wanted to get notified when backups failed, so this worked very nicely for me. You'll have to tweak the examples in order to make it suit your needs.

Defining The Service

First off you're going to have to define a service in your object configuration file for the SNMP traps (in this example, I am defining a service for ArcServe backup jobs). Assuming that the host that the alerts are originating from is called **novellserver**, a sample service definition might look something like this:

```
define service{
        host_name                       novellserver
        service_description             ArcServe Backup
        is_volatile                     1
        active_checks_enabled           0
        passive_checks_enabled          1
        max_check_attempts              1
        contact_groups                  novell-backup-admins
        notification_interval           120
        notification_period             24x7
        notification_options            w,u,c,r
        check_command                   check_none
        }
```

Important things to note are the fact that this service has the *volatile* option enabled. We want this option enabled because we want a notification to be generated for every alert that comes in. Also of note is the fact that active checks are disabled for the service, while passive checks are enabled. This means that the service will never be actively checked - all alert information will have to be sent in passively by the *nsca client* on the SNMP management host (in my example, it will be called **firestorm**).

ArcServe and Novell SNMP Configuration

In order to get ArcServe (and my Novell server) to send SNMP traps to my management host, I had to do the following:

1. Modify the ArcServe autopilot job to send SNMP traps on job failures, successes, etc.
2. Edit SYS:\ETC\TRAPTARG.CFG and add the IP address of my management host (the one receiving the SNMP traps)
3. Load SNMP.NLM
4. Load ALERT.NLM to facilitate the actual sending of the SNMP traps

SNMP Management Host Configuration

On my Linux SNMP management host (**firestorm**), I installed the UCD-SNMP (NET-SNMP) software. Once the software was installed I had to do the following:

1. Install the ArcServe MIBs (included on the ArcServe installation CD)
2. Edit the snmptrapd configuration file *(/etc/snmp/snmptrapd.conf)* to define a trap handler for ArcServe alerts. This is detailed below.
3. Start the *snmptrapd* daemon to listen for incoming SNMP traps

In order to have the *snmptrapd* daemon route ArcServe SNMP traps to our Nagios host, we've got to define a traphandler in the */etc/snmp/snmptrapd.conf* file. In my setup, the config file looked something like this:

```
#############################
# ArcServe SNMP Traps
#############################

# Tape format failures
traphandle ARCserve-Alarm-MIB::arcServetrap9 /usr/local/nagios/libexec/eventhandlers/handle-arcserve-
trap 9

# Failure to read tape header
traphandle ARCserve-Alarm-MIB::arcServetrap10 /usr/local/nagios/libexec/eventhandlers/handle-
arcserve-trap 10

# Failure to position tape
traphandle ARCserve-Alarm-MIB::arcServetrap11 /usr/local/nagios/libexec/eventhandlers/handle-
arcserve-trap 11

# Cancelled jobs
traphandle ARCserve-Alarm-MIB::arcServetrap12 /usr/local/nagios/libexec/eventhandlers/handle-
arcserve-trap 12

# Successful jobs
traphandle ARCserve-Alarm-MIB::arcServetrap13 /usr/local/nagios/libexec/eventhandlers/handle-
arcserve-trap 13

# Imcomplete jobs
traphandle ARCserve-Alarm-MIB::arcServetrap14 /usr/local/nagios/libexec/eventhandlers/handle-
arcserve-trap 14

# Job failures
traphandle ARCserve-Alarm-MIB::arcServetrap15 /usr/local/nagios/libexec/eventhandlers/handle-
arcserve-trap 15
```

This example assumes that you have a */usr/local/nagios/libexec/eventhandlers/* directory on your SNMP mangement host and that the *handle-arcserve-trap* script exists there. You can modify these to fit your setup. Anyway, the *handle-arcserve-trap* script on my management host looked something like this:

```
#!/bin/sh

# Arguments:
#  $1 = trap type

        # First line passed from snmptrapd is FQDN of host that sent the trap
     read host

    # Given a FQDN, get the short name of the host as it is setup in Nagios
    hostname="unknown"
    case $host in
        novellserver.mylocaldomain.com)
            hostname="novellserver"
            ;;
        nt.mylocaldomain.com)
            hostname="ntserver"
            ;;
        esac
    # Get severity level (OK, WARNING, UNKNOWN, or CRITICAL) and plugin output based on trape type
    state=-1
    output="No output"
    case "$1" in

        # failed to format tape - critical
        11)
            output="Critical: Failed to format tape"
            state=2
            ;;

        # failed to read tape header - critical
        10)
            output="Critical: Failed to read tape header"
            state=2
            ;;

        # failed to position tape - critical
        11)
            output="Critical: Failed to position tape"
            state=2
            ;;

        # backup cancelled - warning
        12)
            output="Warning: ArcServe backup operation cancelled"
            state=1
            ;;

        # backup success - ok
        13)
            output="Ok: ArcServe backup operation successful"
            state=0
            ;;

        # backup incomplete - warning
        14)
            output="Warning: ArcServe backup operation incomplete"
            state=1
            ;;

        # backup failure - critical
        15)
            output="Critical: ArcServe backup operation failed"
            state=2
            ;;
    esac


    # Submit passive check result to monitoring host
    /usr/local/nagios/libexec/eventhandlers/submit_check_result $hostname "ArcServe Backup" $state
"$output"

exit 0
```

Notice that the *handle-arcserve-trap* script calls the *submit_check_result* script to actually send the alert back to the monitoring host. Assuming your monitoring host is called **monitor**, the *submit check_result* script might look like this (you'll have to modify this to specify the proper location of the *send_nsca* program on your management host):

```
#!/bin/sh

# Arguments
#       $1 = name of host in service definition
#       $2 = name/description of service in service definition
#       $3 = return code
#       $4 = output

/bin/echo -e "$1\t$2\t$3\t$4\n" | /usr/local/nagios/bin/send_nsca monitor -c
/usr/local/nagios/etc/send_nsca.cfg
```

Finishing Up

You've now configured everything you need to, so all you have to do is restart the Nagios on your monitoring server. That's it! You should be getting alerts in Nagios whenever ArcServe jobs fail, succeed, etc.

TCP Wrapper Integration

Introduction

This example explains how to easily generate alerts in Nagios for connection attempts that are rejected by TCP wrappers. These directions assume that the host which you are generating alerts for (i.e. the host you are using TCP wrappers on) is not the same host on which Nagios is running. If you want to generate alerts on the same host that Nagios is running you will need to make a few modifications to the examples I provide. Also, I am assuming that you having installed the nsca daemon on your monitoring server and the nsca client (*send_nsca*) on the machine that you are generating TCP wrapper alerts from.

Defining The Service

First off you're going to have to define a service in your object configuration file for the TCP wrapper alerts. Assuming that the host that the alerts are originating from is called **firestorm**, a sample service definition might look something like this:

```
define service{
        host_name                       firestorm
        service_description             TCP Wrappers
        is_volatile                     1
        active_checks_enabled           0
        passive_checks_enabled          1
        max_check_attempts              1
        contact_groups                  security-admins
        notification_interval           120
        notification_period             24x7
        notification_options            w,u,c,r
        check_command                   check_none
        }
```

Important things to note are the fact that this service has the *volatile* option enabled. We want this option enabled because we want a notification to be generated for every alert that comes in. Also of note is the fact that active checks of the service as disabled, while passive checks are enabled. This means that the service will never be actively checked - all alert information will have to be sent in passively by the *nsca client* on the **firestorm** host.

Configuring TCP Wrappers

Now you're going to have to modify the */etc/hosts.deny* file on the host called **firestorm**. In order to have the TCP wrappers send an alert to the monitoring host whenever a connection attempt is denied, you'll have to add a line similiar to the following:

```
ALL: ALL: RFC931: twist (/usr/local/nagios/libexec/eventhandlers/handle_tcp_wrapper %h %d) &
```

This line assumes that there is a script called *handle_tcp_wrapper* in the */usr/local/nagios/libexec/eventhandlers/* directory on **firestorm**. The directory and script name can be changed to whatever you want.

Writing The Script

The last thing you need to do is write the *handle_tcp_wrapper* script on **firestorm** that will send the alert back to the monitoring host. It might look something like this:

```
#!/bin/sh

/usr/local/nagios/libexec/eventhandlers/submit_check_result firestorm "TCP Wrappers" 2 "Denied $2-
$1" > /dev/null 2> /dev/null
```

Notice that the *handle_tcp_wrapper* script calls the *submit_check_result* script to actually send the alert back to the monitoring host. Assuming your monitoring host is called **monitor**, the *submit check_result* script might look like this (you'll have to modify this to specify the proper location of the *send_nsca* program on **firestorm**):

```
#!/bin/sh

# Arguments
#       $1 = name of host in service definition
#       $2 = name/description of service in service definition
#       $3 = return code
#       $4 = output

/bin/echo -e "$1\t$2\t$3\t$4\n" | /usr/local/nagios/bin/send_nsca monitor -c
/usr/local/nagios/etc/send_nsca.cfg
```

Finishing Up

You've now configured everything you need to, so all you have to do is restart the *inetd* process on **firestorm** and restart Nagios on your monitoring server. That's it! When the TCP wrappers on **firestorm** deny a connection attempt, you should be getting alerts in Nagios. The plugin output for the alert will look something like the following:

```
Denied sshd2-sdn-ar-002mnminnP321.dialsprint.net
```

Securing Nagios

Introduction

This is intended to be a brief overview of some things you should keep in mind when installing Nagios, so as to not set it up in an insecure manner. This document is new, so if anyone has additional notes or comments on securing Nagios, please drop me a note at nagios@nagios.org

Do Not Run Nagios As Root!

Nagios doesn't need to run as root, so don't do it. Even if you start Nagios at boot time with an init script, you can force it to drop privileges after startup and run as another user/group by using the nagios_user and nagios_group directives in the main config file.

If you need to execute event handlers or plugins which require root access, you might want to try using sudo.

Enable External Commands Only If Necessary

By default, external commands are disabled. This is done to prevent an admin from setting up Nagios and unknowingly leaving its command interface open for use by "others".. If you are planning on using event handlers or issuing commands from the web interface, you will have to enable external commands. If you aren't planning on using event handlers or the web interface to issue commands, I would recommend leaving external commands disabled.

Set Proper Permissions On The External Command File

If you enable external commands, make sure you set proper permissions on the */usr/local/nagios/var/rw* directory. You only want the Nagios user (usually *nagios*) and the web server user (usually *nobody*) to have permissions to write to the command file. If you've installed Nagios on a machine that is dedicated to monitoring and admin tasks and is not used for public accounts, that should be fine.

If you've installed it on a public or multi-user machine, allowing the web server user to have write access to the command file can be a security problem. After all, you don't want just any user on your system controlling Nagios through the external command file. In this case, I would suggest only granting write access on the command file to the *nagios* user and using something like CGIWrap to run the CGIs as the *nagios* user instead of *nobody*.

Instructions on setting up permissions for the external command file can be found here.

Require Authentication In The CGIs

I would strongly suggest requiring authentication for accessing the CGIs. Once you do that, read the documentation on the default rights that authenticated contacts have, and only authorize specific contacts for additional rights as necessary. Instructions on setting up authentication and configuring authorization rights can be found here. If you disable the CGI authentication features using the use_authentication directive in the CGI config file, the command CGI will refuse to write any commands to the external command file. After all, you don't want the world to be able to control Nagios do you?

Use Full Paths In Command Definitions

When you define commands, make sure you specify the *full path* to any scripts or binaries you're executing.

Hide Sensitive Information With $USERn$ Macros

The CGIs read the main config file and object config file(s), so you don't want to keep any sensitive information (usernames, passwords, etc) in there. If you need to specify a username and/or password in a command definition use a $USERn$ macro to hide it. $USERn$ macros are defined in one or more resource files. The CGIs will not attempt to read the contents of resource files, so you can set more restrictive permissions (600 or 660) on them. See the sample *resource.cfg* file in the base of the Nagios distribution for an example of how to define $USERn$ macros.

Strip Dangerous Characters From Macros

Use the illegal_macro_output_chars directive to strip dangerous characters from the $OUTPUT$ and $PERFDATA$ macros before they're used in notifications, etc. Dangerous characters can be anything that might be interpreted by the shell, thereby opening a security hole. An example of this is the presence of backtick (`) characters in the $OUTPUT$ and/or $PERFDATA$ macros, which could allow an attacker to execute an arbitrary command as the nagios user (one good reason not to run Nagios as the root user).

Tuning Nagios For Maximum Performance

Introduction

So you've finally got Nagios up and running and you want to know how you can tweak it a bit... Here are a few things to look at for optimizing Nagios. Let me know if you think of any others...

Optimization Tips:

1. **Use aggregated status updates**. Enabling aggregated status updates (with the aggregate_status_updates option) will greatly reduce the load on your monitoring host because it won't be constantly trying to update the status log. This is especially recommended if you are monitoring a large number of services. The main trade-off with using aggregated status updates is that changes in the states of hosts and services will not be reflected immediately in the status file. This may or may not be a big concern for you.

2. **Use a ramdisk for holding status data**. If you're using the standard status log and you're *not* using aggregated status updates, consider putting the directory where the status log is stored on a ramdisk. This will speed things up quite a bit (in both the core program and the CGIs) because it saves a lot of interrupts and disk thrashing.

3. **Check service latencies to determine best value for maximum concurrent checks**. Nagios can restrict the number of maximum concurrently executing service checks to the value you specify with the max_concurrent_checks option. This is good because it gives you some control over how much load Nagios will impose on your monitoring host, but it can also slow things down. If you are seeing high latency values (> 10 or 15 seconds) for the majority of your service checks (via the extinfo CGI), you are probably starving Nagios of the checks it needs. That's not Nagios's fault - its yours. Under ideal conditions, all service checks would have a latency of 0, meaning they were executed at the exact time that they were scheduled to be executed. However, it is normal for some checks to have small latency values. I would recommend taking the minimum number of maximum concurrent checks reported when running Nagios with the **-s** command line argument and doubling it. Keep increasing it until the average check latency for your services is fairly low. More information on service check scheduling can be found here.

4. **Use passive checks when possible**. The overhead needed to process the results of passive service checks is much lower than that of "normal" active checks, so make use of that piece of info if you're monitoring a slew of services. It should be noted that passive service checks are only really useful if you have some external application doing some type of monitoring or reporting, so if you're having Nagios do all the work, this won't help things.

5. **Avoid using interpreted plugins**. One thing that will significantly reduce the load on your monitoring host is the use of compiled (C/C++, etc.) plugins rather than interpreted script (Perl, etc) plugins. While Perl scripts and such are easy to write and work well, the fact that they are compiled/interpreted at every execution instance can significantly increase the load on your monitoring host if you have a lot of service checks. If you want to use Perl plugins, consider compiling them into true executables using perlcc(1) (a utility which is part of the standard Perl distribution) or compiling Nagios with an embedded

Perl interpreter (see below).

6. **Use the embedded Perl interpreter**. If you're using a lot of Perl scripts for service checks, etc., you will probably find that compiling an embedded Perl interpreter into the Nagios binary will speed things up. In order to compile in the embedded Perl interpreter, you'll need to supply the *--enable-embedded-perl* option to the configure script before you compile Nagios. Also, if you use the *--with-perlcache* option, the compiled version of all Perl scripts processed by the embedded interpreter will be cached for later reuse.

7. **Optimize host check commands**. If you're checking host states using the check_ping plugin you'll find that host checks will be performed much faster if you break up the checks. Instead of specifying a *max_attempts* value of 1 in the host definition and having the check_ping plugin send 10 ICMP packets to the host, it would be much faster to set the *max_attempts* value to 10 and only send out 1 ICMP packet each time. This is due to the fact that Nagios can often determine the status of a host after executing the plugin once, so you want to make the first check as fast as possible. This method does have its pitfalls in some situations (i.e. hosts that are slow to respond may be assumed to be down), but I you'll see faster host checks if you use it. Another option would be to use a faster plugin (i.e. check_fping) as the *host_check_command* instead of check_ping.

8. **Don't use agressive host checking**. Unless you're having problems with Nagios recognizing host recoveries, I would recommend *not* enabling the use_aggressive_host_checking option. With this option turned off host checks will execute much faster, resulting in speedier processing of service check results. However, host recoveries can be missed under certain circumstances when this it turned off. For example, if a host recovers and all of the services associated with that host stay in non-OK states (and don't "wobble" between different non-OK states), Nagios may miss the fact that the host has recovered. A few people may need to enable this option, but the majority don't and I would recommend *not* using it unless you find it necessary...

9. **Increase external command check interval**. If you're processing a lot of external commands (i.e. passive checks in a distributed setup, you'll probably want to set the command_check_interval variable to **-1**. This will cause Nagios to check for external commands as often as possible. This is important because most systems have small pipe buffer sizes (i.e. 4KB). If Nagios doesn't read the data from the pipe fast enough, applications that write to the external command file (i.e. the NSCA daemon) will block and wait until there is enough free space in the pipe to write their data.

10. **Optimize hardware for maximum performance**. Your system configuration and your hardware setup are going to directly affect how your operating system performs, so they'll affect how Nagios performs. The most common hardware optimization you can make is with your hard drives. CPU and memory speed are obviously factors that affect performance, but disk access is going to be your biggest bottlenck. Don't store plugins, the status log, etc on slow drives (i.e. old IDE drives or NFS mounts). If you've got them, use UltraSCSI drives or fast IDE drives. An important note for IDE/Linux users is that many Linux installations do not attempt to optimize disk access. If you don't change the disk access parameters (by using a utility like **hdparam**), you'll loose out on a **lot** of the speedy features of the new IDE drives.

Using Macros In Commands

Macros

One of the features available in Nagios is the ability to use macros in command defintions. Immediately prior to the execution of a command, Nagios will replace all macros in the command with their corresponding values. This allows you to define a few generic commands to handle all your needs.

Macro Validity

Although macros can be used in all commands you define, not all macros may be "valid" in a particular type of command. For example, some macros may only be valid during service notification commands, whereas other may only be valid during host check commands. There are nine types of commands that Nagios recognizes and treats differently. They are as follows:

1. Service checks
2. Service notifications
3. Host checks
4. Host notifications
5. Service event handlers and/or a global service event handler
6. Host event handlers and/or a global host event handler
7. OCSP command
8. Service performance data commands
9. Host performance data commands

The table below lists all macros currently available in Nagios, along with a brief description of each and the types of commands in which they are valid. If a macro is used in a command in which it is invalid, it is replaced with an empty string. It should be noted that macros consist of all uppercase characters and are enclosed in **$** characters.

Macro Availability Chart

Macro Name	**Macro Description**	**Service Checks**	**Service Notifications**	**Host Checks**	**Host Notifications**	**Service Event Handlers & Global Service Event Handler & OCSP Command**	**Host Event Handlers & Global Host Event Handler**	**Service Performance Data Commands**	**Host Performance Data Commands**
$CONTACTNAME$	Short name for the contact (i.e. "jdoe") that is being notified of a host or service problem	No	Yes	No	Yes	No	No	No	No

$CONTACTALIAS$	Long name/description for the contact (i.e. "John Doe") being notified	No	Yes	No	Yes	No	No	No	No
$CONTACTEMAIL$	Email address of the contact being notified	No	Yes	No	Yes	No	No	No	No
$CONTACTPAGER$	Pager number/address of the contact being notified	No	Yes	No	Yes	No	No	No	No
$HOSTNAME$	Short name for the host (i.e. "biglinuxbox"). During a service notification, this refers to the host associated with the service.	No	Yes	No	Yes	Yes	Yes	Yes	Yes
$HOSTALIAS$	Long name/description for the host (i.e. "Big Linux Server")	No	Yes	No	Yes	Yes	Yes	Yes	Yes
$HOSTADDRESS$	The IP address of the host	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
$HOSTSTATE$	The current state of the host ("UP", "DOWN", or "UNREACHABLE")	No	Yes	No	Yes	Yes	Yes	Yes	Yes
$ARGn$	The nth argument passed to the service check command. Nagios supports up to 32 argument macros ($ARG1$ through $ARG32$).	Yes	No	No	No	No	No	No	No
$SERVICEDESC$	The long name/description of the service being monitored (i.e. "Main Website")	No	Yes	No	No	Yes	No	Yes	No
$SERVICESTATE$	The current status of the service being monitored ("WARNING", "UNKNOWN", "CRITICAL", or "OK")	No	Yes	No	No	Yes	No	Yes	No

$OUTPUT$	The text output from the service or host check (i.e. "FTP ok - 1 second response time"). For service notifications and event handlers, this will contain the text output from the service check. For host notifications and event handlers, this will contain the text output from the host check.	No	Yes	No	Yes	Yes	Yes	Yes	Yes
$PERFDATA$	This macro contains any performance data that may have been returned by the service or host check.	No	Yes	No	Yes	Yes	Yes	Yes	Yes
$EXECUTIONTIME$	This is the number of seconds that the host or service check took to execute (i.e. the amount of time the check was executing).	No	Yes	No	Yes	Yes	Yes	Yes	Yes
$LATENCY$	This is the number of seconds that a service check lagged behind its scheduled check time. For instance, if a check was scheduled for 03:04:15 and it didn't get executed until 03:14:17, there would be a check latency of 2 seconds.	No	Yes	No	No	Yes	No	Yes	No
$NOTIFICATIONTYPE$	Identifies the type of notification that is being sent ("PROBLEM", "RECOVERY", or "ACKNOWLEDGEMENT").	No	Yes	No	Yes	No	No	No	No

$NOTIFICATIONNUMBER$	The current notification number for the service or host. The notification number increases by one (1) each time a new notification is sent out for a host or service (except for acknowledgements). The notification number is reset to 0 when the host or service recovers (*after* the recovery notification has gone out). Acknowledgements do not cause the notification number to increase.	No	Yes	No	Yes	No	No	No	No
$DATETIME$	Date/time stamp (i.e. *Fri Oct 13 00:30:28 CDT 2000*)	No	Yes	No	Yes	Yes	Yes	Yes	Yes
$SHORTDATETIME$	Date/time stamp (i.e. *10-13-2000 00:30:28*)	No	Yes	No	Yes	Yes	Yes	Yes	Yes
$DATE$	Date stamp (i.e. *10-13-2000*)	No	Yes	No	Yes	Yes	Yes	Yes	Yes
$TIME$	Time stamp (i.e. *00:30:28*)	No	Yes	No	Yes	Yes	Yes	Yes	Yes
$TIMET$	Time stamp in time_t format (seconds since the UNIX epoch)	No	Yes	No	Yes	Yes	Yes	Yes	Yes
$LASTCHECK$	This is a timestamp in time_t format (seconds since the UNIX epoch) indicating the time at which a service or host check was last performed.	No	Yes	No	Yes	Yes	Yes	Yes	Yes
$LASTSTATECHANGE$	This is a timestamp in time_t format (seconds since the UNIX epoch) indicating the time at which a service or host last changed state.	No	Yes	No	Yes	Yes	Yes	Yes	Yes
$ADMINEMAIL$	Email address for the local administrator (of the host doing the monitoring)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

$ADMINPAGER$	Pager number/address for the local administrator	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
$STATETYPE$	The state type for the current service or host check ("HARD" or "SOFT"). Soft states occur when service or host checks return a non-OK state and are in the process of being retried. Hard states result when service or host checks have been checked a specified maximum number of times. Notifications are sent out only when hard state changes occur.	No	No	No	No	Yes	Yes	Yes	Yes
$SERVICEATTEMPT$	This refers to the number of the current service check retry. For instance, if this is the second time that the service is being rechecked, this will be the number two. Current attempt number is only useful when writing service event handlers for "soft" states that take a specific action based on the service retry number.	No	No	No	No	Yes	No	Yes	No
$HOSTATTEMPT$	This refers to the number of the current host check retry. For instance, if this is the second time that the host is being rechecked, this will be the number two. Current attempt number is only useful when writing host event handlers for "soft" states that take a specific action based on the host retry number.	No	No	No	No	No	Yes	No	Yes

$USERn$	The nth user-definable macro. User macros can be defined in one or more resource files . Nagios supports up to thrity-two user macros ($USER1$ through $USER32$).	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

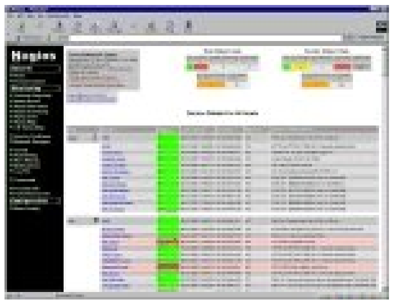
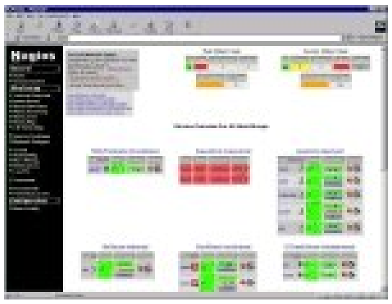
Information On The CGIs

Introduction

The various CGIs distributed with Nagios are described here, along with the authorization requirements for accessing and using each CGI. By default the CGIs require that you have authenticated to the web server and are authorized to view any information you are requesting. For more information on configuring your web server and CGI configuration file to allow for this, read the sections on setting up the web interface and CGI authorization.

Index

Status CGI

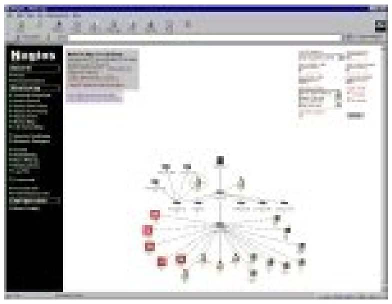
File Name: **status.cgi**

Description:
This is the most important CGI included with Nagios. It allows you to view the current status of all hosts and services that are being monitored. The status CGI can produce two main types of output - a status overview of all host groups (or a particular host group) and a detailed view of all services (or those associated with a particular host). Pretty icons can be associated with hosts by defining extended host and service information entries.

Authorization Requirements:

- If you are *authorized for all hosts* you can view all hosts **and** all services.
- If you are *authorized for all services* you can view all services.
- If you are an *authenticated contact* you can view all hosts and services for which you are a contact.

Status Map CGI



File Name: **statusmap.cgi**

Description:
This CGI creates a map of all hosts that you have defined on your network. The CGI uses Thomas Boutell's gd library (version 1.6.3 or higher) to create a PNG image of your network layout. The coordinates used when drawing each host (along with the optional pretty icons) are taken from extended host information definitions. If you'd prefer to let the CGI automatically generate drawing coordinates for you, use the default_statusmap_layout directive to specify a layout algorithm that should be used. If you can't seem to find this CGI, or if you have get errors when trying to compile or run it, read this FAQ.

Authorization Requirements:

- If you are *authorized for all hosts* you can view all hosts.
- If you are an *authenticated contact* you can view hosts for which you are a contact.

Note: Users who are not authorized to view specific hosts will see *unknown* nodes in those positions. I realize that they really shouldn't see *anything* there, but it doesn't make sense to even generate the map if you can't see all the host dependencies...

WAP Interface CGI

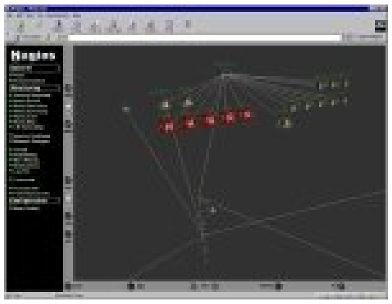
File Name: **statuswml.cgi**

Description:
This CGI serves as a WAP interface to network status information. If you have a WAP-enable device (i.e. an Internet-ready cellphone), you can view status information while you're on the go. Different status views include hostgroup summary, hostgroup overview, host detail, service detail, all problems, and unhandled problems. In addition to viewing status information, you can also disable notifications and checks and acknowledge problems from your cellphone. Pretty cool, huh?

Authorization Requirements:

- If you are *authorized for system information* you can view Nagios process information.
- If you are *authorized for all hosts* you can view status data for all hosts **and** services.
- If you are *authorized for all services* you can view status data for all services.
- If you are an *authenticated contact* you can view status data for all hosts and services for which you are a contact.

Status World CGI (VRML)



File Name: **statuswrl.cgi**

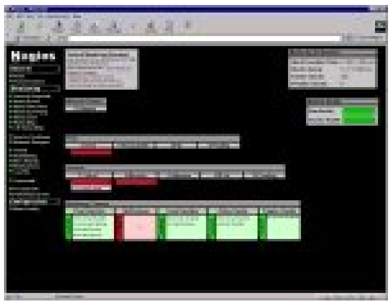
Description:
This CGI creates a 3-D VRML model of all hosts that you have defined on your network. Coordinates used when drawing the hosts (as well as pretty texture maps) are defined using extended host information definitions. If you'd prefer to let the CGI automatically generate drawing coordinates for you, use the default_statuswrl_layout directive to specify a layout algorithm that should be used. You'll need a VRML browser (like Cortona, Cosmo Player or WorldView) installed on your system before you can actually view the generated model.

Authorization Requirements:

- If you are *authorized for all hosts* you can view all hosts.
- If you are an *authenticated contact* you can view hosts for which you are a contact.

Note: Users who are not authorized to view specific hosts will see *unknown* nodes in those positions. I realize that they really shouldn't see *anything* there, but it doesn't make sense to even generate the map if you can't see all the host dependencies...

Tactical Overview CGI



File Name: **tac.cgi**

Description:
This CGI is designed to server as a "birds-eye view" of all network monitoring activity. It allows you to quickly see network outages, host status, and service status. It distinguishes between problems that have been "handled" in some way (i.e. been acknowledged, had notifications disabled, etc.) and those which have not been handled, and thus need attention. Very useful if you've got a lot of hosts/services you're monitoring and you need to keep a single screen up to alert you of problems.

Authorization Requirements:

- If you are *authorized for all hosts* you can view all hosts **and** all services.
- If you are *authorized for all services* you can view all services.
- If you are an *authenticated contact* you can view all hosts and services for which you are a contact.

Network Outages CGI

File Name: **outages.cgi**

Description:
This CGI will produce a listing of "problem" hosts on your network that are causing network outages. This can be particularly useful if you have a large network and want to quickly identify the source of the problem. Hosts are sorted based on the severity of the outage they are causing. More information on how the network outage CGI works can be found here.

Authorization Requirements:

- If you are *authorized for all hosts* you can view all hosts.
- If you are an *authenticated contact* you can view hosts for which you are a contact.

Configuration CGI

File Name: **config.cgi**

Description:
This CGI allows you to view objects (i.e. hosts, host groups, contacts, contact groups, time periods, services, etc.) that you have defined in your object configuration file(s).

Authorization Requirements:

- You must be *authorized for configuration information* in order to any kind of configuration information.

Command CGI

File Name: **cmd.cgi**

Description:
This CGI allows you to send commands to the Nagios process. Although this CGI has several arguments, you would be better to leave them alone. Most will change between different revisions of Nagios. Use the extended information CGI as a starting point for issuing commands.

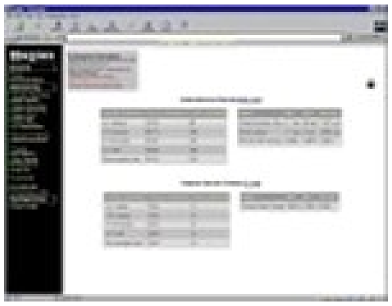
Authorization Requirements:

- You must be *authorized for system commands* in order to issue commands that affect the Nagios process (restarts, shutdowns, mode changes, etc.).
- If you are *authorized for all host commands* you can issue commands for all hosts **and** services.
- If you are *authorized for all service commands* you can issue commands for all services.
- If you are an *authenticated contact* you can issue commands for all hosts and services for which you are a contact.

Notes:

- If you have chosen not to use authentication with the CGIs, this CGI will *not* allow anyone to issue commands to Nagios. This is done for your own protection. I would suggest removing this CGI altogether if you decide not to use authentication with the CGIs.
- In order for the CGI to issue commands to Nagios, you will have to set the proper file and directory permissions as described in this FAQ.

Extended Information CGI



File Name: **extinfo.cgi**

Description:
This CGI allows you to view Nagios process information, host and service state statistics, host and service comments, and more. It also serves as a launching point for sending commands to Nagios via the command CGI. Although this CGI has several arguments, you would be better to leave them alone - they are likely to change between different releases of Nagios. You can access this CGI by clicking on the 'Network Health' and 'Process Information' links on the side navigation bar, or by clicking on a host or service link in the output of the status CGI.

Authorization Requirements:

- You must be *authorized for system information* in order to view Nagios process information.
- If you are *authorized for all hosts* you can view extended information for all hosts **and** services.
- If you are *authorized for all services* you can view extended information for all services.
- If you are an *authenticated contact* you can view extended information for all hosts and services for which you are a contact.

Event Log CGI

File Name: **showlog.cgi**

Description:
This CGI will display the log file. If you have log rotation enabled, you can browse notifications present in archived log files by using the navigational links near the top of the page.

Authorization Requirements:

- You must be *authorized for system information* in order to view the log file.

Alert History CGI

File Name: **history.cgi**

Description:
This CGI is used to display the history of problems with either a particular host or all hosts. The output is basically a subset of the information that is displayed by the log file CGI. You have the ability to filter the output to display only the specific types of problems you wish to see (i.e. hard and/or soft alerts, various types of service and host alerts, all types of alerts, etc.). If you have log rotation enabled, you can browse history information present in archived log files by using the navigational links near the top of the page.

Authorization Requirements:

- If you are *authorized for all hosts* you can view history information for all hosts **and** all services.
- If you are *authorized for all services* you can view history information for all services.
- If you are an *authenticated contact* you can view history information for all services and hosts for which you are a contact.

Notifications CGI

File Name: **notifications.cgi**

Description:
This CGI is used to display host and service notifications that have been sent to various contacts. The output is basically a subset of the information that is displayed by the log file CGI. You have the ability to filter the output to display only the specific types of notifications you wish to see (i.e. service notifications, host notifications, notifications sent to specific contacts, etc). If you have log rotation enabled, you can browse notifications present in archived log files by using the navigational links near the top of the page.

Authorization Requirements:

- If you are *authorized for all hosts* you can view notifications for all hosts **and** all services.
- If you are *authorized for all services* you can view notifications for all services.
- If you are an *authenticated contact* you can view notifications for all services and hosts for which you are a contact.

Trends CGI

File Name: **trends.cgi**

Description:
This CGI is used to create a graph of host or service states over an arbitrary period of time. In order for this CGI to be of much use, you should enable log rotation and keep archived logs in the path specified by the log_archive_path directive. The CGI uses Thomas Boutell's gd library (version 1.6.3 or higher) to create the trends image. If you can't seem to find this CGI or if you have get errors when trying to compile or run it, read this FAQ.

Authorization Requirements:

- If you are *authorized for all hosts* you can view trends for all hosts **and** all services.
- If you are *authorized for all services* you can view trends for all services.
- If you are an *authenticated contact* you can view trends for all services and hosts for which you are a contact.

Availability Reporting CGI

File Name: **avail.cgi**

Description:
This CGI is used to report on the availability of hosts and services over a user-specified period of time. In order for this CGI to be of much use, you should enable log rotation and keep archived logs in the path specified by the log_archive_path directive.

Authorization Requirements:

- If you are *authorized for all hosts* you can view availability data for all hosts **and** all services.
- If you are *authorized for all services* you can view availability data for all services.
- If you are an *authenticated contact* you can view availability data for all services and hosts for which you are a contact.

Alert Histogram CGI

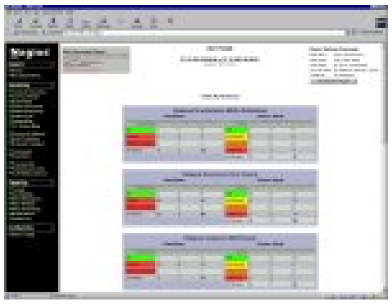
File Name: **histogram.cgi**

Description:
This CGI is used to report on the availability of hosts and services over a user-specified period of time. In order for this CGI to be of much use, you should enable log rotation and keep archived logs in the path specified by the log_archive_path directive. The CGI uses Thomas Boutell's gd library (version 1.6.3 or higher) to create the histogram image. If you can't seem to find this CGI or if you have get errors when trying to compile or run it, read this FAQ.

Authorization Requirements:

- If you are *authorized for all hosts* you can view histograms for all hosts **and** all services.
- If you are *authorized for all services* you can view histograms for all services.
- If you are an *authenticated contact* you can view histograms for all services and hosts for which you are a contact.

Alert Summary CGI



File Name: **summary.cgi**

Description:
This CGI provides some generic reports about host and service alert data, including alert totals, top alert producers, etc.

Authorization Requirements:

- If you are *authorized for all hosts* you can view summary information for all hosts **and** all services.
- If you are *authorized for all services* you can view summary information for all services.
- If you are an *authenticated contact* you can view summary information for all services and hosts for which you are a contact.

Custom CGI Headers and Footers

Introduction

If you're doing custom installs of Nagios for clients, you may want to have a custom header and/or footer displayed in the output of the CGIs. This is particularly useful for displaying support contact information, etc. to the end user.

It is important to note that custom headers and footers are **not** processed in any way before they are displayed. The contents of the header and footer include files are simply read and displayed in the CGI output. That means they can only contain information a web browser can understand (HTML, JavaScript, etc.).

How Does It Work?

You can include custom headers and footers in the output of the CGIs by dropping some appropriately named HTML files in the *ssi/* subdirectory of the Nagios HTML directory (i.e. */usr/local/nagios/share/ssi*).

Custom headers are included immediately after the **<BODY>** tag in the CGI output. Similarly, customer headers are included immediately before the closing **</BODY>** tag.

There are two types of customer headers and footers:

- **Global headers/footers**. These files should be named *common-header.ssi* and *common-footer.ssi*, respectively. If these files exist, they will be included in the output of all CGIs.
- **CGI-specific headers/footers**. These files should be named in the format *CGINAME-header.ssi* and *CGINAME-footer.ssi*, where *CGINAME* is the physical name of the CGI without the .cgi extension. For example, the header and footer files for the alert summary CGI (summary.cgi) would be named *summary-header.ssi* and *summary-footer.ssi*, respectively.

You are not required to use any custom headers or footers. You can use only a global header if you wish. You can use only CGI-specific headers and a global footer if you wish. Whatever you want. Really.
