NetSaint Documentation

Version 0.0.6
Last Updated: June 19th, 2000

About NetSaint

What Is NetSaint?

NetSaint is a network monitoring application. It is designed to run under Linux, although it should work under most other unices as well. Some of its features 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.

NetSaint is *not*...

- Designed to run under NT - it never has been and never will be.
- An SNMP manager. If that's what you need, look elsewhere.

System Requirements

The only requirement of running NetSaint 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 the core NetSaint distribution. 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 CGI)

Licensing

NetSaint is free software and may be used, copied, modified, etc. in accordance with the GNU General Public License version 2 or later. Information on the GPL license and open source software model can be found at www.opensource.org

History

- *Version 0.0.6b1 - 06/11/2000*
- Version 0.0.5 - 04/26/2000
- Version 0.0.4 - 09/02/1999
- Version 0.0.3 - 05/21/1999
- Version 0.0.2p1 - 04/18/1999
- Version 0.0.2 - 04/10/1999
- Version 0.0.1 - 03/14/1999

Known Issues

NetSaint is still an immature program, so there are bound to be a lot of bugs in it. The current list of known issues and bugs can be found at http://www.netsaint.org/bugs.html

Acknowledgements

Several people have contributed to NetSaint by either reporting bugs, suggesting improvements, writing plugins, etc. A list of some of the many contributors to the development of NetSaint can be found at http://www.netsaint.org/contributors.html. Unfortunately, this list is quite of of date. I've been getting so many bug reports, patches, suggestions, plugins, etc. that I can't keep up...

Comments And Feedback

I developed NetSaint for my own use. Once I was reasonably happy with it, I decided to release it so that others could use it. NetSaint is free software, so I don't get any compensation for the hours I spend working on it. In order to keep more versions coming, all I ask is that you give me some feedback. I need to know what doesn't work in the current version and what you want to see in future releases. Positive feedback is appreciated, as it helps assure me that NetSaint is actually being used and is working for people. You can email me at netsaint@netsaint.org

Downloading The Latest Version

You can check for new versions of NetSaint at the following sites:

- http://www.netsaint.org
- http://www.freshmeat.net (appindex 923738250)

Other Monitoring Utilities

In case you weren't aware, there are other network monitoring utilities available besides NetSaint. I think NetSaint is a pretty good contender, but I'm obviously biased... Have a look at the competition for yourself - here are links to a few of them:

- Angel Network Monitor
- Autostatus
- Big Brother
- Eclipse

- The Event Monitor Project
- MARS
- Mon
- Netup (French)
- NocMonitor
- NOCOL
- NodeWatch
- Over-CR
- PIKT
- RITW
- Spong
- Sysmon

Information On The CGIs

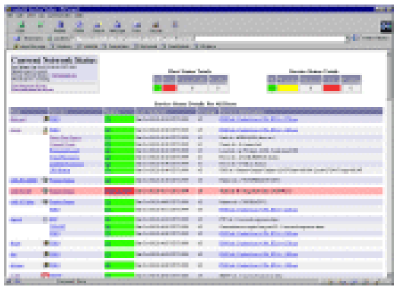
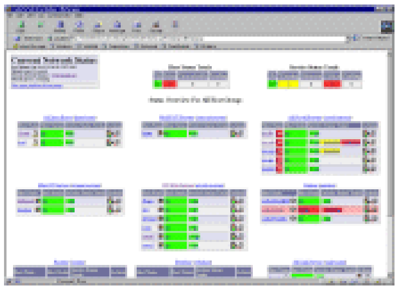
Introduction

This is a brief description of each CGI distributed with NetSaint, along with the various options that can be specified in the URL to control output. Authorization requirements for each CGI are also discussed.

Important: 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 NetSaint. 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 information entries in the CGI configuration file.

Authorization Requirements:

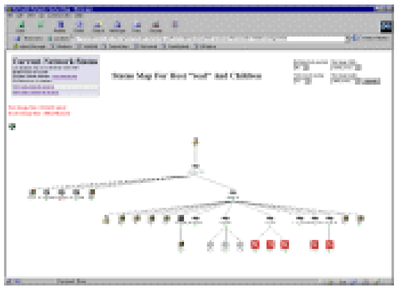
CGI Arguments:

Argument	Description
host=all	This will produce a detailed view of the status of all services being monitored with NetSaint
host=xxxx	This will produce a detailed view of the status of all services associated with host *xxxx*, where *xxxx* is the short name of the host as defined in the host configuration file.
hostgroup=all	This will produce an overview of all services (and their associated hosts) being monitored with NetSaint, grouped into various host groups.
hostgroup=xxxx	This will produce an overview of all services (and their associated hosts) belonging to host group *xxxx*, where *xxxx* is the short name of the host group as defined in the host configuration file.

- 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.

columns=x	This option may only be used in conjunction with the **hostgroup=all** argument. It allows you to control how many columns of hostgroups are displayed on the generated page. For instance, supplying *hostgroup=all&columns=4* as arguments to the CGI will produce an overview page that contains four columns of host groups.
style=detail	This option may only be used in conjunction with the **hostgroup** argument. Supplying this option will produce a detailed view of all services for hosts that are members of the hostgroup you specified. If you do not supply this option, the default action is to produce a status overview page.
nopopup	This option will suppress the host alert console window that gets displayed when one or more monitored hosts is either down or unreachable.

Status Map CGI



File Name: **statusmap.cgi**

Description:

This CGI dynamically 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. Pretty icons can be associated with the hosts in the generated image by defining extended host information entries in the CGI configuration file. If you can't seem to find this CGI, or if you have get errors when trying to compile it, read this FAQ.

Authorization Requirements:

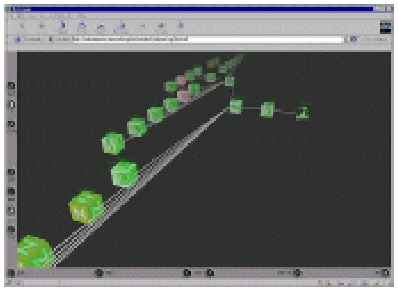
- If you are *authorized for all hosts* you can view all hosts.
- If you are an *authenticated contact* you can view hosts for which you are a contact.

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...

CGI Arguments:

Argument	Description
host=all	This will produce a network map of all hosts being monitored with NetSaint
host=xxxx	This will produce a network map of host *xxxx* and all of its child hosts, where *xxxx* is the short name of the host as defined in the host configuration file.
maxwidth=xxxx	This will limit the maximum width of the produced image to *xxxx* pixels.
maxheight=xxxx	This will limit the maximum height of the produced image to *xxxx* pixels.
hspacing=xx	This sets the horizontal spacing between host nodes to *xx* pixels.
vspacing=xx	This sets the vertical spacing between host nodes to *xx* pixels.
createimage	This instructs the CGI to create the PNG image instead of the HTML code with imagemap coordinates

Status World CGI (VRML)



File Name: **statuswrl.cgi**

Description:
This CGI dynamically creates a 3-D VRML model of all hosts that you have defined on your network. Images can be used as texture maps on host objects defining extended host information entries in the CGI configuration file. You'll need a VRML browser (like Cortona, Cosmo Player or WorldView) installed on your system before you can actually view the generated model.

CGI Arguments:

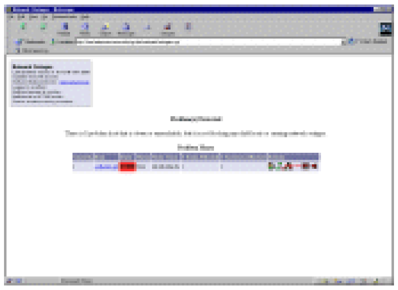
Argument	Description
host=all	This will produce a network model of all hosts being monitored with NetSaint
host=xxxx	This will produce a network model of host *xxxx* and all of its child hosts, where *xxxx* is the short name of the host as defined in the host configuration file.
notextures	This will prevent images from being texture mapped onto host objects.
notext	This will suppress the billboard text (host name and status) that is displayed over the host objects.

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...

Network Outages CGI



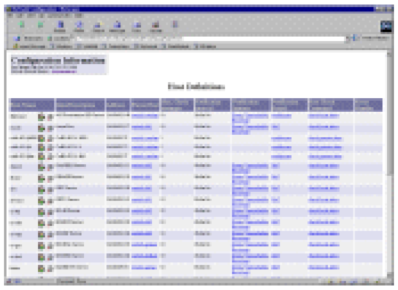
File Name: **outages.cgi**

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 host, host group, contact, contact group, time period, service, and command definitions that you have defined in your host configuration file(s).

CGI Arguments:

Argument	Description
type=xxxx	This option allows you to specify what type of definitions you would like to view. Valid options include "hosts", "hostgroups", "contacts", "contactgroups", "timeperiods", "commands", and "services".

Authorization Requirements:

- You must be *authorized for configuration information* in order to view contact, contact group, host group, time period, and command definitions. You will also be able to view all host and service definitions.
- If you are *authorized for all hosts* you can view all host **and** service definitions.
- If you are *authorized for all services* you can view all service definitions.
- If you are an *authenticated contact* you can view all host and service definitions for which you are a contact.

Command CGI

File Name: **cmd.cgi**

Description:
This CGI allows you to send commands to the NetSaint process. Although this CGI has several arguments, you would be better to leave them alone. Most will change between different revisions of NetSaint. Use the extended information CGI as a starting point for issuing commands.

Authorization Requirements:

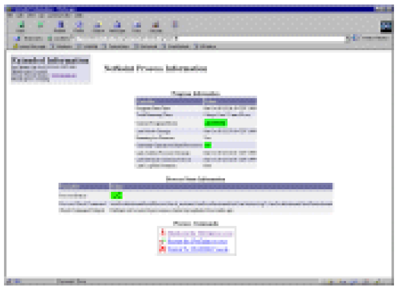
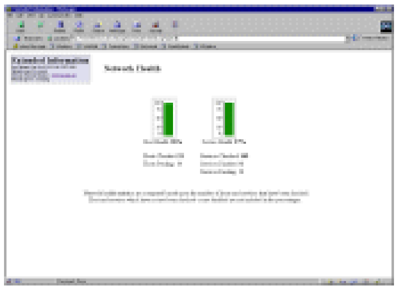
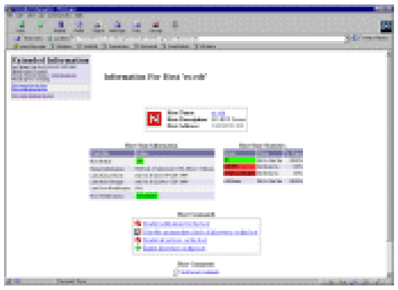
- You must be *authorized for system commands* in order to issue commands that affect the NetSaint process (restarts, shutdowns, mode changes, etc.).
- If you are *authorized for all host commands* you can issue commands for all hosts **and** services.
- If you are *authorized for all service commands* 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 NetSaint. 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 NetSaint, you will have to set the proper file and directory permissions as described in this FAQ.

Extended Information CGI

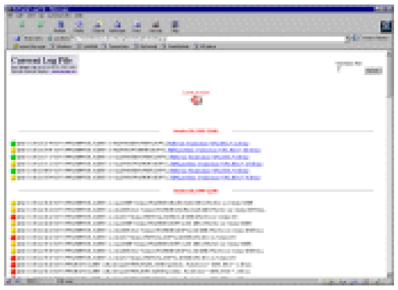
File Name: **extinfo.cgi**

Description:
This CGI allows you to view NetSaint process information, host and service state statistics, host and service comments, and more. It also serves as a launching point for sending commands to NetSaint 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 NetSaint. 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 NetSaint 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.

Log File 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.

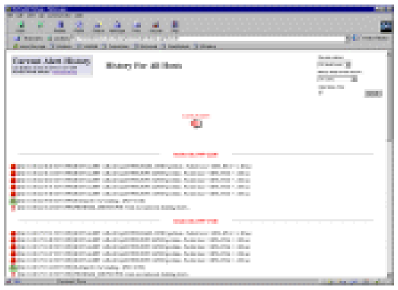
Authorization Requirements:

- You must be *authorized for system information* in order to view the log file.

CGI Arguments:

Argument	Description
archive=x	This option allows you to browse notifications in the x^{th} latest log archive. A value of 0 will cause the current log file to be used, a value of 1 will cause the most recent archived log to be used, and so on...
oldestfirst	This option allows view notifications with older entries at the top of the page and newer entries at the bottom. The CGI will normally reverse the log file so that newer log entries show up at the top of the page while older ones are at the bottom.

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.

CGI Arguments:

Argument	Description
host=all	This will display the history of all hosts being monitored with NetSaint
host=xxxx	This will display the history of host *xxxx*, where *xxxx* is the short name of the host as defined in the host configuration file.
type=x	This option allows you to control which types of historical alerts are displayed. As *x* is a numerical value generated by the CGI, I would suggest using the dropdown box to select the type of alerts you want to view.
statetype=x	This option allows you to control whether soft or hard alerts (or both) are displayed. As *x* is a numerical value generated by the CGI, I would suggest using the dropdown box to select the type of alerts you want to view.
archive=x	This option allows you to browse the history information in the x^{th} latest log archive. A value of 0 will cause the current log file to be used, a value of 1 will cause the most recent archived log to be used, and so on...
oldestfirst	This option allows view history information with older entries at the top of the page and newer entries at the bottom. The CGI will normally reverse the log file so that newer log entries show up at the top of the page while older ones are at the bottom.

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.

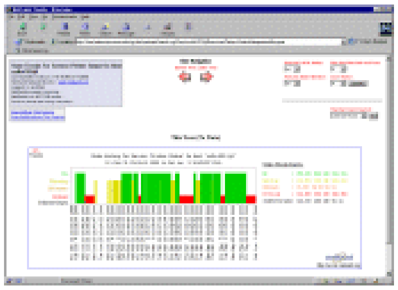
Authorization Requirements:

- 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.

CGI Arguments:

Argument	Description
host=all	This will display all notifications that have been sent out for all hosts (and their associated services) being monitored with NetSaint
host=xxxx	This will display all notifications that have been sent out for host *xxxx* (and its associated services), where *xxxx* is the short name of the host as defined in the host configuration file.
contact=all	This will display all service and host notifications that have been sent out to all contacts.
contact=xxxx	This will display all service and host notifications that have been sent out to contact *xxxx*, where *xxxx* is the short name of the contact as defined in the host configuration file.
type=x	This option allows you to control which types of notifications are displayed. As *x* is a numerical value generated by the CGI, I would suggest using the dropdown box to select the types of notifications you want to view.
archive=x	This option allows you to browse notifications in the *x*th latest log archive. A value of 0 will cause the current log file to be used, a value of 1 will cause the most recent archived log to be used, and so on...
oldestfirst	This option allows view notifications with older entries at the top of the page and newer entries at the bottom. The CGI will normally reverse the log file so that newer log entries show up at the top of the page while older ones are at the bottom.

Trends CGI



File Name: **trends.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 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.

What's New in Version 0.0.6

Note: This is a beta release of NetSaint. Any bugs should be reported to the netsaint-devel mailing list or to me at netsaint@netsaint.org.

Here are some of the things that have been changed or added since the 0.0.5 release...

New Features

1. **Multiple Parent Hosts**. You may now specify multiple parent hosts for each host definition. The order in which you specify parent hosts has no effect on how things are monitored. However, the statusmap and statuswrl CGIs will use the first parent host that you specify as the primary parent for purposes of drawing only.

2. **Passive Service Checks**. Previous to version 0.0.6, the only way NetSaint could check the status of any service was to actively check (i.e. perform the check itself). In 0.0.6, NetSaint can now access service check results from external apps. External apps can submit service check results to NetSaint via the newly added PROCESS_SERVICE_CHECK_RESULT external command. NetSaint will treat and act upon passive service checks in the same way it does "normal" active checks. More information on how passive service checks work can be found here.

3. **Volatile Services**. Service definitions have been extended to distinguish between normal services and newly added "volatile" services. Volatile services differ from normal services in that they get logged, generated a notification, and have an event handler run *every time* they are in a hard, non-OK state and the result of a service check shows the service to be in the same non-OK state. Volatile services are especially useful for monitoring asynchronous events like SNMP traps and security alerts. More information on how volatile services work can be found here.

4. **Notification Escalations**. Two new types of definitions have been added to the host config file to support *optional* escalation of service and host notifications. The two new definitions are service escalations and hostgroup escalations. More information on how notification escalations work can be found here.

5. **Distributed Monitoring**. NetSaint can now be configured to do distributed monitoring of your network. More gory details on how distributing monitoring works can be found here.

6. **Network Outages CGI**. A new network outages CGI has been added to help pinpoint the cause of network outages (from the view of NetSaint). More information on how the new CGI works can be found here.

7. **Trends CGI**. A new trends CGI has been added to allow you to view a graph of historical state

data for any given host or service over an arbitrary period of time. In order to produce useful results, this CGI expects that you have enabled log rotation and are storing historical log files in the directory specified by the log_archive_path variable.

8. **Sorting In The Status CGI**. This has been requested for some time now, and I finally got around to doing something about it. Service result entries in the status CGI (detail view) can be sorted by host name, service description, state, attempt number, and last check time. Sort orders can also be reversed. In order to sort the entries in the status CGI, click on the arrows located in the table headers.

9. **Audio Alerts In The Status CGI**. If you want to get an audible notification of network problems in the status CGI, you can use the audio_alerts in the CGI configuration file. You're able to specify different sounds to play for services that are in critical, warning, and unknown states, as well as hosts that are in unreachable or down states. If you configure audio files for multiple alert states, NetSaint will only play the sound that corresponds to the most critical problem.

10. **CGIs Now Use Stylesheets**. Everyone has their own idea of how the CGIs should look. I've moved most of the formatting code in the CGIs out to stylesheets. Each CGI has its own stylesheet that you can modify as you like. You'll need to have at least a 3.0 browser to actually be able to use the stylesheets - the output looks fairly dull without any style (duh!). BTW, Netscape and IE look like they both have rather horrid support of stylesheets when it comes to tables. Netscape is probably worse that IE, but they both have their problems...

11. **State Retention During Restarts**. Service and host status information can be preserved between program restarts. This is useful if there are pre-existing problems on your network (at the time NetSaint is restarted) and you don't want to receive initial notifications right away. This option will preserve state information, plugin output, last notification time, and state statistics for both hosts and services. In order to save state information between restarts you must enable the retain_state_information variable and specify a file in which to save the information by using the state_retention_file variable.

12. **Logging Of Initial States**. Initial host and service states can be logged if you find the need to do so. This is useful if you are using an application that scans the log file to determine long-term state statistics for services and hosts. Normally, states are only logged when there is a problem or recovery. You can enable initial service and host state logging by using the log_initial_states option in the main config file.

13. **Acknowledgement of Problems**. Users can now acknowlege host and service problems via the extinfo CGI. Acknowledgements can only be made after a host or service experiences a problem at at least one notification has been sent out. Upon making an acknowledgement of a problem, a comment will be added to the appropriate host or service, an acknowledgement notification is sent out, and future problem notifications will be temporarily disabled until the host or service changes state.

14. **Command Timeouts**. Command timeouts can now be specified globally for service checks, host checks, event handlers, and notifications. Timeout values are controled by the service_check_timeout, host_check_timeout, event_handler_timeout, and command_timeout options in the main config file.

15. **Macro Changes**. This one is important. I've changed the $SERVICESTATE$ And $HOSTSTATE$ macros to reflect the actual state of the service or host during recoveries, instead of setting the macro equal to "RECOVERY". For service recoveries the $SERVICESTATE$ macro is set to "OK" and for host recoveries the $HOSTSTATE$ macro is set to "UP". This was an inconsistency which had been annoying me for a long time, so I decided to change it and be done with it. Make sure to modify any event handlers you have that use the state macros! Also, a new macro ($NOTIFICATIONTYPE$) has been introduced, which can be used to identify what type of notification is being sent out. Values for the macro include "PROBLEM", "RECOVERY", and "ACKNOWLEDGEMENT". The $SUMMARY$ macro has been removed - at some point it stopped working and I just decided to kill it off. Lastly, the $OUTPUT$ macro can now be used in host notifications as well as service notifications. When the $OUTPUT$ macro is used in host notifications, it will contain the text returned from the host check command. More information on macros can be found here.

16. **Change In Location of CGI Config File**. The CGIs now expect that the CGI config file (nscgi.cfg) resides in the same directory as your main and host config files (usually */usr/local/netsaint/etc*). This was done to make things a bit more consistent and make it easier for creating RPMs.

17. **Developer Documentation**. I've added a new section to the documentation for developers who are wanting to interface third-party apps with NetSaint or exploit some of its internal capabilities (which are not yet available through the config files). Documentation is provided on the format of the various files that NetSaint uses, as well as internal functions which can be used to extend NetSaint's ability to read/save configuration information. I'll keep this information updated throughout the various releases of NetSaint. The developer documentation can be found here.

18. **Internal Overhauls**. A lot of the internal code in the core program and CGIs has been overhauled. End users won't see a difference, but it makes the code easier to work with. Some of the changes that have been made include changing static buffers in the data structures to use dynamically allocated memory, an overhaul of the internal logging code, and shared data structures and functions between the core and CGIs.

19. **The Usual Bug Fixes**. Would any new release ever be complete without bug fixes from previous versions?

Host 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 host configuration file can be created by running the '**make config**' command. The default name of the main configuration file is **hosts.cfg** - look for it in the NetSaint distribution directory or in the etc/ subdirectory of your installation.

Relationship of Data

In order to better help you understand how hosts, host groups, contacts, contact groups, services, etc. relate to each other I've throw together some diagrams. You can find them over in the theory of operation documentation.

Index

Host Definition

Format: **host[<host_name>]=<host_alias>;<address>;<parent_hosts>;<host_check_command>;<max_attempts>;<notification_interval>;<notification_period>;<notify_recovery>;<notify_down>;<notify_unreachable>;<event_handler>**
Example: **host[es-gra]=ES-GRA Server;192.168.0.1;;check-host-alive;3;120;24x7;1;1;1;**

A host definition is used to define a physical server, workstation, device, etc. that resides on your network. The different arguments to a host definition are described below.

<host_name>	This is a short name used to identify the host. It is used in host group and service definitions to reference this particular host. Hosts can have multiple services (which are monitored) associated with them. When used properly, the $HOSTNAME$ macro will contain this short name.
<host_alias>	This is a longer name or description used to identify the host. It is provided in order to allow you to more easily identify a particular host. When used properly, the $HOSTALIAS$ macro will contain this alias/description.
<address>	This is the IP address of the host. You can use a FQDN to identify the host, but if DNS services are not availble this could cause problems. When used properly, the $HOSTADDRESS$ macro will contain this address.
<parent_hosts>	This is a comma-delimited list of short names of the "parent" hosts for this particular host. Parent hosts are typically routers, switches, firewalls, etc. that lie between the monitoring host and a remote hosts. A router, switch, etc. which is closest to the remote host is considered to be that host's "parent". Read the "Determining Status and Reachability of Network Hosts" document in the theory of operation section for more information. If this host is on the same network segment as the host doing the monitoring (without any intermediate routers, etc.) the host is considered to be on the local network and will not have a parent host. Leave this value blank if the host does not have a parent host (i.e. it is on the same segment as the NetSaint host). The order in which you specify parent hosts has no effect on how things are monitored. However, the statusmap and statuswrl CGIs will use the first parent host that you specify as the primary parent for purposes of drawing only.
<host_check_command>	This is the *short name* of the command that should be used to check if the host is up or down. Typically, this command would try and ping the host to see if it is "alive". The command must return a status of OK (0) or NetSaint will assume the host is down. If you leave this argument blank, the host will not be checked - NetSaint will always assume the host is up. This is useful if you are monitoring printers or other devices that are frequently turned off.
<max_attempts>	This is the number of times that NetSaint will retry the host check command if it returns any state other than an OK state. Setting this value to 1 will cause NetSaint to generate an alert without retrying the host check again. Note: If you do not want to check the status of the host, you must still set this to a minimum value of 1. To bypass the host check, just leave the **<host_check_command>** option blank.
<notification_interval>	This is the number of "time units" to wait before re-notifying a contact that this server is *still* down. Unless you've changed the **interval_length** value in the main configuration file from the default value of 60, this number will mean minutes.
<notification_period>	This is the short name of the time period during which notifications of events for this host can be sent out to contacts. If a host goes down, becomes unreachable, or recoveries during a time which is not covered by the time period, no notifications will be sent out. Read the "Time Periods" document in the theory of operation section for more information.
<notify_recovery>	This value determines whether or not notifications should be sent to any contacts if the host is in a RECOVERY state. Set this value to **1** if notifications should be sent out about recovery states, **0** if they *shouldn't*. Note: If a contact is configured to not receive notifications of host recoveries, they will not be notified, regardless of this setting.
<notify_down>	This value determines whether or not notifications should be sent to any contacts if the host is in a DOWN state. Set this value to **1** if notifications should be sent out when the host goes down, **0** if they *shouldn't*. Note: If a contact is configured to not receive notifications about hosts that go down, they will not be notified, regardless of this setting.
<notify_unreachable>	This value determines whether or not notifications should be sent to any contacts if the host is in aa UNREACHABLE state. Set this value to **1** if notifications should be sent out when the host becomes unreachable, **0** if they *shouldn't*. Note: If a contact is configured to not receive notifications about unreachable hosts, they will not be notified, regardless of this setting.
<event_handler>	This is the *short name* of the command that should be run whenever a change in the state of the host is detected (i.e. whenever it goes down or recovers). Read the documentation on event handlers for a more detailed explanation of how to write scripts for handling events. If you do not wish to define an event handler for the host, leave this option blank (as shown in the example above).

Host Group Definition

Format: **hostgroup[<group_name>]=<group_alias>;<contact_groups>;<hosts>**
Example: **hostgroup[nt-servers]=All NT Servers;nt-admins;rosie,dev,liatris**

A host group definition is used to group one or more hosts together for the purposes of simplifying notifications. Each host that you define must be a member of at least one host group - even if it is the only host in that group. Hosts can be in more than one host group. When a host goes down, becomes unreachable, or recovers, NetSaint will find which host group(s) the host is a member of, get the contact group for each of those hostgroups, and notify all contacts associated with those contact groups. This may sound complex, but for most people it doesn't have to be. It does, however, allow for flexibility in determining who gets paged for what kind of problems. The different arguments to a host group definition are outlined below.

<group_name>	This is a short name used to identify the host group.
<group_alias>	This is a longer name or description used to identify the host group. It is provided in order to allow you to more easily identify a particular host group.
<contact_groups>	This is a list of the *short names* of the contact groups that should be notified whenever there are problems (or recoveries) with any of the hosts in this host group. Multiple contact groups should be separated by commas.
<hosts>	This is a list of the *short names* of hosts that should be included in this group. Multiple host names should be separated by commas.

Contact Definition

Format: **contact[<contact_name>]=<contact_alias>;<svc_notification_period>;<host_notification_period>;<svc_notify_recovery>;<svc_notify_critical>;<svc_notify_warning>;lt;host_notify_recovery>;<host_notify_down>;<host_notify_unreachable>;<service_notify_commands>;<host_notify_commands>;<email_address>;<pager>**
Example: **contact[egalstad]=Ethan Galstad;24x7;24x7;1;1;1;1;1;1;notify-by-email,notify-by-epager;host-notify-by-epager;egalstad@nospam.extension.umn.edu;pagegalstad@pagenet.com**

A contact definition is used to identify someone who should be contacted in the event of a problem on your network. The different arguments to a contact definition are described below.

<contact_name>	This is the short name used to identify the contact. It is referenced in contact group definitions. Under the right circumstances, the $CONTACTNAME$ macro will contain this value.
<contact_alias>	This is a longer name or description for the contact. Under the rights circumstances, the $CONTACTALIAS$ macro will contain this value.
<svc_notification_period>	This is the short name of the time period during which the contact can be notified about service problems or recoveries. You can think of this as an "on call" time for service notifications for the contact. Read the "Time Periods" document in the theory of operation section of the documentation for more information on how this works and potential problems that may result from improper use.
<host_notification_period>	This is the short name of the time period during which the contact can be notified about host problems or recoveries. You can think of this as an "on call" time for host notifications for the contact. Read the "Time Periods" document in the theory of operation section of the documentation for more information on how this works and potential problems that may result from improper use.
<svc_notify_recovery>	This value determines whether or not the contact will be notified of service recoveries. Set this value to **1** if the contact should be notified, **0** if they shouldn't. Note: If a service is configured to not send out notifications upon recovery, contacts will not be notified about recoveries for that service, regardless of this setting.
<svc_notify_critical>	This value determines whether or not the contact will be notified if a service is in a critical state. Set this value to **1** if the contact should be notified of critical states, **0** if they shouldn't. Note: If a service is configured to not send out notifications for critical states, contacts will not be notified about critical states for that service, regardless of this setting.
<svc_notify_warning>	This value determines whether or not the contact will be notified if a service is in either a warning or an unknown state. Set this value to **1** if the contact should be notified of warning/unknown states, **0** if they shouldn't. Note: If a service is configured to not send out notifications for warning/unknown states, contacts will not be notified about warning/unknown states for that service, regardless of this setting.

<host_notify_recovery>	This value determines whether or not the contact will be notified if any host recovers. Set this value to **1** if the contact should be notified of hosts that recover, **0** if they shouldn't. Note: If a host is configured to not send out notifications for recoveries, contacts will not be notified when the host recovers, regardless of this setting.
<host_notify_down>	This value determines whether or not the contact will be notified if any host goes down. Set this value to **1** if the contact should be notified of hosts that go down, **0** if they shouldn't. Note: If a host is configured to not send out notifications for down states, contacts will not be notified when the host goes down, regardless of this setting.
<host_notify_unreachable>	This value determines whether or not the contact will be notified if any host becomes unreachable. Set this value to **1** if the contact should be notified of hosts that become unreachable, **0** if they shouldn't. Note: If a host is configured to not send out notifications for unreachable states, contacts will not be notified when the host becomes unreachable, regardless of this setting.
<service_notify_commands>	This is a list of the *short names* of the commands used to notify the contact of a *service* problem or recovery. Multiple notification commands should be separated by commas. All notification commands are executed when the contact needs to be notified.
<host_notify_commands>	This is a list of the *short names* of the commands used to notify the contact of a *host* problem or recovery. Multiple notification commands should be separated by commas. All notification commands are executed when the contact needs to be notified.
<email_address>	This is the email address for the contact. Depending on how you configure your notification commands, it can be used to send out an alert email to the contact. Under the right circumstances, the $CONTACTEMAIL$ macro will contain this value. fs
<pager>	This is the pager number for the contact. It can also be an email address to a pager gateway (i.e. pagejoe@pagenet.com). Depending on how you configure your notification commands, it can be used to send out an alert page to the contact. Under the right circumstances, the $CONTACTPAGER$ macro will contact this value.

Contact Group Definition

Format: **contactgroup[<group_name>]=<group_alias>;<contacts>**
Example: **contactgroup[nt-admins]=NT Administrators;egalstad,jdoe**

A contact group definition is used to group one or more contacts together for the purpose of sending out alert/recovery notifications. When a host or service has a problem or recovers, NetSaint will find the appropriate contact groups to send notifications to, and notify all contacts in those contact groups. This may sound complex, but for most people it doesn't have to be. It does, however, allow for flexibility in determining who gets notified for particular events. The different arguments to a contact group definition are outlined below.

<group_name>	This is a short name used to identify the contact group.
<group_alias>	This is a longer name or description used to identify the contact group.
<contacts>	This is a list of the *short names* of contacts that should be included in this group. Multiple contact names should be separated by commas.

Command Definition

Format: **command[<command_name>]=<command_line>**
Example 1: **command[check-host-alive]=/usr/local/netsaint/libexec/check_ping $HOSTADDRESS$ 100 100 1000.0 1000.0**
Example 2: **command[check_pop]=/usr/local/netsaint/libexec/check_pop $HOSTADDRESS$**
Example 3: **command[check_disk]=/usr/local/netsaint/libexec/check_disk 85 95 $ARG1$**

A command definition is just that. It defines a command. Commands that can be defined include service checks, service notifications, service event handlers, host checks, host notifications, and host event handlers. Command definitions can contain macros, but you must make sure that you include only those macros that are "valid" for the circumstances when the command will be used. More information on what macros are available and when they are "valid" can be found here. The different arguments to a command definition are outlined below.

<command_name>	This is a short name used to identify the command. It is referenced in contact, host, and service definitions.
<command_line>	This is what is actually executed by NetSaint when the command is used for service or host checks, notifications, or event handlers. Before the command line is executed, all valid macros are replaced with their respective values. See the documentation on macros for determining when you can use different macros. Note that the command line is *not* surrounded in quotes.

Service Definition

Format: **service[<host>]=<description>;<volatile>;<check_period>;<max_attempts>;<check_interval>;<retry_interval>;<contactgroups>;<notification_interval>;<notification_period>;<notify_recovery>;<notify_critical>;<notify_warning>;<event_handler>;<check_command>**
Example 1: **service[rosie]=FTP;0;24x7;3;5;1;nt-admins;120;24x7;1;1;1;;check_ftp**
Example 2: **service[dev]=HTTP;0;24x7;3;5;1;nt-admins;240;24x7;1;1;1;;check_http2!192.168.0.2!/!88**
Example 3: **service[real]=Zombie Processes;0;24x7;3;5;1;linux-admins;240;24x7;1;1;1;;check_procs!5!10!Z**

A service definition is used to identify a "service" that runs on a host. The term "service" is used very loosely. It can mean an actual service that runs on the host (POP, SMTP, HTTP, etc.) or some other type of metric associated with the host (response to a ping, number of logged in users, free disk space, etc.). The different arguments to a service definition are outlined below.

<host>	This is the *short name* of the host that the service "runs" on or is associated with.
<description>	A description of the service, which may contain spaces, dashes, and colons (semicolons, apostrophes, and quotation marks should be avoided). No two services associated with the same host can have the same description.
<volatile>	This field is used to denote whether the service is "volatile". Services are normally *not* volatile. More information on volatile service and how they differ from normal services can be found here. Set this field to **1** to mark the service as being volatile, **0** to mark it as a normal service.
<check_period>	This is the short name of the time period that identifies when this service can be checked. Services checks are scheduled in such a way that they are only checked (or rechecked) during times that are valid within the specified service check time period. See the "Time Periods" documentation in the theory of operation section for more information on how time periods works and potentials problems with using them improperly.
<max_attempts>	This is the number of times that NetSaint will retry the service check if it returns any state other than an OK state. Setting this value to 1 will cause NetSaint to generate an alert (if the service check detected a problem) without retrying the service check again.
<check_interval>	This is the number of "time units" to wait before scheduling the next "regular" check of the service. "Regular" checks are those that occur when the service is in an OK state or when the service is in a non-OK state, but has already been rechecked **max_attempts** number of times. Unless you've changed the **interval_length** value in the main configuration file from the default value of 60, this number will mean minutes.
<retry_interval>	This is the number of "time units" to wait before scheduling a re-check of the service. Services are rescheduled at the retry interval when the have changed to a non-OK state. Once the service has been retried **max_attempts** times without a change in its status, it will revert to being scheduled at its "normal" rate as defined by the **check_interval** value. Unless you've changed the **interval_length** value in the main configuration file from the default value of 60, this number will mean minutes.
<contactgroups>	This is a comma-delimited list of the short names of contact groups that should be notified about problems or recoveries for this service. If a problem or recovery occurs for this service, NetSaint will attempt to notify all the contacts in each contact group (depending on the notification options that are set below).
<notification_interval>	This is the number of "time units" to wait before re-notifying a contact that this service is *still* at a non-OK state. Unless you've changed the **interval_length** value in the main configuration file from the default value of 60, this number will mean minutes.
<notification_period>	This is the short name of the time period that identifies when notifications about problems or recoveries for this service may be sent out. If a service problem or recovery occurs outside valid times within this time period, notifications will not be sent out. See the "Time Periods" documentation in the theory of operation section for more information on how time periods works and potentials problems with using them improperly.
<notify_recovery>	This value determines whether or not alert notifications will be generated if the service recovers from a non-OK state. Set this value to **1** if the service should generate alerts for recoveries, **0** if it shouldn't. Note: If a contact is configured to not receive recovery notifications, they will not be notified of any recoveries for this service, regardless of this setting.
<notify_critical>	This value determines whether or not alert notifications will be generated if the service is in a CRITICAL state. Set this value to **1** if the service should generate alerts for critical states, **0** if it shouldn't. Note: If a contact is configured to not receive critical notifications, they will not be notified of any critical states for this service, regardless of this setting.
<notify_warning>	This value determines whether or not alert notifications will be generated if the service is in a WARNING or UNKNOWN state. Set this value to **1** if the service should generate alerts for warning/unknown states, **0** if it shouldn't. Note: If a contact is configured to not receive warning/unknown notifications, they will not be notified of any warning/unknown states for this service, regardless of this setting.
<event_handler>	This is the *short name* of the command that should be run whenever a change in the status of the services is detected (i.e. whenever it goes down or recovers). Read the documentation on event handlers for a more detailed explanation of how to write scripts for handling events. If you do not wish to define an event handler for the service, leave this option blank (as shown in the examples above).
<check_command>	This is the command that NetSaint will run in order to check the status of the service. There are three command formats that can be used: **1. "Vanilla" Command:** The command name is just the name of command that was previously defined. Example 1 above shows this type of command. **2. Command w/ Arguments:** This is basically the same as the "vanilla" command style, but with command options separated by a **!** character. Example 2 above shows this type of command. Arguments are separated from the command name (and other arguments) with the **!** character. The command should be defined to make use of the $ARGx$ macros. In Example 2 above, $ARG1$ would resolve to **134.84.92.128**, $ARG2$ would resolve to **/**, and $ARG3$ would resolve to **88** for that particular service. Note: NetSaint will handle a maximum of sixteen command line arguments ($ARG1$ through $ARG16$). **3. "Raw" Command Line:** You may optionally specify an actual command line to be executed. To do so you must enclose the entire command line in double quotes. The outer double quotes will be stripped off before the command is actually executed. No macros are processed inside of raw command lines. Note: I haven't really tested this format too much, but it should work. Remember that the command must return a proper status level. See the documentation on writing plugins for numeric codes for each status level.

Time Period Definition

Format: **timeperiod[<timeperiod_name>]=<timeperiod_alias>;<sunday_ranges>;<monday_ranges>;<tuesday_ranges>;<wenesday_ranges>;<thursday_ranges>;<friday_ranges>;<saturday_ranges>;**
Example 1: **timeperiod[24x7]=All Day, Every Day;00:00-24:00;00:00-24:00;00:00-24:00;00:00-24:00;00:00-24:00;00:00-24:00;00:00-24:00**
Example 2: **timeperiod[workhours]="Normal" Working Hours;;09:00-17:00;09:00-17:00;09:00-17:00;09:00-17:00;09:00-17:00;**
Example 3: **timeperiod[none]=No Time Is A Good Time;;;;;;;**
Example 4: **timeperiod[nonworkhours]=Non-Work Hours;00:00-24:00;00:00-09:00,17:00-24:00;00:00-09:00,17:00-24:00;00:00-09:00,17:00-24:00;00:00-09:00,17:00-24:00;00:00-09:00,17:00-24:00;00:00-24:00**

A time period is a list of times during various days that are considered to be "valid" times for notifications and service checks. It consists one or more time periods for each day of the week that "rotate" once the week has come to an end. Exceptions to the normal weekly time range rotations are not suported.

<timeperiod_name>	This is a short name used to identify the time period.
<timeperiod_alias>	This is a longer name or description used to identify the time period.
<xday_ranges>	This is a comma-delimited list of time ranges that are "valid" times for a particular day of the week. Notice that there are seven different days for which you must define time ranges (Sunday through Saturday). Each time range is in the form of **HH:MM-HH:MM**, where hours are specified on a 24 hour clock. For example, **00:15-24:00** means 12:15am in the morning for this day until 12:20am midnight (a 23 hour, 45 minute total time range). If you leave a particular day's time range blank, it means that there are no "valid" times for that day.

Service Escalation Definition

Format:	**serviceescalation[<host>;<description>]=<first_notification>-<last_notificiation>;<contact_groups>**
Examples:	**serviceescalation[real;Zombie Processes]=3-5;linux-admins,managers** **serviceescalation[dev;HTTP]=6-0;nt-admins,managers,everyone**

A service escalation definition is *completely optional* and is used to escalate notifications for a particular service. More information on how notification escalations work can be found here.

<host>	This is the *short name* of the host that the service "runs" on or is associated with.
<description>	A description of the service, which may contain spaces, dashes, and colons (semicolons, parentheses, and apostrophes are not allowed). No two services associated with the same host can have the same description.
<first_notification>	This is a number that identifies the *first* notification for which this escalation is effective. For instance, if you set this value to 3, this escalation will only be used if the service is in a non-OK state long enough for a third escalation to go out.
<last_notification>	This is a number that identifies the *last* notification for which this escalation is effective. For instance, if you set this value to 5, this escalation will not be used if more than five notifications are sent out for the specified service. Setting this value to 0 means to keep using this escalation entry forever (no matter how many notifications go out).
<contact_groups>	This is a list of the *short names* of the contact groups that should be notified when a service notification is escalated. Multiple contact groups should be separated by commas.

Host Group Escalation Definition

Format:	**hostgroupescalation[<group_name>]=<first_notification>-<last_notificiation>;<contact_groups>**
Examples:	**hostgroupescalation[nt-servers]=3-5;nt-admins,managers** **hostgroupescalation[nt-servers]=6-0;nt-admins,managers,everyone**

A host group escalation definition is *completely optional* and is used to escalate notifications for hosts in a particular hostgroup. More information on how notification escalations work can be found here.

<group_name>	This is a short name used to identify the host group (as previously defined in a hostgroup definition) that the escalation should apply to.
<first_notification>	This is a number that identifies the *first* notification for which this escalation is effective. For instance, if you set this value to 3, this escalation will only be used if a host in the hostgroup is down or unreachable long enough for a third escalation to go out.
<last_notification>	This is a number that identifies the *last* notification for which this escalation is effective. For instance, if you set this value to 5, this escalation will not be used if more than five notifications are sent out for any particular host in the specified hostgroup. Setting this value to 0 means to keep using this escalation entry forever (no matter how many notifications go out).
<contact_groups>	This is a list of the *short names* of the contact groups that should be notified when a host notification is escalated. Multiple contact groups should be separated by commas.

External Commands

Introduction

NetSaint 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, NetSaint 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

Note: If external applications or CGIs will be issuing commands to NetSaint, you will have to grant the user that those processes run as permission to write to the command file. An outline of how to do this for CGIs can be found in this FAQ.

When Does NetSaint 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 NetSaint.

Using External Commands

External commands can be used to accomplish a variety of things while NetSaint is running. Example of what can be done include changing program modes, 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 NetSaint can be found in the *eventhandlers/* subdirectory of the NetSaint 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. The various commands that are available, along with their *command_id* and a description of their *command_arguments*, can be found in the table below.

Implemented Commands

This is a description of the external commands which have been implemented in NetSaint thus far. More commands will be added in future releases. 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 NetSaint 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 NetSaint 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 NetSaint 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 NetSaint 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 NetSaint). 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 NetSaint). Notifications are automatically re-enabled when NetSaint restarts. This command is most often used in conjunction with redundant monitoring hosts.
ENTER_STANDBY_MODE	<execution_time>	This will change the current program mode to *Standby* at the time specified by the *execution time* argument.
ENTER_ACTIVE_MODE	<execution_time>	This will change the current program mode to *Active* at the time specified by the *execution time* argument.
SHUTDOWN_PROGRAM	<execution_time>	This will cause NetSaint to shutdown at the time specified by the *execution_time* argument. Note: NetSaint cannot be restarted via the web interface once it has been shutdown.
RESTART_PROGRAM	<execution_time>	This will cause NetSaint 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 NetSaint. 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 NetSaint 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 NetSaint 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, NetSaint will not keep requeuing checks for a later time, but will not actually execute any checks. This essentially puts NetSaint 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.

Passive Service Checks

Introduction

Beginning with release 0.0.6, NetSaint can now process service check results that are submitted by external applications. Service checks which are performed and submitted to NetSaint by external apps are called *passive* checks. Passive checks can be contrasted with *active* checks, which are service checks that have been initiated by NetSaint.

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 NetSaint
- 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 NetSaint, 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 NetSaint through the external command file.

The next time NetSaint 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.

NetSaint 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 NetSaint 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, -1=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 NetSaint, a service must have alread been defined in the host configuration file! NetSaint will ignore all check results for services that had not been configured before it was last (re)started.

An example shell script of how to submit passive service check results to NetSaint 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 NetSaint 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 NetSaint, I've developed the nsca addon. The addon consists of a daemon that runs on the NetSaint 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...

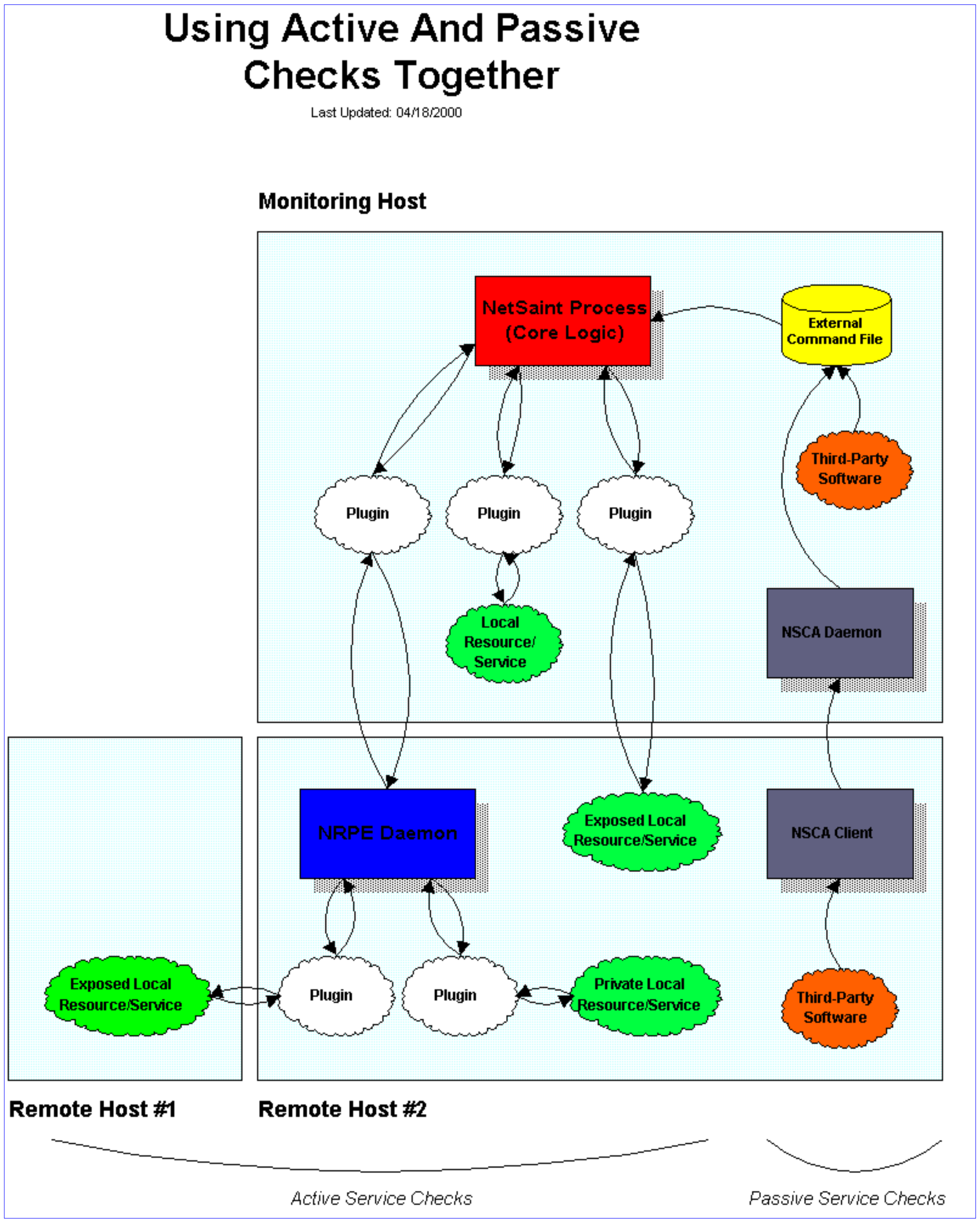
Using Both Active And Passive Service Checks

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 NetSaint's external command file. One of the applications resides on the same host as NetSaint, 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 NetSaint.

The items on the left side of the image represent active service checks that NetSaint 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.


Using Active And Passive Checks Together
Last Updated: 04/18/2000
Monitoring Host
NetSaint Process (Core Logic)
External Command File
Third-Party Software
Plugin
Plugin
Plugin
Local Resource/ Service
NSCA Daemon
NRPE Daemon
Exposed Local Resource/Service
NSCA Client
Exposed Local Resource/Service
Plugin
Plugin
Private Local Resource/Service
Third-Party Software
Remote Host #1
Remote Host #2
Active Service Checks
Passive Service Checks

Volatile Services

Introduction

Beginning with release 0.0.6 of NetSaint, service definitions have been extended to allow for a distinction between "normal" services and "volatile" services. The <*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 NetSaint know about port scans, you could do the following..

In NetSaint:

- Configure a service called *Port Scans* and associate it with the host that PortSentry is running on.
- Set the <*max_attempts*> option in the service definition to 1. This will tell NetSaint to immediate force the service into a hard state when a non-OK state is reported.

- Set the <*check_time*> option in the service definition to a timeperiod that contains *no* valid time ranges. This will prevent NetSaint 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/netsaint/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/netsaint/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 NetSaint command file to cause
# it to process a service check result

echocmd="/bin/echo"

CommandFile="/usr/local/netsaint/var/rw/netsaint.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 NetSaint 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 NetSaint
- NetSaint reads the command file, recognized the port scan entry as a passive service check
- NetSaint 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)

Notification Escalations

Introduction

Beginning with release 0.0.6, NetSaint supports *optional* escalation of contact notifications for specific services or hosts within specific hostgroups. 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 the host config file. Service escalation definitions are used to escalate notifications for a particular service.

Host Notification Escalations

Escalation of host notifications is accomplished by defining hostgroup escalation definitions in the host config file. Hostgroup escalation definitions are used to escalate host notifications for all hosts in a particular hostgroup. The examples I provide below all use service escalation definitions, but 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:

```
service[dev]=HTTP;0;24x7;3;5;1;nt-admins;240;24x7;1;1;1;;check_http
serviceescalation[dev;HTTP]=3-5;nt-admins,managers
serviceescalation[dev;HTTP]=6-10;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. In the example above, this would mean that the *nt-admins* contact group would be the only group that was notified during these "holes".

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:

```
service[dev]=HTTP;0;24x7;3;5;1;nt-admins;240;24x7;1;1;1;;check_http
serviceescalation[dev;HTTP]=3-5;nt-admins,managers
```

serviceescalation[dev;HTTP]=6-0;nt-admins,managers,everyone

The default contact group for the service 'HTTP' on host 'dev' is the group named *nt-admins*. 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:

serviceescalation[dev;HTTP]=3-5;nt-admins,managers
serviceescalation[dev;HTTP]=4-0;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:

serviceescalation[dev;HTTP]=3-5;nt-admins,managers
serviceescalation[dev;HTTP]=4-0;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.

Distributed Monitoring

Introduction

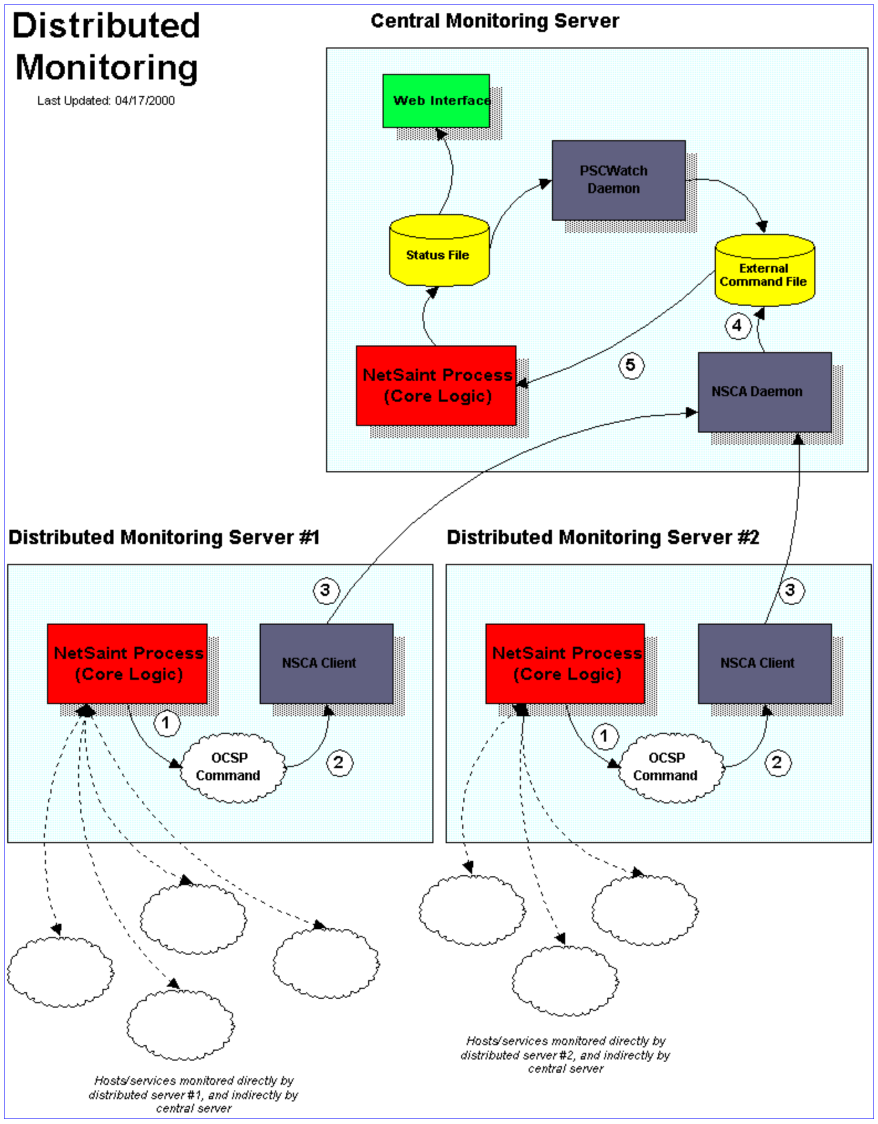
Beginning with release 0.0.6, NetSaint can *optionally* be configured to support distributed monitoring of network services and resources. I'll try to briefly explan how this can be accomplished...

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 NetSaint, this becomes quite important.

Reference Diagram

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


Distributed Monitoring
Last Updated: 04/17/2000
Central Monitoring Server
Web Interface
PSCWatch Daemon
Status File
External Command File
NetSaint Process (Core Logic)
NSCA Daemon
4
5
Distributed Monitoring Server #1
Distributed Monitoring Server #2
NetSaint Process (Core Logic)
NSCA Client
OCSP Command
1
2
3
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 NetSaint, 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 NetSaint and monitors the services on the hosts in the cluster. A distributed server is usually a bare-bones installation of NetSaint. 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 actively checked from the central server, the active checks are only performed at long intervals (as will be described later), 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 NetSaint from same host that NetSaint 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 NetSaint (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 NetSaint 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 NetSaint 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 host configuration file.
- The distributed server has its initial program mode set to *STANDBY*. 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 NetSaint. 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:

command[submit_check_result]=/usr/local/netsaint/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/netsaint/bin/send_nsca
central_server -c /usr/local/netsaint/var/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/netsaint/bin/* and */usr/local/netsaint/var/* directories, respectively.

That's it! We've sucessfully configured a remote host running NetSaint 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 NetSaint (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/netsaint/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 NetSaint.
5. The NetSaint 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 with:

- The web interface (optional, but recommended)
- Notifications (optional, but recommended)
- Event handlers (optional)
- Active service checks enabled (required)
- External command checks enabled (required)
- Passive service checks enabled (required)

There are two 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. NetSaint will ignore passive check results if they do not correspond to a service that has been defined.
- The normal *check_interval* argument for each service definition should be set to a long time interval (i.e. 24 hours or a week).

It is important that you set the *check_interval* argument for each service definition to a long interval. This will ensure that active service checks account for only a minimal load on the central server. We don't want to disable service checks, as it will be necessary to sometimes force NetSaint to actively check services (as discussed below).

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. While it does perform active checks of all services, it only does so at very long intervals, so lets disregard that fact. The main problem with relying completely on passive checks for monitoring is the fact that NetSaint 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 NetSaint isn't actively checking the services on the host, how will it know that there is a problem?

We can protect against this type of problem by using another addon to monitoring incoming passive check results...

Watchdog Daemon

In order to protect against situations where remote hosts may stop sending passive service checks into the central monitoring server, I've developed the pscwatch daemon. The daemon's sole purpose in life is to ensure that service checks are being either performed actively by the central server or being provided passively be distributed servers on a regular basis.

If the *pscwatch* daemon detects that a service check has not been performed within a given threshold of time, it will send a command to NetSaint via the external command file telling it to schedule an immediate active check of the service. When NetSaint performs an active check of the service, it will be able to tell if there is a real problem or not. Problem solved.

Note: If service checks are disabled, NetSaint will refuse to actively perform a service check. This is the reason why we don't want to disable active checks on the central server. Instead, we just set the normal check interval for all services to a very long time period.

Combining Distributed Monitoring With Redundancy

Nothing here yet...

Network Outages

Introduction

The outages CGI was added with release 0.0.6 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 NetSaint). All other hosts are up.

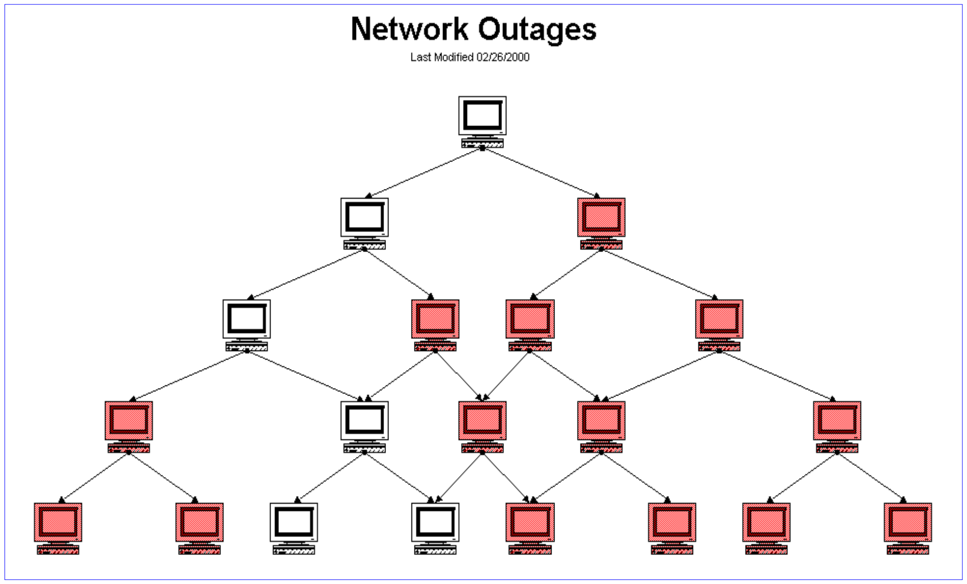
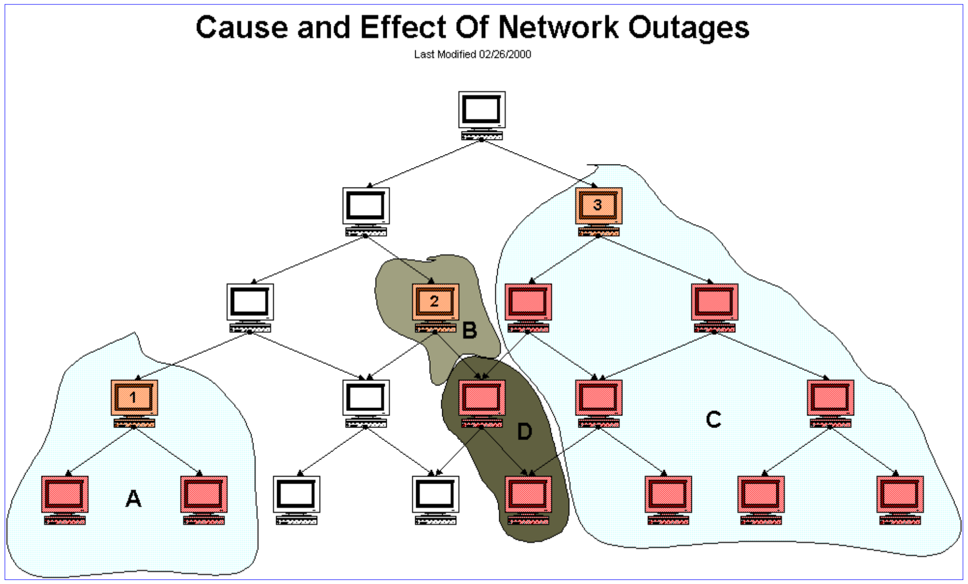


Diagram 2

This diagram pinpoints the causes of the network outages (from the view of NetSaint), 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.

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 can be created by running the '**make config**' command. The default name of the main configuration file is **netsaint.cfg -** look for it in the NetSaint distribution directory or in the etc/ subdirectory of your installation.

Index

Log File

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

This variable specifies where NetSaint should create its main log file. This should be the first variable that you define in your configuration file, as NetSaint will try to write errors that it finds in the rest of your configuration data to this file. This file is never deleted, pruned or rotated by NetSaint. I suggest adding a cron job to do log rotations every month or so (more often if you have a lot of alarms).

Host Configuration File

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

This specifies the host configuration file that NetSaint should use for monitoring. Host configuration files contain configuration data for hosts, host groups, contacts, contact groups, services, commands, etc. You can split your configuration information into several files and specify multiple **cfg_file=** statements to include each of them.

Status File

Format: **status_file=<file_name>**
Example: **status_file=/usr/local/netsaint/var/status.log**

This is the file that NetSaint 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 "status" CGI 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 NetSaint stops and recreated when it starts.

Temp File

Format: **temp_file=<file_name>**
Example: **temp_file=/usr/local/netsaint/var/netsaint.tmp**

This is the temporary file into which NetSaint redirects the standard output and error from the execution of plugins. The output from the plugins is scooped from the temp file and used for both display in the "status" CGI output and use in notification macros. This file is deleted after the plugin has been executed. This file is also used as a scratch file when NetSaint updates the status log.

Note: On most systems, the temp file will have to reside on the same filesystem as the status file, the log file, and the log file archive path.

Program Mode

Format: **program_mode=<a/s>**
Example: **program_mode=a**

This is the intial program mode that NetSaint should use when it starts or restarts. More information on program modes can be found here. Values are as follows:

- a = Active mode (default)
- s = Standby mode

Service Check Execution Option

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

This option determines whether or not NetSaint will execute service checks when it initially (re)starts. If this option is disabled, NetSaint 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. 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 NetSaint will accept passive service checks when it initially (re)starts. If this option is disabled, NetSaint will not accept any passive service checks. 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 NetSaint will run event handlers when it initially (re)starts. If this option is disabled, NetSaint will not run any host or service event handlers. 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 NetSaint 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/netsaint/var/archives/**

This is the directory where NetSaint 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 NetSaint 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 NetSaint 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>**
Example: **command_check_interval=1**

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. Each time NetSaint 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/netsaint/var/rw/netsaint.cmd**

This is the file that NetSaint 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 this FAQ. More information on external commands can be found here.

Comment File

Format: **comment_file=<file_name>**
Example: **comment_file=/usr/local/netsaint/var/comment.log**

This is the file that NetSaint 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/netsaint.lock**

This option specifies the location of the lock file that NetSaint 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 NetSaint process.

State Retention Option

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

This option determines whether or not NetSaint 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, NetSaint 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/netsaint/var/status.sav**

This is the file that NetSaint will use for storing service and host state information before it shuts down. When NetSaint 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 NetSaint reads in initial state information when it (re)starts. In order to make NetSaint retain state information between program restarts, you must enable the retain_state_information option.

Log Severity Level

Format: **log_level=<1-2>**
Example: **log_level=1**

This is the level of severity needed for service messages to be logged to the main log file. Values are as follows:

- 1 = Log services which are in WARNING, UNKNOWN, or CRITICAL states.
- 2 = Log only services which are in a CRITICAL state.

Notes:

- This should almost *always* be set to 1. If it isn't your mileage may vary, as I haven't really tested the consequences.

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

Syslog Severity Level

Format: **syslog_level=<1-2>**
Example: **syslog_level=1**

This is the level of severity needed for service messages to be logged to the syslog facility. Values are as follows:

- 1 = Log services which are in WARNING, UNKNOWN, or CRITICAL states.
- 2 = Log only services which are in a CRITICAL state.

Notes:

This should almost *always* be set to 1. If it isn't your mileage may vary, as I haven't really tested the consequences.

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 NetSaint 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 NetSaint 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 NetSaint 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 NetSaint 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 NetSaint 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 NetSaint 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 NetSaint 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 definition that you define in your host configuration file. 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 definition that you define in your host configuration file. 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 NetSaint will sleep before checking to see if the next service check in the scheduling queue should be executed. Note that NetSaint 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>**
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 NetSaint 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)

Service Interleave Factor

Format: **service_interleave_factor=<s|*n*>**
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 NetSaint 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 NetSaint 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.

- *n* = 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. You'll have to modify this value based on the system resources you have available on the machine that runs NetSaint, 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 NetSaint.

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**

Beginning with release 0.0.4, NetSaint tries to be a little smarter about how and when it checks the status of hosts. In general, disabling this option will allow NetSaint 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. If you want to know more about exactly what this option does, search the source code in the **netsaint.c** file for the string "**use_agressive_host_checking**" and read some of the comments I've added. Unless you have problems with NetSaint 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

Service Check Timeout

Format: **service_check_timeout=<seconds>**

Example: **service_check_timeout=60**

This is the maximum number of seconds that NetSaint 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.

Host Check Timeout

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

This is the maximum number of seconds that NetSaint 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.

Event Handler Timeout

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

This is the maximum number of seconds that NetSaint 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.

Notification Timeout

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

This is the maximum number of seconds that NetSaint 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.

Obsessive Compulsive Service Processor Timeout

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

This is the maximum number of seconds that NetSaint 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.

Obsess Over Services Option

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

This value determines whether or not NetSaint 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. 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. This option is useful for performing distributed monitoring. More information on distributed monitoring can be found here.

Administrator Email Address

Format: **admin_email=<email_address>**
Example: **admin_email=root**

This is the email address for the administrator of the local machine (i.e. the one that NetSaint 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@pagenet.com**

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

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 **nscgi.cfg**.

Index

Main Configuration File Location

Format: **main_config_file=<file_name>**
Example: **main_config_file=/usr/local/netsaint/etc/netsaint.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/netsaint/share**

This is the *physical* path where the HTML files for NetSaint are kept on your workstation or server. NetSaint 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=/netsaint**

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

Process Check Command

Format:	**process_check_command=<command_line>**
Example:	**process_check_command=/usr/local/netsaint/libexec/check_netsaint /usr/local/netsaint/var/status.log 5 '/usr/local/netsaint/bin/netsaint -d /usr/local/netsaint/etc/netsaint.cfg'**

This is the command that the CGIs should use to check the status of the NetSaint process. This provides the CGIs (as well as yourself) with some idea of whether or not NetSaint is still running. If the CGIs cannot determine whether or not NetSaint is running on the local machine, some features like external commands in the extended information and command CGIs may not be available. The process check command that you specify should follow the same guidelines that are required of the plugins.

Notes:

- The check_netsaint plugin is ideal for the purpose of checking both the status of the NetSaint process and the "freshness" of the data in the status log. I would highly recommend using it in this situation.
- If you are running a chroot'ed web server, you will have to place the plugin (or whatever you're using) in the **sbin/** subdirectory of your NetSaint installation.

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 NetSaint. The CGI will not issue commands to NetSaint 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=netsaintadmin,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=netsaintadmin**

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=netsaintadmin**

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=netsaintadmin,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=netsaintadmin**

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=netsaintadmin,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=netsaintadmin**

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.

Extended Host Information

Format: **hostextinfo[<host_name>]=<notes_url>;<icon_image>;<vrml_image>;<gd2_image>;<alt_tag>**
Example: **hostextinfo[router3]=/hostinfo/router3.html;cat5000.gif;cat5000.jpg;cat5000.gd2;Cisco Catalyst 5000**

Extended host information entries are basically used to make the output from the status, statusmap, statuswrl, and extinfo CGIs look pretty. They have no effect on monitoring and are completely optional.

<host_name>	This is a short name of the host, as defined in the host configuration file.
<notes_url>	This is an optional URL that can be used to provide more information about the host. If you specify an URL, you will see a link that says "Notes About This Host" in the extended information CGI (when you are viewing information about the specified host). Any valid URL can be used. If you plan on using relative paths, the base path will the the same as what is used to access the CGIs (i.e. */cgi-bin/netsaint/*). This can be very useful if you want to make detailed information on the host, emergency contact methods, etc available to other support staff.

<icon_image>	The name of a GIF, PNG, or JPG image that should be associated with this host. This image will be displayed in the status and extended information CGIs. The image will look best if it is 40x40 pixels in size. Images for hosts are assumed to be in the **logos/** subdirectory in your HTML images directory (i.e. */usr/local/netsaint/share/images/logos*).
<vrml_image>	The name of a GIF, PNG, or JPG image that should be associated with this host. This image will be used as the texture map for the specified host in the statuswrl CGI. Unlike the image you use for the *<icon_image>* variable, this one should probably *not* have any transparency. If it does, the host object will look a bit wierd. Images for hosts are assumed to be in the **logos/** subdirectory in your HTML images directory (i.e. */usr/local/netsaint/share/images/logos*).
<gd2_image>	The name of a GD2 format image that should be associated with this host. This image will be used in the image created by the statusmap CGI. GD2 images can be created from PNG images by using the **pngtogd2** utility supplied with Thomas Boutell's gd library. The GD2 images should be created in *uncompressed* format in order to minimize CPU load when the statusmap CGI is generating the network map image. The image will look best if it is 40x40 pixels in size. You can leave these option blank if you are not using the statusmap CGI. Images for hosts are assumed to be in the **logos/** subdirectory in your HTML images directory (i.e. */usr/local/netsaint/share/images/logos*).
<alt_tag>	An optional string that is used in the ALT tag of the image specified by the *<icon_image>* argument. The ALT tag is used in both the status and statusmap CGIs.

Alert Window Suppression

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

This option allows you to specify whether or not you want to permanently suppress the host alert window in the status CGI. Normally the alert window will be displayed if one or more hosts is down or unreachable.

- 0 = Don't suppress alert window, allow it to be displayed (default)
- 1 = Don't display alert window at any time

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 images directory (i.e. */usr/local/netsaint/share/media*).

Using Macros In Commands

Macros

One of the features available in NetSaint is the ability to use macros in command defintions. Immediately prior to the execution of a command, NetSaint 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 seven types of commands that NetSaint recognizes and treats differently. Six types are listed below, along with the macros that can be used with them. The seventh type of command is the ocsp command - any macros which are valid for service event handlers can be used with the ocsp command.

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

The table below lists all macros currently available in NetSaint, 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.

Available Macros

Macro	Description	Service Checks	Service Notifications	Host Checks	Host Notifications	Service Event Handlers & Global Service Event Handler	Host Event Handlers & Global Host Event Handler
$CONTACTNAME$	Short name for the contact (i.e. "jdoe") that is being notified of a host or service problem	No	Yes	No	Yes	No	No
$CONTACTALIAS$	Long name/description for the contact (i.e. "John Doe") being notified	No	Yes	No	Yes	No	No
$CONTACTEMAIL$	Email address of the contact being notified	No	Yes	No	Yes	No	No
$CONTACTPAGER$	Pager number/address of the contact being notified	No	Yes	No	Yes	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
$HOSTALIAS$	Long name/description for the host (i.e. "Big Linux Server")	No	Yes	No	Yes	Yes	Yes
$HOSTADDRESS$	The IP address of the host	Yes	Yes	Yes	Yes	Yes	Yes
$HOSTSTATE$	The current state of the host ("UP", "DOWN", or "UNREACHABLE")	No	Yes	No	Yes	Yes	Yes
$ARGn$	The nth argument passed to the service check command. Read the documentation on service definitions for more info. NetSaint supports up to sixteen argument macros ($ARG1$ through $ARG16$).	Yes	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
$SERVICESTATE$	The status of the service being monitored ("WARNING", "UNKNOWN", "CRITICAL", or "OK")	No	Yes	No	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
$NOTIFICATIONTYPE$	Identifies the type of notification that is being sent ("PROBLEM", "RECOVERY", or "ACKNOWLEDGEMENT").	No	Yes	No	Yes	No	No
$DATETIME$	Date/time stamp	No	Yes	No	Yes	Yes	Yes
$ADMINEMAIL$	Email address for the local administrator (of the host doing the monitoring)	Yes	Yes	Yes	Yes	Yes	Yes
$ADMINPAGER$	Pager number/address for the local administrator	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
$SERVICEATTEMPT$	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
$HOSTATTEMPT$	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

NetSaint Developer Documentation

Version 0.0.6
Last Updated: April 14th, 2000

Note: This documentation is far from complete. For the beta releases, I've just settled with providing information on file formats. More information will be coming as I find the time...

Installing NetSaint

Unpacking The Distribution

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

gunzip netsaint-0.0.6.tar.gz
tar xf netsaint-0.0.6.tar

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

unzip netsaint-0.0.6.zip

When you have finished executing these commands, you should find a **netsaint-0.0.6** directory that has been created in your current directory. Inside that directory you will find all the files that compromise the core NetSaint distribution.

Compiling The Binaries

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

mkdir /usr/local/netsaint

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

./configure --prefix=*prefix* --with-cgiurl=*cgiurl* --with-htmurl=*htmurl* --with-netsaint-user=*someuser* --with-netsaint-grp=*somegroup*

- Replace *prefix* with the actual directory that you created in the step above (default is */usr/local/netsaint*)
- Replace *cgiurl* with the actual url you will be using to access the CGIs (default is */cgi-bin/netsaint*). 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 */netsaint/*)
- Replace *someuser* with the name of a user on your system that will be used for setting permissions on the installed files (default is *netsaint*)
- Replace *somegroup* with the name of a group on your system that will be used for setting permissions on the installed files (default is *netsaint*)

IMPORTANT: The --*prefix* argument of the configure script is very important, as it determines what directory everything gets installed under. If you do not supply this option, the configure script will use */usr/local/netsaint* as the target directory. Make sure that this directory already exists on your system before attempting to install everything.

Compile NetSaint 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

Creating And Installing Sample Configuration Files

You can optionally create sample main, host, and CGI configuration files with the following command:

make config

You can install the sample configuration files with the following command:

make install-config

Installing An Init Script

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

make install-init

...or if you plan on running NetSaint as a daemon, you can install the sample daemon init script to */etc/rc.d/init.d/netsaint* with the following command:

make install-daemoninit

Directory Structure And File Locations

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

cd /usr/local/netsaint

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

Sub-Directory	Contents
bin/	NetSaint core program
etc/	Main and host configuration files (netsaint.cfg and hosts.cfg)
eventhandlers/	Sample scripts that can be used in event handlers. There are also example scripts for implementing redundant monitoring.
sbin/	CGIs programs and config file (nscgi.cfg)
share/	HTML files and images for web interface and documentation
var/	Empty directory for log files

Notes:

1. The default hosts.cfg file created by the configure script will expect that all plugins reside in a **libexec/** subdirectory off of your NetSaint installation. While this directory is not created by the install script distributed with NetSaint, it is created by the install script supplied with the plugins.

Plugins can be obtained from http://www.netsaint.org/download.

Where To Go From Here

Okay, so you're done compiling and installing NetSaint. Now you can move on to configuring NetSaint before starting it up. You'll also probably want to use the web interface, so you'll also have to read the instructions on installing the web interface and configuring web authentication, etc.

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 NetSaint to proactively fix problems before anyone is notified. Another use for event handlers is 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. The diagrams below show the general logic for event handler execution...

Host Event Handler Logic	Service Event Handler Logic

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 NetSaint 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 NetSaint 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 NetSaint user for executing the necessary system commands. I'll leave the details of how to do that up to you...

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_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).

First off, we must define the event handler as a command. Notice the macros that I am passing to the event handler command - these are important!

command[restart-httpd]=/usr/local/netsaint/restart-httpd $SERVICESTATE$ $STATETYPE$ $SERVICEATTEMPT$

Now, let's actually write the event handler script (this is the **/usr/local/netsaint/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 NetSaint 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 NetSaint from continuously executing the script to restart the web server when it is in a "hard" state.

Redundant 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 NetSaint fails or when portions of your network become unreachable.

Note: If you are just learning how to use NetSaint, 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 NetSaint, you need to be familiar with the following...

- Implementing event handlers for hosts and services
- Issuing external commands to NetSaint via shell scripts
- Executing plugins on remote hosts
- Checking the status of the NetSaint process with the check_netsaint plugin

Considerations

There are a few things you need to understand before you jump into implementing redundancy...

First off, 0.0.5 is a first release of NetSaint where redundancy can actually be implemented in any kind of reasonable manner. It just so happened that all the pieces fell into place for accomodating this (event handlers, program modes, and external commands). Additional support for implementing redundancy will be incorporated into future versions of NetSaint, but I need your feedback!

Sample Scripts

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

Scenario 1 - Implementing Redundancy On The Same Network Segment

Introduction

This is the easiest method of implementing redundant monitoring hosts on your network. However, this method only will only protect against a limited number of failures. More complex setups are necessary in order to provide better redundancy across different network segments.

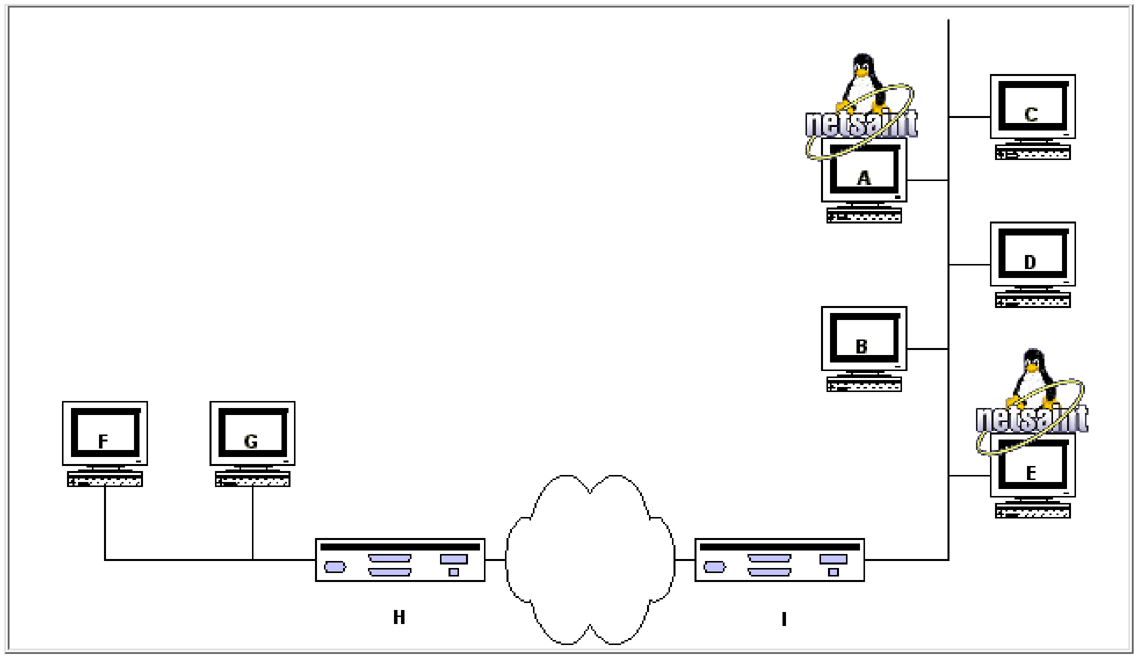
Goals

The goal of this type of redundancy implementation is for a "slave" host running NetSaint to take over the job of monitoring *the entire network* if:

1. The "master" host that runs NetSaint is down or..
2. The NetSaint 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 NetSaint 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 Modes

First off, we need to define what program mode the master and slave hosts will be in when they start monitoring. This is done by using the program_mode option in the main configuration file. The master host (host A) should have its initial program mode set to *active*, while the slave host (host B) should have its initial program mode set to *standby*. That was easy enough...

Initial Configuration

Next we need to consider the differences between the host configuration files 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 NetSaint process on host A. Lets assume that you define this service check to run the check_netsaint plugin on host A. This can be done by using one of the methods described in this FAQ.
- The service definition for the NetSaint 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...

```
command[handle-master-host-event]=/usr/local/netsaint/libexec/eventhandlers/handle-master-host-event $HOSTSTATE$ $STATETYPE$
command[handle-master-proc-event]=/usr/local/netsaint/libexec/eventhandlers/handle-master-proc-event $SERVICESTATE$ $STATETYPE$
```

This assumes that you have placed the event handler scripts in the */usr/local/netsaint/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 enter active mode...
                /usr/local/netsaint/libexec/eventhandlers/enter_active_mode
                ;;
        UP)
                # The master host has recovered!
                # We should go back to being the slave host and
                # let the master host do the monitoring, so
                # enter standby mode...
                /usr/local/netsaint/libexec/eventhandlers/enter_standby_mode
                ;;
        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 NetSaint process is not running!
                # We should now become the master host and
                # take over the responsibility of monitoring
                # the network, so enter active mode...
                /usr/local/netsaint/libexec/eventhandlers/enter_active_mode
                ;;
        WARNING)
        UNKNOWN)
                # The master NetSaint 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...
                ;;
        RECOVERY)
                # The master NetSaint process running again!
                # We should go back to being the slave host,
                # so enter standby mode...
                /usr/local/netsaint/libexec/eventhandlers/enter_standby_mode
                ;;
        esac
        ;;
esac
exit 0
```

What This Does For Us

When things first start out, host A (the master host) is in *active* mode. This means that it monitors all services and sends out notifications if there are problems or recoveries. Host E (the slave host) is in *standby* mode, which means that it will monitor all services but will *not* send out any notifications.

The NetSaint process on host E becomes the master host when...

- Host A goes down (the *handle-master-host-event* host event handler is executed).
- The NetSaint process on host A is not running (the *handle-master-proc-event* service event handler is executed).

When the NetSaint process on host E has entered active mode, 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 monitoring the network!

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

- Host A has recovers (the *handle-master-host-event* host event handler is executed).
- The NetSaint process on host A recovers (the *handle-master-proc-event* service event handler is executed).

When the NetSaint process on host E has entered standby mode, 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 monitoring the network back to host A. Everything is now as it was when we first started!

Time Lags

Redundancy in NetSaint 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 NetSaint checks for external commands

You can minimize this lag by...

- Ensuring that the NetSaint 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_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 NetSaint recovers on the host A, there is also some lag time before host E returns to standby mode. This is affected by the following...

- The time between a recovery of host A and the time the NetSaint process on host E detects the recovery
- The time between the execution of the event handler on host B and the next time the NetSaint 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 switch to *active* mode and take over the responsibilities of monitoring. When host A recovers, host E will switch to *standby* mode. If - when host A recovers - the NetSaint process on host A does not start up properly, there will be a period of time when neither host is monitoring the network! Fortunately, the service check logic in NetSaint accounts for this. The next time the NetSaint process on host E checks the status of the NetSaint process on host A, it will find that it is not running. Host E will then switch back to *active* mode and take over all responsibilities of monitoring.

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 NetSaint process on host A. The rest is up to pure chance, but the total "blackout" time shouldn't be too bad...

Scenario 2 - A Simple Way To Implement Redundancy Across Network Segments

Introduction

If you're monitoring hosts that reside on different network segments, you're going to need a more substantial redundancy model that described in scenario 1. The following example is more complex than that in the first scenario, but the logic behind it should become clear if you study it closely enough.

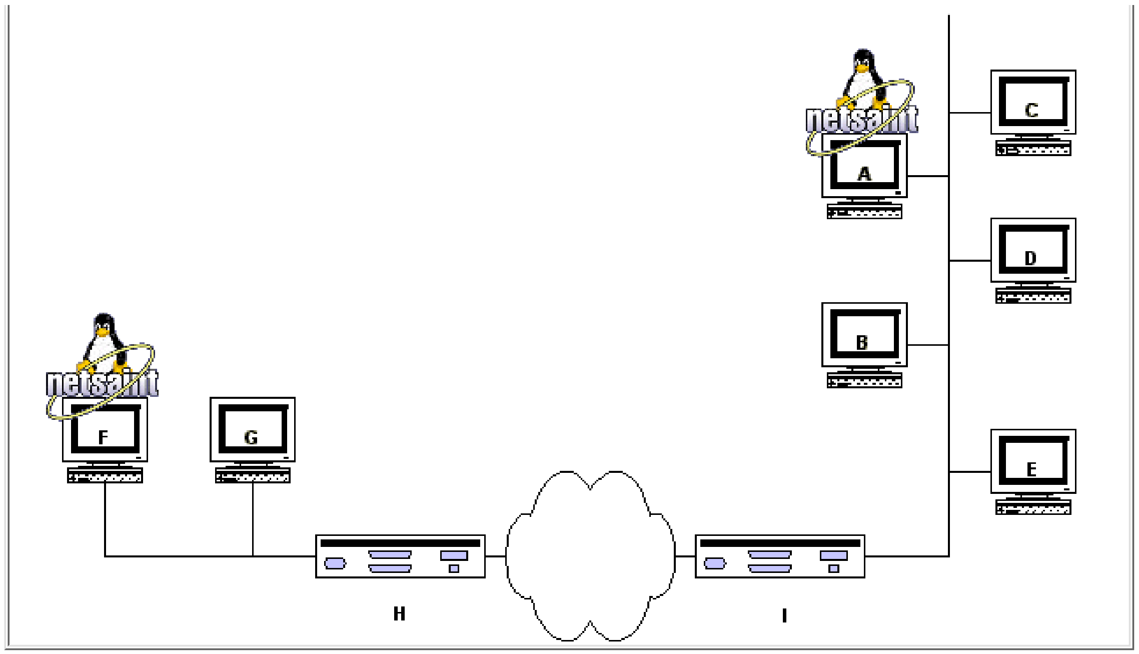
Goals

The goal of this type of redundancy implementation is for a "slave" host running NetSaint to take over the job of monitoring *the entire network* if:

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

Network Layout Diagram

The diagram below shows a relatively simple network setup with host on two network segments. For this scenario I will be assuming that hosts A and F are both running NetSaint and are monitoring all the hosts shown. Host A will be considered the "master" host and host F will be considered the "slave" host. Nodes H and I are routers that lie between the two network segments.



Initial Program Modes

For this example, the master host (host A) should have its initial program mode set to *active*, while the slave host (host F) should have its initial program mode set to *standby*.

Initial Configuration

Next we need to consider the differences between the host configuration files 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 F) 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 F 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 F should have a service defined to check the status of the NetSaint process on host A. Lets assume that you define this service check to run the check_netsaint plugin on host A. This can be done by using one of the methods described in this FAQ.
- The service definition for the NetSaint 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.
- The host definitions for both host H and I should have event handlers defined. Lets say the name of the host event handler in both definitions is handle-router-event

It is important to note that host A (the master host) has no knowledge of host F (the slave host). In this scenario it simply doesn't need to. Of course you may be monitoring services on host F 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...

```
command[handle-master-host-event]=/usr/local/netsaint/libexec/eventhandlers/handle-master-host-event $HOSTSTATE$ $STATETYPE$
command[handle-master-proc-event]=/usr/local/netsaint/libexec/eventhandlers/handle-master-proc-event $SERVICESTATE$ $STATETYPE$
command[handle-router-event]=/usr/local/netsaint/libexec/eventhandlers/handle-router-event $HOSTSTATE$ $STATETYPE$
```

This assumes that you have placed the event handler scripts in the */usr/local/netsaint/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 enter active mode...
                /usr/local/netsaint/libexec/eventhandlers/enter_active_mode
                ;;
        UP)
                # The master host has recovered!
                # We should go back to being the slave host and
                # let the master host do the monitoring, so
                # enter standby mode...
                /usr/local/netsaint/libexec/eventhandlers/enter_standby_mode
                ;;
        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 NetSaint process is not running!
                # We should now become the master host and
                # take over the responsibility of monitoring
                # the network, so enter active mode...
                /usr/local/netsaint/libexec/eventhandlers/enter_active_mode
                ;;

        WARNING)
                ;;
        UNKNOWN)
                ;;
                # The master NetSaint 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...

        RECOVERY)
                # The master NetSaint process running again!
                # We should go back to being the slave host,
                # so enter standby mode...
                /usr/local/netsaint/libexec/eventhandlers/enter_standby_mode
                ;;
        esac
        ;;
esac
exit 0
```

Host Event Handler (handle-router-event)

```
#!/bin/sh

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

What This Does For Us

When things first start out, host A (the master host) is in *active* mode. This means that it monitors all services and sends out notifications if there are problems or recoveries. Host F (the slave host) is in *standby* mode, which means that it will monitor all services but will *not* send out any notifications.

The NetSaint process on host F becomes the master host when...

- Host A goes down (the *handle-master-host-event* host event handler is executed).
- The NetSaint process on host A is not running (the *handle-master-proc-event* service event handler is executed). If either router H or I goes down (the *handle-router-event* host event handler is executed).

When the NetSaint process on host F has entered active mode, it will be able to send out notifications about any service or host problems or recoveries. At this point host F has effectively taken over the responsibility of monitoring the network!

The NetSaint process on host F returns to being the slave host when...

- Host A has recovers (the *handle-master-host-event* host event handler is executed).
- The NetSaint process on host A recovers (the *handle-master-proc-event* service event handler is executed). If either router H or I recovers (the *handle-router-event* host event handler is executed).

When the NetSaint process on host F has entered standby mode, it will not send out notifications about any service or host problems or recoveries. At this point host F has handed over the responsibilities of monitoring the network back to host A. Everything is now as it was when we first started!

Shortcomings

This simple example has some shortcomings that you should be aware of. Note that when one of the routers goes down, the NetSaint process on host F acts as if the NetSaint process on host A is no longer running. This may or may not be the case. If the process on host A *is* running, you'll get potentially bogus notifications being sent out from both NetSaint processes...

As an example, lets say that router H goes down and severs the connection between the two network segments, but everything else is okay. From the view of the NetSaint process on host F, all hosts beyond router H (hosts A, B, C, D, E, and I) are unreachable. At the same time, the NetSaint process on host A (which is on the other side of router H) thinks that all hosts beyond router H (hosts F and G) are unreachable. Both NetSaint processes see that router H is down, but that's the only thing they agree on. This might lead to an enormous amount of bogus notifications being sent out to you. You could potentially get two notifications about router H being down (one from each process) and one notification about every other host on the network being unreachable!

Scenario 3 - A Smarter Way To Implement Redundancy Across Network Segments

Introduction

This is basically just an improvement in the redundancy logic described above in scenario 2. What we will do is make both monitoring hosts aware of each other. In scenario 2, the slave host (host F) knew about the master host (host A), but the master was unaware of the slave. In this scenario both the slave and master hosts will be aware of each other, and will use that information to make better decisions on how to take over or adjust monitoring responsibilities.

Goals

We have several goals with this redundancy scenario...

The "slave" host running NetSaint should take over the job of monitoring *the entire network* if:

1. The NetSaint process on the "master" host stops running for some reason
2. The "master" host that runs NetSaint is down
3. The "master" host becomes unreachable due to one or both of the routers going down and the "master" host was last known to be either down or unreachable

The "slave" host running NetSaint should take over the job of monitoring *only its local network segment* if:

1. The "master" host becomes unreachable due to one or both of the routers going down and the "master" host was last known to be up

The "master" host running NetSaint should *stop* monitoring the entire network and change to monitoring *only its local network segment* if:

1. The "slave" host becomes unreachable due to one or both of the routers going down and the "slave" host was last known to be up

Network Layout Diagram

See network diagram for scenario 2 - its the same...

Initial Program Modes

The master host (host A) should have its initial program mode set to *active*, while the slave host (host F) should have its initial program mode set to *standby*. This is the same setup as described in scenario 2.

Initial Configuration

Scenario 4 - Implementing Multiple Redundancy Methods

If you've got a large, complex network and are paranoid about ensuring that NetSaint monitors everything, you'll probably want to look into implementing multiple redundancy methods. This basically involves combining the redundancy methods described in scenarios 1 and 3 to create a pool of monitoring hosts that are all aware of each other's state and can take over all or part of the network monitoring responsibilites if necessary. If you found the concepts presented in scenario 3 difficult to understand, you should be aware that the complexity of configuration files and event handler scripts will grow exponentially as you add additional monitoring hosts to a multiple redundancy setup.

Since there are endless possibilities for implementing multiple redundancy methods, I won't try to discuss them here. If you decide to implement mixed redundancy methods on your network be prepared to spend a *lot* of time analyzing your network structure, its critical failure points (i.e. routers, firewalls, etc.), the location of monitoring hosts, and what should happen at each monitoring host in the event of a problem. When implementing multiple redundancy methods you cannot simply create event handler scripts based on the state of routers, etc. - you must also take into account the state of other monitoring hosts on the local network segment and (possibly) on other segments.

NetSaint Plugins

Obtaining Plugins

Note: Plugin development for NetSaint has been moved over to SourceForge. The NetSaint plugin development project page (where the latest version of by plugins can always be found) is located at http://netsaintplug.sourceforge.net/.

In addition to the plugins distributed from the SourceForge project, you can also find miscellanous plugins that have been contributed by users in the contrib downloads area of the NetSaint site at http://www.netsaint.org/download/contrib/plugins/

Command Definition Examples For Services

If you're looking for some examples on how to define commands for service or host checks, you can look at the old plugin documentation here. Please note that this documentation is quite old and may contain some errors, as plugin arguments may have changed.

Configuring NetSaint

Configuration Overview

Configuring NetSaint is done by editing three files - the "main" configuration file, the "host" configuration file, and the CGI configuration file.

Main Configuration File

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/netsaint/etc). The default name of the main configuration file is **netsaint.cfg**.

Host Configuration File

Documentation for the host configuration file can be found here. A sample host 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/netsaint/etc). The default name of the host configuration file is **hosts.cfg**. The "host" configuration file is where you define hosts, host groups, contacts, contact groups, commands, time periods, and services.

CGI Configuration File

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/netsaint/etc). The default name of the CGI configuration file is **nscgi.cfg**.

Where To Go From Here

Once you configure NetSaint to your liking you will need to verify the data you entered before starting to monitor anything.

Installing 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.

Configuring Aliases For The HTML Files And CGIs

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

Add a line in the **httpd.conf** file as follows (change to match the directory structure for you installation)...

Alias /netsaint/ /usr/local/netsaint/share/

This will allow you to use an URL like **http://yourmachine/netsaint/** 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 */netsaint/*).

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

ScriptAlias /cgi-bin/netsaint/ /usr/local/netsaint/sbin/

Important: The ScriptAlias entry for the NetSaint CGIs must precede the standard '**ScriptAlias /cgi-bin/ /some...where../**' directive already present in the configuration file. If it doesn't, you will most likely be unable to access the CGIs.

Once you've 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

Once you've gotten the web server restarted, there is just one minor thing you need to verify. Check the CGI configuration file (nscgi.cfg) in the **sbin/** subdirectory of your NetSaint installation and verify that the **main_config_file** variable points to the correct location of the main configuration file on your system. The CGIs will need to know this in order to find your current status log, history log, etc.

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/netsaint** and get the web interface for NetSaint. The CGIs may not display any information, but this will be remedied once you configure web server authentication for the CGIs and start NetSaint.

Where To Go From Here

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.

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 the host configuration file.

Index

Configuring Web Server Authentication

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

```
<Directory /usr/local/netsaint/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 NetSaint, add something similiar to the following in the **access.conf** file as well.

```
<Directory /usr/local/netsaint/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 NetSaint (usually /usr/local/netsaint/sbin and /usr/local/netsaint/share, respectively). The file(s) should have contents similiar to the following...

```
AuthName "NetSaint Access"
AuthType Basic
AuthUserFile /usr/local/netsaint/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/netsaint/etc* directory. It will also create an username/password entry for *netsaintadmin*. You will be asked to provide a password that will be used when *netsaintadmin* authenticates to the web server.

htpasswd -c /usr/local/netsaint/etc/htpasswd.users netsaintadmin

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/netsaint/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 NetSaint 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 NetSaint process information via the extended information CGI
- Issuing NetSaint 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!

Verifying Your NetSaint Configuration

Verifying The Configuration From The Command Line

Once you've entered all the necessary data into the configuration file, its time to do a sanity check. Everyone make mistakes from time to time, so its best to verify what you've entered. NetSaint 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 NetSaint. In order to do this, you must start NetSaint with the **-v** command line argument as follows...

./netsaint -v <main_config_file>

Note that you should be entering the path/filename of your *main* configuration file as the second argument and *not* your host configuration file. NetSaint will read your main configuration file and from there determine where your host configuration file resides (remember the **cfg_file** option in the main config file?).

Relationships Verified During The Pre-Flight Check

During the "pre-flight check", NetSaint verifies that you have defined the data relationships necessary for monitoring. Services, hosts, host groups, contacts, contact groups, and time periods are all related and need to be setup properly in order for things to run. This is a list of the basic things that NetSaint 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, NetSaint 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, NetSaint 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 NetSaint will start monitoring the services you've specified. On to starting NetSaint!

Program Modes

Introduction

The idea of program modes is quite simple, but you need to understand the difference between them before you start doing anything like implementing redundant monitoring hosts. There are two types of program modes - ***Active*** and ***Standby***...

Active Mode

This is the default program mode for NetSaint. While in *active* mode, NetSaint will monitor all services and hosts that you have defined in your host configuration file(s). When a problem arises with one of those services or hosts, NetSaint will send out notifications to all appropriate contacts. This is equivalent to how previous versions of NetSaint worked.

Standby Mode

While in *standby* mode, NetSaint will continue to monitor all services and hosts you defined in your host configuration file(s), but it **will not** send out notifications to any contacts when problems arise. This is particularly useful when implementing redudant monitoring hosts, and is equivalent to temporarily disabling notifications for all defined services and hosts. NetSaint will not send out notifications to any contacts until it returns to *active* mode.

Configuring The Initial Program Mode

By default, NetSaint will enter *active* mode when it (re)starts. If you wish to change the initial program mode to *standy*, you'll have to use the program_mode option in the main configuration file.

Changing Program Modes During Runtime

You can change the current program mode between *active* and *standby* by issuing an external command to NetSaint via the command file. Of course, this assumes that you have enabled external command checks. For more information on external commands, click here.

Two sample shell scripts (*enter_active_mode* and *enter_standy_mode*) are provided in the **sample-scripts/** subdirectory of the NetSaint distribution as examples of how to change the program mode during runtime. You will have to modify the scripts to match the location of your command file before you can use them.

LINUXBOX®

HOME CUSTOMERS FREQUENT QUESTIONS CONTACT US

THE LINUXBOX NETWORK

Free Developer Hosting

Wednesday, June 28, 1999

- System News
- Features
- Signup
- FAQ Database
- Support
- Customers
- Services
- Meet the Staff
- About Us
- Contact Us
- Message Board
- Press Release
- Focalmail
- Home Page

Oops!!

The user, site or other such Web-related resource you're trying to find seems to be missing, outdated or just plain MIA. Perhaps you'll find what you're looking for by going to the linuxbox homepage and searching from there!

Check for the availability of your domain name!

Domain Lookup yournamehere.com go

- Linux Stuff
- Computer
- Search Engines
- News Sources
- Cool Stuff

Free Developer Hosting

| FEATURES | FAQ | SUPPORT | CUSTOMERS | SERVICES | SIGNUP | ABOUT | CONTACT | MESSAGE BOARD |

Frequently Asked Questions (FAQs)

Index

Items in red seem to be the biggest problems for people - read these first..

I'm having trouble compiling Netsaint - What can I do?

If you are running Linux, this is probably because you don't have the **gcc** compiler installed on your system. Either install the compiler yourself or ask your sysadmin to do it for you. If you are running SunOS, IRIX, HP-UX, *BSD, etc. make have to tweak the **Makefile** a bit. This may involve changing the compiler name, compiler options, and/or linker options.

If you're getting errors about the **strncat()**, **strncpy()**, or **snprintf()** functions, you probably don't have the glibc libraries installed on your system. This tends to happen most often on HP-UX and Solaris boxes. I've tried to prevent potential buffer overflows in NetSaint and the CGIs by using these functions, so they are all over the code. If you don't want to install the glibc libraries for some reason, you'll have to find some other way to get everything compiled. If all you're missing is the snprintf() function, try grabbing the snprintf.c file from http://www.ijs.si/software/snprintf/ and adding it to the Makefiles so that it gets included during when you compile things.

If you have to make changes to the Makefile, configure script, or any code in order to compile NetSaint, let me know what OS you are running and what changes you had to make. I would like to include this information in future releases.

I can't find or am having trouble compiling the statusmap CGI...

If you compile all the CGIs, but don't find the statusmap CGI, you probably don't have Thomas Boutell's gd library installed correctly on your system. The **gd** library (and thus the statusmap CGI) also requires that you also have the **zlib** and **png** libraries installed. Version 1.6.3 or higher of the gd library is required, as the CGI generates a PNG image of your network layout.

If you find that the statusmap CGI has not been compiled, make sure you have the gd library installed on your system and rerun the configure script with the following options:

```
./configure --with-gd-lib=LIBDIR --with-gd-inc=INCDIR
```

Replace LIBDIR with the directory in which the gd library is installed (usually /usr/lib or /usr/local/lib) and replace INCDIR with the directory in which the header files for the gd library are installed (usually /usr/include or /usr/local/include).

After you rerun the configure script, make sure to recompile the CGIs and install them in their proper location.

"NetSaint process may not be running" warnings in the CGIs

If you are getting erroneous messages about the NetSaint process not running while viewing the CGIs, its probably due to one of the following items:

1. You haven't defined a command to check the status of the NetSaint process. This is done by supplying a value for the process_check_command directive in the CGI configuration file.

2. If you have defined a command, perhaps it is not returning the proper exit code. The command must follow the same rules as the plugin: a return code of 0 indicates that NetSaint is running, values of 1, 2, or -1 indicate that NetSaint is either not running or in some degraded state.

3. If you have defined a process check command that uses the check_netsaint plugin, make sure that the plugin is functioning as it should. Execute the check_netsaint plugin from the command line and check the result. If the plugin is reporting that the NetSaint process cannot be found or if it returns a "Could not open pipe" message, you may need to edit the PS_RAW_COMMAND definition in the common/config.h file of the plugin distribution to match the syntax for the **ps** command on your system. For example, under FreeBSD you should use either *"/bin/ps -ao 'state user ppid args'"* or *"/bin/ps -axo 'state user ppid command'"* (it seems to vary). Once you've changed the PS_RAW_COMMAND definition, recompile the plugins and test the newly compiled check_netsaint plugin to see if it works.

The CGIs will not allow you to sumbit any commands while they think the NetSaint process is not running. This is done primarily to prevent people from accidentally submitting multiple shutdown/restart commands that don't get processed until NetSaint is started at some future time.

Hosts are incorrectly listed as being DOWN and/or services have a status of "HOST DOWN"

This seems to be one of the biggest issues for new users. 99.9% of the time this problem is due to an incorrect command definition for the host check command you specified in the host definition.

A major cause for this problem was due to a syntax change to the command line arguments of the check_ping plugin. You need to make sure that the host check command is using the proper syntax for the version of the check_ping plugin that you have. You can check to see if the command works properly by executing it manually from the command line. Recent versions of the check_ping plugin require that a **-p** flag be used to specify the number of packets to send. Previous versions of the plugin did not require this flag - that's where the problem lies. Check your host check command definition(s) to make sure they are using the proper syntax. Example:

```
command[check-host-alive]=/usr/local/netsaint/libexec/check_ping $HOSTADDRESS$ 100 100 1000.0 1000.0 -p 1
```

Important! Just because you have a service that is monitoring ping statistics for a host does *not* mean that the actual host status is being checked. The status of a host is only checked when a service check results in a non-OK state or if the host was previously down and a service check results in an OK state.

Some symptoms of incorrect host check commands include:

1. Hosts incorrectly being listed as DOWN

2. Services that have a status of "HOST DOWN", even though the host they reside on is actually UP
3. Alternating alerts/notifications about host problems and recoveries

When hosts go down, I get notification about services instead of hosts and the service notifications contain incorrect data

Several people have reported this problem and I spent hours trying to find the problem until I realized it wasn't a bug in the code. If you get service notifications when you should be getting host notifications (and the service notifications you get seem to contain bogus data), check your contact definitions in the host config file. They are most likely incorrect.

Make sure that you are not using the same notification command for service and host notification commands. Service and host notifications are very different and make use of macros which are not transferrable between each type. Look at the sample host config file provided with NetSaint to see what the contact definitions look like and how the service and host notification commands differ. If you're wondering what macros can be used in either type of notification, look at this table.

Debugging "unknown variable" errors during configuration file verification or runtime

When trying to run NetSaint or verify your configuration file data using the **-v** argument, NetSaint may print out a message like "Error in configuration file 'xxxxxxx.cfg' - Line 34 (Unknown variable)". A few simple checks will usually resolve this problem...

1. Make sure you are passing the path to the main configuration file and *not* the host configuration file on the command line. Many people have made this mistake. The correct syntax would be as follows (modified for your system, of course):
   ./netsaint -v /usr/local/netsaint/etc/netsaint.cfg

2. Make sure that you don't have any invalid variables defined in your configuration file. Notice that the error message will contain a reference to the name of the configuration file and the line number on which the error was encountered. Make sure that all comment lines contain a pound sign (#) in the first character of the line. If you're not sure about what variables are valid, check the documentation for the main and host configuration files.

3. Make sure all variable identifiers are in lower case. Example:
   "admin_email=someaddress@somedomain.com" instead of
   "ADMIN_EMAIL=somedomain@nowhere.com"

How do I run multiple instances on NetSaint on the same machine?

You can run multiple instances of NetSaint on the same machine, if you ensure that the following variables are unique for each instance of NetSaint...

- Main configuration file
- Temp file
- Status file
- External command file
- Log file
- Log archive path
- Lock file

If you are using the web interface, you will have to setup separate directories to hold the CGIs for each instance of NetSaint and create appropriate script aliases in your web server configuration file. This is necessary because CGI configuration file must be unique for each setup of CGIs, as it contains a reference to which main configuration file the CGIs should read.

One last thing you should check is your init script (if you're using one). The init script should start, stop, restart, and reload all copies of NetSaint (if that's what you want it to do).

How do I change the contents of the default web page?

Several people have asked how to modify the default web page so that service detail or service overview information is displayed in the right hand frame (instead of the intro page). You can do this rather easily by modifying the frameset information in the **index.html** page (located in the root web directory for NetSaint) as follows..

Default Frame Configuration

```
<FRAMESET BORDER="0" FRAMEBORDER="0" FRAMESPACING="0" COLS="180,*">
<FRAME SRC="side.html" NAME="side" TARGET="main">
<FRAME SRC="main.html" NAME="main">
</FRAMESET>
```

Modified Configuration

```
<FRAMESET BORDER="0" FRAMEBORDER="0" FRAMESPACING="0" COLS="180,*">
<FRAME SRC="side.html" NAME="side" TARGET="main">
<FRAME SRC="xxxxx" NAME="main">
</FRAMESET>
```

Replace xxxxx with one of the following values, or anything else you may want...

Option	Description
/cgi-bin/netsaint/status.cgi?host=all	This will display service status details for all hosts in the right hand side of the frame

/cgi-bin/netsaint/status.cgi?hostgroup=all	This will display a service status overview for all hostgroups in the right hand side of the frame
/cgi-bin/netsaint/showlog.cgi	This will display the contents of the log file in the right hand side of the frame
/cgi-bin/netsaint/history.cgi?host=all	This will display the service history for all hosts in the right hand side of the frame

Read the documentation on the CGIs for more information on what options each supports.

When I access the CGIs I don't see everything I should or I get authorzation errors...

If you believe you are unable to see all the information in the CGIs or if you are getting authorization errors, you probably haven't configured the web server to require authentication or haven't setup authorzation correctly. See the documentation on authentication and authorization in the CGIs here.

Where can I find the traceroute CGI?

Newer versions of the check_ping plugin are capable of producing HTML that provides a link to a traceroute CGI written by Ian Cass. The traceroute CGI is not included in the core distribution of NetSaint. However, you can find it in the contrib area of the downloads section at http://www.netsaint.org/download/contrib.

How do I requre users to authenticate before accessing the web interface?

See the documentation on authentication and authorization in the CGIs here.

How do I get those pretty pretty host icons to display in my CGIs?

If you want to associate images with particular hosts for use in the status, status map, status world, and extended information CGIs, you must define extended host information entries in your CGI configuration file.

I'm getting errors when attempting to commit commands to NetSaint via the command CGI

If you are getting '**Could not open command file *somefile* for update**' errors when attempting to commit commands to NetSaint via the command CGI, the most likely problem is with directory and/or file permissions. Here is what you can do to fix it. Note: You must be *root* in order to do some of these steps...

First, find the user that your web server process is running as. On many systems this is the user *nobody*, although it will vary depending on what OS/distribution you are running.

Next, create a new group that will be granted permissions to update the NetSaint command file. Let's say you want to call the group '**nscmd**'. On RedHat Linux you can use the following command to add a new group (other systems may differ):

```
/usr/sbin/groupadd nscmd
```

Next, add all users who should have access to the command file to the group you just created. In this example we'll just add the user *nobody*...

```
/usr/sbin/usermod -G nscmd nobody
```

Next, create the directory where the command file should be stored. By default, this is */usr/local/netsaint/var/rw*, although it can be changed by modifying the command_file variable.

```
mkdir /usr/local/netsaint/var/rw
```

Next, change the group ownership of the directory used to hold the command file...

```
chown -R .nscmd /usr/local/netsaint/var/rw
```

Also check the group permissions on the directory. The group you created needs to have write access there. The last thing you'll have to do is restart your web server with a command similiar to the following..

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

Apparently Apache needs to be restarted in order to inherit the new group permissions you assigned. That's it. You should be able to commit commands to NetSaint via the CGI now (assuming you have the proper authorization).

If you supplied the **--with-command-grp=*somegroup*** option when running the configure script, you can create the directory to hold the command file and set the proper permissions by running '**make install-commandmode**'.

How do I monitor virtual web servers that use host headers?

If you are running a web server with multiple virtual servers and only one IP address, this applies to you. Let's say that your web server has an IP address of 192.168.0.1 and two virtual servers running on it - "www.myfirstdomain.com" and "www.myseconddomain.com". Both of these domain names resolve to the same IP address (192.168.0.1) during a DNS lookup. The check_http plugin can handle this type of situation without a problem. You will need to specify the virtual web site name as an additional command line argument to the plugin (using the -hn option). Example:

```
command[check_http2]=/usr/local/netsaint/check_http $HOSTADDRESS$ -u / -p 80 -hn $ARG1$

service[myhost]=First Virtual Web Server;3;2;120;1;1;1;check_http2!www.myfirstdomain.com
service[myhost]=Second Virtual Web Server;3;2;120;1;1;1;check_http2!www.myseconddomain.com
```

The **check_http2** command defined here will use the check_http plugin to open a connection to port 80 of the host at IP address 192.168.0.1. It will then send an HTTP/1.1 request for the root document, along with either a "Host: www.myfirstdomain.com" or "Host: www.myseconddomain.com" in the request header.

How do I monitor remote host information?

Several people have asked how to use various plugins that check information on the local host to report information from remote hosts. Various methods for doing this are described below..

If you need to actually execute a plugin on a remote host and get the results back, you can use one of the following methods...

- Use the check_by_ssh "plugin" to execute a plugin on a remote host. The **check_by_ssh** plugin is basically a wrapper for executing a plugin on a remote host using SSH. You must have SSH installed and configured properly in order to use this.
- Use the nrpe addon to accomplish this. The plugins and the nrpe daemon reside on the remote host. The check_nrpe plugin (included with the nrpe package) sends a request to the nrpe daemon to execute the plugin on the remote host and then grabs the results for NetSaint.
- Use the nrpep addon. This addon works in a similiar manner to the nrpe package, but it encrypts the transmitted data, runs as a service from inetd, and makes use of the TCP Wrappers package for access control.
- Use **rsh** to execute the plugin remotely, although I guess I wouldn't recommend this..

If all you need is to check disk space, etc. on a remote host, you can use one of the methods below...

- Use one of the plugins included with the netsaint_statd addon for NetSaint. The addon, written by Charlie Cook, includes a Perl daemon which runs on the remote host and four plugins which are used to gather the remote host information from the daemon. The daemon is designed to run on Linux, IRIX, HP-UX, SunOS, and OSF/1 systems. Modifying the code for other systems should be fairly easy. More on the **netsaint_statd** plugin can be found here.
- Use the check_overcr plugin to query information from a remote host. The remote host must be running Eric Molitor's Over-CR collector in order for this to work.

- Use the check_snmp plugin to check the value of various OIDs on the remote host. You must have SNMP services installed and running on the remote host in order to do this.

How can I monitor NT servers?

The good news is that NT has a lot of performance data that you can monitor. The bad news is that its difficult to do. Your best bet is probably going to be to install SNMP services on all your NT boxes. Ian Cass has written a FAQ on how to do this at http://elton.dev.knowledge.com/snmpfaq.html

In order to expose NT performance counters for monitoring, you'll have to run the SNMP service on all servers you want to monitor. You'll also have to install any necessary performance MIBs for the services you want to monitor. I believe these can be found in the NT Resource Kit or in various server admin packages. If you've feeling extra lucky you can try to search the Microsoft site for the terms **SNMP** and **MIB** and maybe you'll find something...

You can search the MRTG mailing list archives for more information on configuring NT servers to expose various performance counters via SNMP. I know this has been discussed in the past, as many people are graphing various NT performance statistics using MRTG. In fact, somebody from Microsoft is actually doing it - you can find their web page at http://snmpboy.rte.microsoft.com/.

Once you've actually got the SNMP stuff working, you can use the check_snmp plugin to query your NT servers and generate alarms.

A few people are looking into the possibilities of creating a service that runs under NT to facilitate easier remote monitoring. Once these efforts solidify, an announcement will be made on the NetSaint mailing lists.

How do I monitor printers?

Assuming you have HP printers with JetDirect© cards installed, you can use the HP printer plugin to monitor them. Before you begin monitoring printers you should carefully plan your configuration to match level of monitoring and response time you need. You need to balance this against the annoyance of getting alerted every time sometime takes the printer offline to manually feed a transparency, etc. A lot of admins probably don't care if the printer is jammed or is out of paper, but some tech support people in large corporations might find this to be a useful feature. Anyway, if you decide to do this you will need to do the following things:

- Enable the TCP/IP protocol stack on the JetDirect© card and assign it an IP address. External JetDirect© devices with multiple parallel ports will need this to be done on each port that has a printer connected that you want to monitor.
- Create a host definition entry for the printer in your config file. Set the **notify_recovery**, **notify_down**, and **notify_unreachable** options to **0** if you don't want NetSaint to send you alert when the printer gets turned off on and on.
- Create a host group for the printer(s) you defined. Call it **printers** or something similiar.
- Create a contact group containing all contacts that should be notified about printer problems. This

group should be the notification group you specified in the **printers** host group you just defined.

- Create a service to be monitored for the printer. Set the **notify_critical** option to **0** if you don't want to get notified when someone turns the printer off. The check_hpjd plugin returns a warning status whenever a problem is detected with a printer, so make sure the **notify_warning** option is set to **1** (assuming you want to the contact be notified). Also, fill in the **contactgroups** option with the name of the contact group you created for printers.

Can NetSaint send SNMP traps to management hosts?

Yes, but not directly. NetSaint relies on plugins to handle the gathering of service and host information and event handler scripts to handle events that occur with services and hosts. If you want to have NetSaint send an SNMP trap to a management host in the event that a particular service has a problem, you will have to write a service event handler script and add it to the **event_handler** option of the service definition. If you have the UCD-SNMP package installed on your host, you could have the script call the **snmptrap** command to actually send a trap message, depending on what type of service event occurred. Look at the example event handler script to get a better idea of how to write a script.

Can NetSaint log host and service events to an external database?

Not directly, but this can be done fairly easily. You'll probably want to define global host and service event handlers to do this. The global event handlers could call a script which inserts the appropriate event information into a database of your choosing. This would allow you to run queries and generate more detailed reports than what are available in the CGIs.

Something isn't working properly - How can I track down the problem?

I've worked in tech support for a few years and have spent my share of time on a helpdesk. Most people are vague when they report a problem and have no desire whatsoever to try and track down the problem - they just want you to fix it **now**. I hope you are not that type of person. NetSaint is relatively new and is probably chock full of bugs, so things will not always work properly. If you suspect that either the service check or notification routines are not working, here are a few things you can do to try and track down the problem...

This first thing you should do is verify your configuration data by running NetSaint with the **-v** option. Example:

```
./netsaint -v /usr/local/netsaint/etc/netsaint.cfg
```

If no errors are found, proceed to the next steps. If NetSaint reports some error, go back and fix your configuration files.

The next step will take more time, but will give you more information on what is going on inside of NetSaint. When I first developed NetSaint I added a lot of debugging code to help me track down problems. I still use that code when I add new features or track down bugs myself. Here is how to use the debugging code...

Reconfigure NetSaint and enable one or more debug options as follows, replacing the "**--**enable-DEBUGx" with one or more of the values from the table below:

```
./configure --prefix=/your/netsaint/directory --enable-DEBUGx
```

Debugging Options

Debug Option	Description
--enable-DEBUG0	Used to trace function calls. A **lot** of messages will be printed out if you uncomment this option, but it very useful to trace what functions are being called. Note that not all functions will print an exit message if code within the function causes an early exit (before reaching the end of the function).
--enable-DEBUG1	Used to print out informational messages about variable settings. Most useful when trying to debug the configuration data as it is being read or verified.
--enable-DEBUG2	Used to print out warning messages, usually when configuration data is being read or verified.
--enable-DEBUG3	Used to print out informational messages during host and service checks. Good to use if you suspect problems are occuring during service checks.
--enable-DEBUG4	Used to print out informational messages during host and service notifications. Good to use if you suspect problems are occurring during the notification events.

Recompile NetSaint.

Verify your configuration data again **-** you'll see a lot more information this time if you have enabled the **DEBUG1** option. Try redirecting output to a file so that you can view or print it at a later time.

If you have defined either the **DEBUG3** or **DEBUG4** options, run NetSaint as a foreground process and start monitoring your services. Example:

```
./netsaint /usr/local/netsaint/etc/netsaint.cfg
```

Kill NetSaint at an appropate point (i.e. after a service check fails) and look through the output. It should help you track down where the problem is occurring. Some code tweaking may be necessary on your part in order to fix things. Let me know if you have to make any such alterations so I can include the fix in future releases.

If you are unable to determine or fix the problem on your own, email me the following items:

1. The version of NetSaint you are running
2. A description of what is going wrong and what you suspect is the problem
3. The OS you're running NetSaint on
4. Your configuration files (**netsaint.cfg** and **hosts.cfg**)
5. Output from the program run (with debugging options on)

NetSaint Status Levels

Different status levels (also referred to as "states") for hosts and services are listed below. Some states are internal to NetSaint and cannot be generated by external plugins. Plugins are only capable of returning OK, UNKNOWN, WARNING, and CRITICAL states. See the documentation on writing plugins for more information

Service Status Levels

Status	Description
PENDING	This status level indicates that the service has not been checked yet. Pending status levels occur only after NetSaint is started and will disappear as services are checked.
OK	This status indicates that the service being monitored appears to be both running and functioning properly.
RECOVERED	This status indicates that the service is functioning properly at the moment, but that at the last check it was at either a warning, an unknown, or a critical status. In other words, it just came back up.
WARNING	This status indicates that the service being monitored appears to have some problems, but is still in a semi-functional state.
UNKNOWN	This status indicates that there was some sort of internal error with the plugin that prevented it from checking the status of a service. For the purposes of notification, unknown status levels are considered to be the same as warning status levels.
CRITICAL	This status indicates that either there is a big problem with the service being checked or that the service is completely unavailable.
UNREACHABLE	This status indicates that the service cannot be checked because the host that it is associated with is unreachable.
HOST DOWN	In the status CGI this indicates that the host associated with the service was down the last time the service was checked.

Host Status Levels

Status	Description
PENDING	This status level indicates that the status of the host is unknown, because no services associated with it have been checked yet. Pending status levels occur only when NetSaint is started, and will disappear as soon as at least one service associated with the host is checked.
UP	This status level indicates that the host appears to be up.
DOWN	This status level indicates that the host is down.
UNREACHABLE	This status level indicates that the host is unreachable because a host that it relied on (i.e. a parent or grandparent host) was down.

Service Check Parallelization

Introduction

Beginning with release 0.0.5, the ability to execute service checks in parallel was built into NetSaint. This documentation will attempt to explain in detail what that means and how it affects services that you have defined.

Changes In Service Check Logic

In order to facilitate parallelized service checks, the service check logic has been changed from that of version 0.0.4 and earlier. These earlier versions of NetSaint executed one service check at a time and processed the results from the check before moving onto the next service.

Beginning with version 0.0.5, the service check logic has been broken up into two distinct parts - *execution of service checks* and *processing of service check results* (also called service "reaper" events).

How The Parallelization Works

Before I can explain how the service check parallelization works, you first have to understand a bit about how NetSaint schedules events. All internal events in NetSaint (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. NetSaint does its best to ensure that all events get executed when they should, although events may fall behing schedule if NetSaint is busy doing other things.

Service checks are one type of event that get scheduled in NetSaint's event queue. When it comes time for a service check to be executed, NetSaint will kick off another process to go out and run the service check (i.e. a plugin of some sort). NetSaint does *not*, however, wait for the service check to finish! Instead, NetSaint 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 NetSaint to run the service check sends a message back to NetSaint containing the results of the service check. It is then up to NetSaint to check for and process the results of that service check when it gets a chance.

In order for NetSaint 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 NetSaint's 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 NetSaint determines whether or not hosts should be checked, notifications should be sent out, etc. When the service check results have been processed, NetSaint 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, NetSaint uses message queues to handle communication between NetSaint 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_attempts argument in each service definition to something greater than one. If the service check does happen to collide with another running check, NetSaint will retry the service check *max_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 NetSaint. 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.

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:

- *check_interval*
- *retry_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 NetSaint first starts or restarts...

Initial Scheduling

When NetSaint (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 NetSaint 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 NetSaint 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 NetSaint is running. NetSaint does *not* start monitoring anything when this argument is used.

Inter-Check Delay

As mentioned before, NetSaint attempts to equalize the load placed on the machine that is running NetSaint 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.

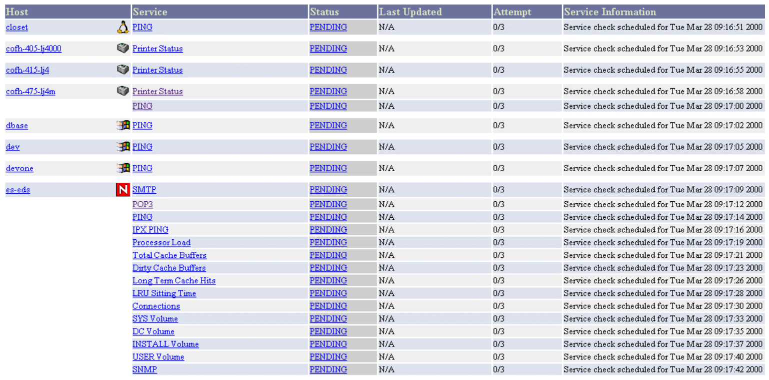
When using the "smart" setting of the *inter_check_delay_method* variable, NetSaint will calculate an inter-check delay value by using the following calculation:

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 NetSaint 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 NetSaint remains somewhat balanced.

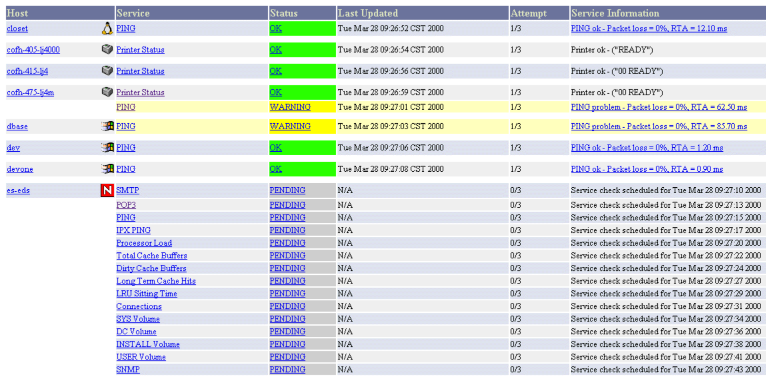
The following two images show some output from the status CGI after NetSaint has been started and demonstrate how the inter-check delay works. For these examples, the inter-check delay was approximately 2.3 seconds (there were a total of 113 services with an average check interval of about 4.3 minutes). The first image shows the inital scheduling of service checks and the second image shows how NetSaint executes service checks (the interleave_factor option was set to 1 for this example, so checks are not interleaved). Click on either image for a larger version.

Image 1. Initial scheduling of service checks (non-interleaved)



Host	Service	Status	Last Updated	Attempt	Service Information
closet	PING	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:16:51 2000
cofh-405-lj4000	Printer Status	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:16:53 2000
cofh-415-lj4	Printer Status	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:16:55 2000
cofh-475-lj4m	Printer Status	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:16:58 2000
	PING	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:17:00 2000
dbase	PING	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:17:02 2000
dev	PING	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:17:05 2000
devone	PING	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:17:07 2000
es-eds	SMTP	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:17:09 2000
	POP3	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:17:12 2000
	PING	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:17:14 2000
	IPX PING	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:17:16 2000
	Processor Load	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:17:19 2000
	Total Cache Buffers	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:17:21 2000
	Dirty Cache Buffers	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:17:23 2000
	Long Term Cache Hits	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:17:26 2000
	LRU Sitting Time	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:17:28 2000
	Connections	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:17:30 2000
	SYS Volume	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:17:33 2000
	DC Volume	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:17:35 2000
	INSTALL Volume	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:17:37 2000
	USER Volume	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:17:40 2000
	SNMP	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:17:42 2000

Image 2. Non-interleaved execution of checks



Host	Service	Status	Last Updated	Attempt	Service Information
closet	PING	OK	Tue Mar 28 09:26:52 CST 2000	1/3	PING ok - Packet loss = 0%, RTA = 12.10 ms
cofh-405-lj4000	Printer Status	OK	Tue Mar 28 09:26:54 CST 2000	1/3	Printer ok - ("READY")
cofh-415-lj4	Printer Status	OK	Tue Mar 28 09:26:56 CST 2000	1/3	Printer ok - ("00 READY")
cofh-475-lj4m	Printer Status	OK	Tue Mar 28 09:26:59 CST 2000	1/3	Printer ok - ("00 READY")
	PING	WARNING	Tue Mar 28 09:27:01 CST 2000	1/3	PING problem - Packet loss = 0%, RTA = 62.50 ms
dbase	PING	WARNING	Tue Mar 28 09:27:03 CST 2000	1/3	PING problem - Packet loss = 0%, RTA = 85.70 ms
dev	PING	OK	Tue Mar 28 09:27:06 CST 2000	1/3	PING ok - Packet loss = 0%, RTA = 1.20 ms
devone	PING	OK	Tue Mar 28 09:27:08 CST 2000	1/3	PING ok - Packet loss = 0%, RTA = 0.90 ms
es-eds	SMTP	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:27:10 2000
	POP3	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:27:13 2000
	PING	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:27:15 2000
	IPX PING	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:27:17 2000
	Processor Load	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:27:20 2000
	Total Cache Buffers	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:27:22 2000
	Dirty Cache Buffers	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:27:24 2000
	Long Term Cache Hits	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:27:27 2000
	LRU Sitting Time	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:27:29 2000
	Connections	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:27:31 2000
	SYS Volume	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:27:34 2000
	DC Volume	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:27:36 2000
	INSTALL Volume	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:27:38 2000
	USER Volume	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:27:41 2000
	SNMP	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:27:43 2000

Service Interleaving

As discussed above, the inter-check delay helps to equalize the load that NetSaint 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, NetSaint 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 NetSaint 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, NetSaint will calculate an interleave factor by using the following calculation:

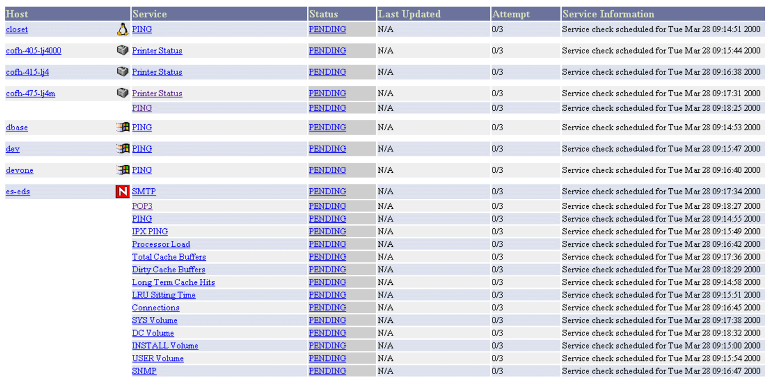
interleave factor = ceil (total number of services / total number of hosts)

Let's take an example. Say you have a total of 1,000 services and 150 hosts that you monitor. NetSaint would calculate the interleave factor to be 7. This means that when NetSaint 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 following two images show some output from the status CGI after NetSaint has been started and demonstrate how the interleaving works. For these examples, the inter-check delay was approximately 2.3 seconds and the interleave factor was 5 (there were a total of 113 services and 28 hosts). The first image shows the inital scheduling of service checks with interleaving and the second image shows how

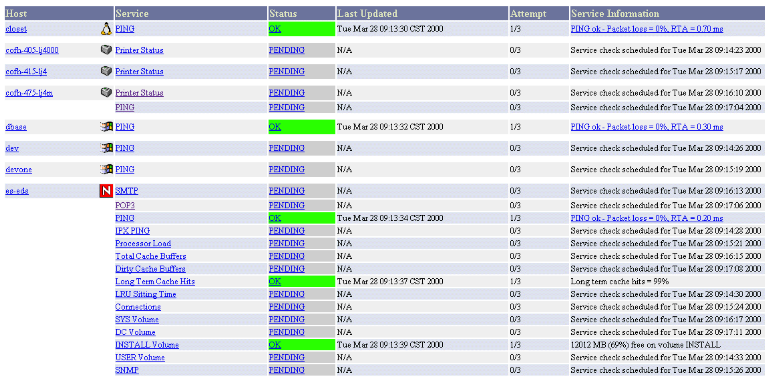
NetSaint executes service checks. Notice the differences between these two images and images 1 and 2 above. Click on either image for a larger version.

Image 3. Initial scheduling of service checks (interleaved)



Host	Service	Status	Last Updated	Attempt	Service Information
closet	PING	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:14:51 2000
cofh-405-lj4000	Printer Status	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:15:44 2000
cofh-415-lj4	Printer Status	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:16:38 2000
cofh-475-lj4m	Printer Status	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:17:31 2000
	PING	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:18:25 2000
dbase	PING	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:14:53 2000
dev	PING	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:15:47 2000
devone	PING	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:16:40 2000
es-eds	SMTP	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:17:34 2000
	POP3	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:18:27 2000
	PING	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:14:55 2000
	IPX PING	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:15:49 2000
	Processor Load	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:16:42 2000
	Total Cache Buffers	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:17:36 2000
	Dirty Cache Buffers	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:18:29 2000
	Long Term Cache Hits	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:14:58 2000
	LRU Sitting Time	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:15:51 2000
	Connections	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:16:45 2000
	SYS Volume	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:17:38 2000
	DC Volume	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:18:32 2000
	INSTALL Volume	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:15:00 2000
	USER Volume	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:15:54 2000
	SNMP	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:16:47 2000

Image 4. Interleaved execution of checks



Host	Service	Status	Last Updated	Attempt	Service Information
closet	PING	OK	Tue Mar 28 09:13:30 CST 2000	1/3	PING ok - Packet loss = 0%, RTA = 0.70 ms
cofh-405-lj4000	Printer Status	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:14:23 2000
cofh-415-lj4	Printer Status	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:15:17 2000
cofh-475-lj4m	Printer Status	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:16:10 2000
	PING	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:17:04 2000
dbase	PING	OK	Tue Mar 28 09:13:32 CST 2000	1/3	PING ok - Packet loss = 0%, RTA = 0.30 ms
dev	PING	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:14:26 2000
devone	PING	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:15:19 2000
es-eds	SMTP	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:16:13 2000
	POP3	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:17:06 2000
	PING	OK	Tue Mar 28 09:13:34 CST 2000	1/3	PING ok - Packet loss = 0%, RTA = 0.20 ms
	IPX PING	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:14:28 2000
	Processor Load	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:15:21 2000
	Total Cache Buffers	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:16:15 2000
	Dirty Cache Buffers	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:17:08 2000
	Long Term Cache Hits	OK	Tue Mar 28 09:13:37 CST 2000	1/3	Long term cache hits = 99%
	LRU Sitting Time	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:14:30 2000
	Connections	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:15:24 2000
	SYS Volume	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:16:17 2000
	DC Volume	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:17:11 2000
	INSTALL Volume	OK	Tue Mar 28 09:13:39 CST 2000	1/3	12012 MB (69%) free on volume INSTALL
	USER Volume	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:14:33 2000
	SNMP	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:15:26 2000

Maximum Concurrent Service Checks

In order to prevent NetSaint 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 NetSaint's 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, NetSaint 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, NetSaint 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 NetSaint 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 NetSaint 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) NetSaint 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 NetSaint 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 NetSaint 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, NetSaint 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, NetSaint 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

NetSaint will execute the check! When it comes time to actually *execute* a service check, NetSaint will verify that the check can be run at the current time. If it cannot, NetSaint 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, NetSaint 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 NetSaint rechecks the service *max_attempts* times and it is still in a non-OK state, NetSaint 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, NetSaint 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), NetSaint 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 NetSaint sees a need. This is a common question asked by users, so it needs to be clarified.

One instance where NetSaint checks the status of a host is when a service check results in a non-OK status. NetSaint 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, NetSaint 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 NetSaint 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 NetSaint can take any further action on the service(s) that are having problems.

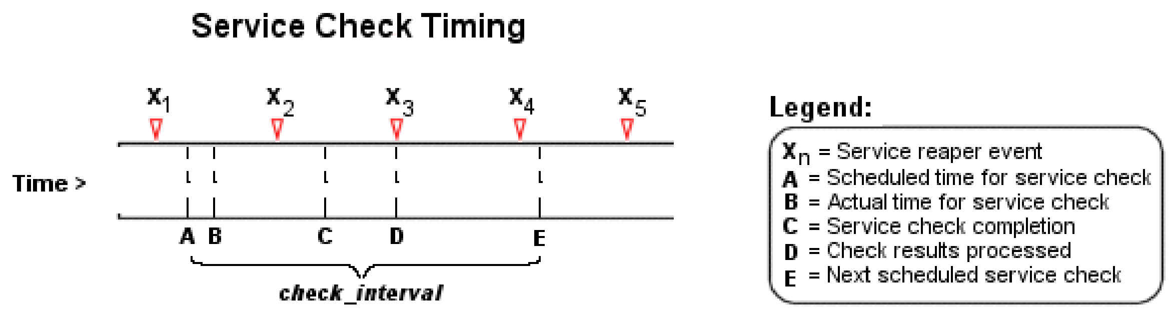
Scheduling Delays

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 NetSaint, 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
X_1
X_2
X_3
X_4
X_5
Time >
A
B
C
D
E
check_interval
Legend:
X_n = 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 NetSaint, 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, NetSaint 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 NetSaint's 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**), NetSaint will compensate by adjusting the next check time. This is done to ensure that NetSaint 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...?

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 something very high (like 24 hours).

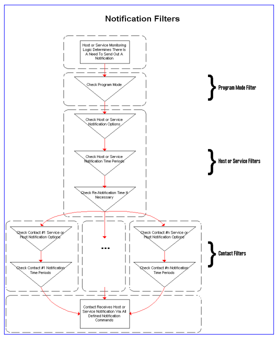
Who Gets Notified?

Each service definition has a <*contactgroups*> option that specifies what contact groups receive notifications for that particular service. Each contact group can contain one or more individual contacts. When NetSaint 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. NetSaint 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 NetSaint sends out a host notification, it will notify contacts that are members of all the contact groups that should be notified for any and all host groups that the host is a member of. NetSaint 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...


Notification Filters
Host or Service Monitoring Logic Determines There Is A Need To Send Out A Notification
Check Program Mode
Program Mode Filter
Check Host or Service Notification Options
Check Host or Service Notification Time Periods
Check Re-Notification Time If Necessary
Host or Service Filters
Check Contact #1 Service or Host Notification Options
Check Contact #1 Notification Time Periods
...
Check Contact #n Service or Host Notification Options
Check Contact #n Notification Time Periods
Contact Filters
Contact Receives Host or Service Notification Via All Defined Notification Commands

Click on the image to the left for a graphical representation of the filters that must be passed before notifications are sent to contacts.

Program Mode Filter:

The first filter that notifications must pass is the program mode test. If NetSaint is in *STANDBY* mode, **no one gets contacted**. If NetSaint is in *ACTIVE* mode, the notification gets passed to the next filter...

Service and Host Filters:

The first set of host or service filters that must be passed are the 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 second set of host or service filters 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, NetSaint 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 NetSaint 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 NetSaint 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 NetSaint?

I've gotten several questions about why notification methods (paging, etc.) are not directly incorporated into the NetSaint code. The answer is simple - it just doesn't make much sense. The "core" of NetSaint is not designed to be an all-in-one application. If service checks were embedded in NetSaint's 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

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 NetSaint 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...

- Danpage (alphanumeric pager software)
- QuickPage (alphanumeric pager software)
- Sendpage (paging software)
- SMS Client (command line utility for sending messages to pagers and mobile phones)

Lastly, there in an area in the contrib downloads section on the NetSaint 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...

Theory of Operation

Although the general concept of what NetSaint does is relatively easy to understand, its internal workings can sometimes be difficult to understand. In order to help you better understand how the NetSaint code works, I've provided some notes here. This isn't very extensive yet, but will be improved in later versions once everything stabilizes a bit more and I have time to catch up.

Determining Status and Reachability of Network Hosts

Click here to read up on how NetSaint determines the status and reachability of networked hosts in the process of its monitoring. This document also describes what "parent" hosts are (as defined in host definitions), and how they affect the way in which host reachability is determined.

Network Outages

Click here to read up on how NetSaint determines what hosts are causing outages on your network. This mainly pertains to the way in which the network outages CGI works, but it is still worth a quick read.

Notifications

Click here to read up on how service and host notifications work. It describes when and how notifications occur, as well as the various filters that must be passed before they can actually be sent out to individual contacts.

Service Check Scheduling

Click here to read up on how service checks are scheduled, and how scheduling differs from when checks are actually executed and their results processed.

State Types

Click here to read up on what "soft" and "hard" states are, when they occur, and the importance of the role that they play in the monitoring logic.

Time Periods

Click here to read up on how the use of time periods affects service checks, service notifications, and host notifications. This document also describes potential problems you may run into when using time periods. If you are using time periods that don't cover a 24 hour a day, 7 day a week span, you need to read this!

Plugin Development Guidelines

Plugin development for NetSaint has been moved over to SourceForge. The NetSaint plugin development project page can be found here.

The latest version of the plugin developers guide can be found at http://netsaintplug.sourceforge.net/doc/developer-guidelines.html

---

Starting NetSaint

IMPORTANT: Before you actually start NetSaint, you'll have to make sure that you have configured it properly (see the docs on the main and host files), verified the config data, and *installed some plugins on your system*! Plugins are distributed separately from NetSaint, but are necessary if you actually want to monitor anything. You can grab the plugins from the downloads page at http://www.netsaint.org

Methods For Starting NetSaint

There are basically four different ways you can start NetSaint:

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 NetSaint Manually as a Foreground Process

If you enabled the debugging options when running the configure script (and recompiled NetSaint), this would be your first choice for testing and debugging. Running NetSaint 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 NetSaint as a foreground process for testing, invoke NetSaint like this...

```
./netsaint <main_config_file>
```

Note that you must specify the path/filename of the *main* configuration file on the command line. Passing the name of the host configuration file will result in an error message and program termination.

To stop NetSaint 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 NetSaint Manually as a Background Process

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

```
./netsaint <main_config_file> &
```

Note that you must specify the path/filename of the *main* configuration file on the command line. Passing the name of the host configuration file will result in an error message and program termination.

Running NetSaint Manually as a Daemon

NetSaint 0.0.5 has some experimental code for running NetSaint as a daemon. In order to run Netsaint in daemon mode you must supply the **-d** switch on the command line as follows...

```
./netsaint -d <main_config_file>
```

Running NetSaint Automatically at System Boot

When you have tested NetSaint 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 NetSaint 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 **init-script**) is created in the base directory of the NetSaint 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. If you choose to run NetSaint in daemon mode, you can use the '**make install-daemoninit**' command to install an appropriate init script.

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

Stopping and Restarting NetSaint

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

Stopping And Restarting NetSaint

Once you have NetSaint 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 NetSaint, make sure that you have verified the configuration data using the -v command line switch, *especially* if you have made any changes to your main or host config files. If NetSaint encounters problem with one of the config files when it restarts, it will log an error and stop.

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 NetSaint 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 **netsaint** in the examples below...

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

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

Manually Stopping and Restarting NetSaint

If you aren't using an init script to start NetSaint, you'll have to do things manually. First you'll have to find the process ID that NetSaint 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 Process ID

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

```
ps axu | grep netsaint
```

The output should look something like this:

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

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

Stopping NetSaint

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

```
kill 11149
```

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

Restarting NetSaint

If you have modified the configuration data, you will want to 'restart' NetSaint and have it re-read the new configuration.

If you have changed the source code and recompiled the main netsaint executable you should *not* use this method. Instead, stop NetSaint by killing it (as outlined above) and restart it manually. Restarting NetSaint using the method below does not actually reload NetSaint **-** it just causes NetSaint to flush its current configuration, re-read the new configuration, and start monitoring all over again. To restart NetSaint, you need to send the **SIGHUP** signal to NetSaint. Assuming that the process id for NetSaint 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 NetSaint is running as on your machine.

NetSaint Plugins

Note: This document is quite outdated. Plugin development for NetSaint has been moved off to SourceForge. The NetSaint plugin development project page can be found here.

The following is a basic description of the some of the plugins that are available for use with NetSaint. If you have further questions, try running the plugins manually using the "manual execution" examples I've provided.

Other Resources

If you are confused about what the macros are all about, read up on them here. If you have questions about configuring services to actually make use these plugin examples, read up on the configuration documentation here. If you read the documentation and still can't figure things out, feel free to email me and I'll give you a hand.

Documented Plugins

TCP port plugin (check_tcp)
UDP port plugin (check_udp)
SMTP plugin (check_smtp)
POP3 plugin (check_pop)
FTP plugin (check_ftp)
NNTP plugin (check_nntp)
HTTP plugin (check_http)
Time plugin (check_time)
Ping plugin (check_ping)
DNS plugin (check_dns)
SSH plugin (check_ssh)
SNMP plugin (check_snmp)
Disk space plugin (check_disk)
Current users plugin (check_users)
Process plugin (check_procs)
Processor load plugin (check_load)
HP printer plugin (check_hpjd)
MRTG traffic plugin (check_mrtgtraf)
MRTG generic plugin (check_mrtg)
Novell server statistics plugin (check_nwstat)
Over-CR collector plugin (check_overcr)
Process image size plugin (check_vsz)
Swap usage plugin (check_swap)
Oracle database server plugin (check_oracle)

Other Undocumented Plugins

These are some other plugins which are included in the core plugin distribution, but are not documented because of a lack of time on my part. Help for each plugin is usually available by running the plugin with no command line arguments.

- **Dummy plugin** (check_dummy)
- **FPing plugin** (check_fping)
- **Game server** plugin (check_game)
- **IMAP plugin** (check_imap)
- **LDAP plugin** (check_ldap)
- **MySQL plugin** (check_mysql)
- **REAL server plugin** (check_real)
- **Reply plugin** (check_reply)
- **Breezecom wireless signal strength plugin** (check_breeze.pl)
- **WaveLAN wireless signal strength plugin** (check_wave.pl)
- **FlexLM license manager plugin** (check_flexlm.pl)
- **IRCD server plugin** (check_ircd.pl)
- **NFS plugin** (check_nfs.pl)
- **NTP plugin** (check_ntp.pl)
- **SMB share disk space plugin**

TCP Port Plugin (check_tcp)

Command Line Format:	**check_tcp <host_address> [-p port] [-wt warn_time] [-ct crit_time] [-to to_sec]**
Manual Execution Example:	**check_tcp 192.168.0.2 -p 23**
Command Definition Example:	**command[check_tcp]=/usr/local/netsaint/libexec/check_tcp $HOSTADDRESS$ -p $ARG1$**

This plugin is fairly simple - it just checks to see if it can connect to the specified host on the specified port number. A critical status is returned if the host cannot be contacted within *crit_time* seconds (if the -ct option is supplied) and a warning status is returned if the host cannot be contacted within *warn_time* seconds (if the -wt option is supplied). A critical status is returned if the plugin cannot contact the host within the timeout period specified by the *to_sec* option (default is 10 seconds).

UDP Port Plugin (check_udp)

Command Line Format:	**check_udp <host_address> [-p port] [-s send] [-e expect] [-wt warn_time] [-ct crit_time] [-to to_sec]**
Manual Execution Example:	**check_udp 192.168.0.2 -p 20796 -s "Hello, Mr. Server" -e "Hi there, Mr. Client"**
Command Definition Example:	**command[check_udp]=/usr/local/netsaint/libexec/check_udp $HOSTADDRESS$ -p $ARG1$ -s $ARG2$ -e $ARG3$**

This plugin will attempt to connect to the specified UDP port on the given host. The plugin will send the string specified by the *send* argument upon making a connection. The plugin will expect to recieve a response from the server, which should include the substring specified by the *expect* argument. If the plugin does not receive a response that contains the substring specified by the *expect* argument, it will return a critical status. A critical status is returned if the host cannot be contacted within *crit_time* seconds (if the -ct option is supplied) and a warning status is returned if the host cannot be contacted within *warn_time* seconds (if the -wt option is supplied). A critical status is returned if the plugin cannot contact the host within the timeout period specified by the *to_sec* option (default is 10 seconds).

SMTP Plugin (check_smtp)

Command Line Format:	**check_smtp <host_address> [-p port] [-e expect] [-wt warn_time] [-ct crit_time] [-to to_sec]**

Manual Execution Example:	**check_smtp 192.168.0.2**
Command Definition Example:	**command[check_smtp]=/usr/local/netsaint/libexec/check_smtp $HOSTADDRESS$**

This plugin will check to see if it can connect to the SMTP port on the specified host. The plugin will look for the string specified by the *expect* argument in the first line of the response from the host (default is "220"). Specifying an optional port number on the command line will override the default port (25). A critical status is returned if the host cannot be contacted within *crit_time* seconds (if the -ct option is supplied) and a warning status is returned if the host cannot be contacted within *warn_time* seconds (if the -wt option is supplied). A critical status is returned if the plugin cannot contact the host within the timeout period specified by the *to_sec* option (default is 10 seconds).

POP3 Plugin (check_pop)

Command Line Format:	**check_pop <host_address> [-p port] [-e expect] [-wt warn_time] [-ct crit_time] [-to to_sec]**
Manual Execution Example:	**check_pop 192.168.0.2**
Command Definition Example:	**command[check_pop]=/usr/local/netsaint/libexec/check_pop $HOSTADDRESS$**

This plugin will check to see if it can connect to the POP3 port on the specified host. The plugin will look for the string specified by the *expect* argument in the first line of the response from the host (default is "+OK"). Specifying an optional port number on the command line will override the default port (110). A critical status is returned if the host cannot be contacted within *crit_time* seconds (if the -ct option is supplied) and a warning status is returned if the host cannot be contacted within *warn_time* seconds (if the -wt option is supplied). A critical status is returned if the plugin cannot contact the host within the timeout period specified by the *to_sec* option (default is 10 seconds).

FTP Plugin (check_ftp)

Command Line Format:	**check_ftp <host_address> [-p port] [-e expect] [-wt warn_time] [-ct crit_time] [-to to_sec]**
Manual Execution Example:	**check_ftp 192.168.0.2**
Command Definition Example:	**command[check_ftp]=/usr/local/netsaint/libexec/check_ftp $HOSTADDRESS$**

This plugin will check to see if it can connect to the FTP port on the specified host. The plugin will look for the string specified by the *expect* argument in the first line of the response from the host (default is "220"). Specifying an optional port number on the command line will override the default port (21). A critical status is returned if the host cannot be contacted within *crit_time* seconds (if the -ct option is supplied) and a warning status is returned if the host cannot be contacted within *warn_time* seconds (if the -wt option is supplied). A critical status is returned if the plugin cannot contact the host within the timeout period specified by the *to_sec* option (default is 10 seconds).

NNTP Plugin (check_nntp)

Command Line Format:	**check_nntp <host_address> [-p port] [-e expect] [-wt warn_time] [-ct crit_time] [-to to_sec]**
Manual Execution Example:	**check_nntp 192.168.0.2**
Command Definition Example:	**command[check_nntp]=/usr/local/netsaint/libexec/check_nntp $HOSTADDRESS$**

This plugin will check to see if it can connect to the NNTP port on the specified host. The plugin will look for the string specified by the *expect* argument in the first line of the response from the host (default is "220"). Specifying an optional port number on the command line will override the default port (119). A critical status is returned if the host cannot be contacted within *crit_time* seconds (if the -ct option is supplied) and a warning status is returned if the host cannot be contacted within *warn_time* seconds (if the -wt option is supplied). A critical status is returned if the plugin cannot contact the host within the timeout period specified by the *to_sec* option.

HTTP Plugin (check_http)

Command Line Format:	**check_http <host_address> [-e expect] [-u url] [-p port] [-hn host_name] [-wt warn_time] [-ct crit_time] [-to to_sec] [-nohtml]**
Manual Execution Example:	**check_http 192.168.0.1**
Command Definition Example:	**command[check_http]=/usr/local/netsaint/libexec/check_http $HOSTADDRESS$**

This plugin will check to see if it can connect to the HTTP port on the specified host and retrieve the specified URL. If no URL is specified on the command line, the plugin will fetch the root document. The plugin looks for a "HTTP/1." message from the host or whatever you specify for the *expect* argument. Specifying an optional port number on the command line will override the default port (80). Specifying the optional *host_name* argument will cause the plugin to send a "Host: [host_name]" header string to the HTTP server immediately after the GET request. This is useful when trying to monitor virtual servers that use host headers. A critical status is returned if the host cannot be contacted within *crit_time* seconds (if the -ct option is supplied) and a warning status is returned if the host cannot be contacted within *warn_time* seconds (if the -wt option is supplied). A critical status is returned if the plugin cannot contact the host within the timeout period specified by the *to_sec* option (default is 10 seconds). By default, this plugin produces HTML output which provides a link to host/url that you specify on the command line. If you want to suppress the HTML output, use the *nohtml* argument.

Time Plugin (check_time)

Command Line Format:	**check_time <host_address> [-p port] [-wd warn_diff] [-cd crit_diff] [-wt warn_time] [-ct crit_time] [-to to_sec]**
Manual Execution Example:	**check_ttime 192.168.0.2 -wd 300 -cd 600**
Command Definition Example:	**command[check_time]=/usr/local/netsaint/libexec/check_tcp $HOSTADDRESS$ -wd 300 -cd 600**

This plugin will attempt to check the time on a remote host. Specifying an optional port number on the command line will override the default (37). The plugin will return a critical status if the time difference in seconds between the remote and local hosts exceeds the value of the *crit_diff* argument (if the -cd option is supplied). It will return a warning status if the time difference exceeds the value of the *warn_diff* argument (assuming the -wd option is supplied). A critical status is returned if the host cannot be contacted within *crit_time* seconds (if the -ct option is supplied) and a warning status is returned if the host cannot be contacted within *warn_time* seconds (if the -wt option is supplied). A critical status is returned if the plugin cannot contact the host within the timeout period specified by the *to_sec* option (default is 10 seconds).

Ping Plugin (check_ping)

Command Line Format:	**check_ping <host_address> <wpl> <cpl> <wrta> <crta> [-p packets] [-nohtml]**
Manual Execution Example:	**check_ping 192.168.0.1 40 100 100.0 1000.0**
Command Definition Example:	**command[check_ping]=/usr/local/netsaint/libexec/check_ping $HOSTADDRESS$ $ARG1$ $ARG2$ $ARG3$ $ARG4$**

This plugin will check to see if it can ping the specified host. It will also check the packet loss and round trip average and compare that against the warning and critical threshold levels specified for each. The <wpl> and <cpl> arguments are the warning and critical thresholds for percent packet loss, respectively. Likewise, the <wrta> and <crta> arguments are the warning and critical thresholds for round trip average (in milliseconds). The optional *packets* argument allows you to control how many ICMP ECHO packets are sent to the specified host (default is 5). By default, this plugin produces HTML output which provides a link to a traceroute CGI, which allows you to run a traceroute to the specified host via the web interface. If you want to suppress the HTML output, use the *nohtml* argument.

DNS Plugin (check_dns)

Command Line Format:	**check_dns <host_query> [dns_server]**
Manual Execution Example:	**check_dns www.onepermanentdomain.com 192.168.0.1**

Command Definition Example:	**command[check_dns]=/usr/local/netsaint/libexec/check_dns www.onepermanentdomain.com $HOSTADDRESS$**

This plugin will check to see if it can resolve the host or domain name specified by the <host_query> option. If you don't want to use the default DNS servers specified in /etc/resolv.conf you can specify a different one by supplying it as the second argument. The most useful purpose of this plugin is to check the status of one of your DNS servers. If you want to monitor one of your DNS servers, supply a well known host/domain name as the first argument and the address of your DNS server as the second argument. The "well known host/domain name" should be something that is widely know and is should always resolve to a valid IP address - try picking the name of a big search engine or corporate website.

Notes:

- This plugin uses the **nslookup** command to do the actual DNS lookup.

SSH Plugin (check_ssh)

Command Line Format:	**check_ssh <host_address> [port]**
Manual Execution Example:	**check_ssh 192.168.0.1**
Command Definition Example:	**command[check_ssh]=/usr/local/netsaint/libexec/check_ssh $HOSTADDRESS$ $ARG1$**

This plugin will attempt to connect to the SSH server on the port number specified by the *port* argument (default is port 22). Upon successfully contacting the SSH server and receiving a valid response, the plugin will display the protocol and server version information. If the plugin receives and invalid response it will display the message returned from the server and generate a warning state.

Notes:

- This plugin was contributed by Remi Paulmier.

SNMP Plugin (check_snmp)

Command Line Format:	**check_snmp <host_address> [-c community] [-o object_id] [-e eval_method] [-wv warn_value] [-cv crit_value] [-l label] [-r rate]**
Manual Execution Examples:	**check_snmp 192.168.0.1 -o ip.ipOutNoRoutes.0 -e GT -wv 5000 -cv 10000 -l "IP Packets Out W/ No Route" -r "total packets"**
Command Definition Example:	**command[check_ssnmp]=/usr/local/netsaint/libexec/check_snmp $HOSTADDRESS$ -c $ARG1$ -o $ARG2$ -e $ARG3$ -wv $ARG4$ -cv $ARG5$ -l $ARG6$ -r $ARG7$**

This plugin will attempt to obtain the value of the objectID (specified by the *object_id* argument) from the SNMP source. The value can then be evaluated with a variety of methods (listed in the table below) against optionally specified warning and critical thresholds.

Evalution Methods

Method	Description
PR	The plugin will check only see see if some sort of value was present in the server response
GT	The plugin will return a non-OK state if the data receieved is greater than either the warning or the critical threshold values
LT	The plugin will return a non-OK state if the data receieved is less than either the warning or the critical threshold values
GTE	The plugin will return a non-OK state if the data receieved is greater than or equal to either the warning or the critical threshold values
LTE	The plugin will return a non-OK state if the data receieved is less than or equal to either the warning or the critical threshold values
EQ	The plugin will return a non-OK state if the data receieved is equal to either the warning or the critical threshold values
NE	The plugin will return a non-OK state if the data receieved is not equal to either the warning or the critical threshold values

Notes:

- This plugin used the **snmpget** command distributed in the UCD-SNMP package. If you don't have it installed on your system you will need to get it from here before you can use the plugin.
- For all evalutations methods other than **PR**, it is expected that the data being evaluated is an unsigned integer.

Disk Space Plugin (check_disk)

Command Line Format:	**check_disk <wusp> <cusp> <file_system>**
Manual Execution Example:	**check_disk 85 95 /dev/hda1**
Command Definition Example:	**command[check_disk]=/usr/local/netsaint/libexec/check_disk 85 95 $ARG1$**

This plugin will check the free disk space on a specific file system. If used disk space percentage exceeds the <*cusp*> threshold value, a critical state results. If the used disk space percentage exceeds the <*wusp*> threshold value, a warning state results. The <*file_system*> argument should be in the form of */dev/hda1*, */dev/hdb2*, etc.

Notes:

- This plugin uses the **df** command to do the actual disk space check.
- This plugin is UNIX-specific, in that you cannot check the free disk space on NT or Novell servers, etc.
- This plugin can only check disk space on the host that is doing the monitoring. You can use **rsh**, **ssh**, or similiar methods to check space on remote hosts, but the netsaint_statd remote perl daemon will do the job just as well and make your life easier.

Current Users Plugin (check_users)

Command Line Format:	**check_users <wusers> <cusers>**
Manual Execution Example:	**check_users 50 75**
Command Definition Example:	**command[check_users]=/usr/local/netsaint/libexec/check_users $ARG1$ $ARG2$**

This plugin will check the number of currently logged in users. If the number of logged in users exceeds the *<cusers>* threshold value, a critical state results. If it exceeds the *<wusers>* threshold value, a warning state results.

Notes:

- This plugin uses the **who** command to do the actual check of logged in users.
- This plugin is UNIX-specific, in that you cannot check the number of logged in users on NT or Novell servers, etc.
- This plugin can only check users on the host that is doing the monitoring. You can use **rsh**, **ssh**, or similiar methods to check users on remote hosts, but the netsaint_statd remote perl daemon will do the job just as well and make your life easier.

Process Plugin (check_procs)

Command Line Format:	**check_procs <wprocs> <cprocs> [process_flags]**
Manual Execution Example:	**check_procs 5 10 ZT**
Command Definition Example:	**command[check_procs]=/usr/local/netsaint/libexec/check_procs $ARG1$ $ARG2$ $ARG3$**

This plugin will check the number of processes on the current machine. Optional process flags include R (Running), S (Sleeping), Z (Zombie), T (Stopped or Traced), and D (Uninterruptible Sleep). If no process flags are specified, the plugin will count all types of processes. The example above will check for processed that are in either a zombie state or are stopped/traced.

Notes:

- This plugin uses the **ps** command to do the actual check of processes.
- This plugin is UNIX-specific, in that you cannot check processes on NT or Novell servers, etc.
- This plugin can only check processes on the host that is doing the monitoring. You can use **rsh**, **ssh**, or similiar methods to check processes on remote hosts, but the netsaint_statd remote perl daemon will do the job just as well and make your life easier.

Processor Load Plugin (check_load)

Command Line Format:	**check_load <wload1> <cload1> <wload5> <cload5> <wload15> <cload15>**
Manual Execution Example:	**check_load 95 100 90 95 80 90**
Command Definition Example:	**command[check_load]=/usr/local/netsaint/libexec/check_load $ARG1$ $ARG2$ $ARG3$ $ARG4$ $ARG5$ $ARG6$**

This plugin will check the load average on the local machine over 1, 5, and 15 minute time periods using the data found in the **/proc/loadavg** file. A critical status is returned if the 1, 5, or 15 minute load averages exceed the *<cload1>*, *<cload5>*, or *<cload15>* thresholds specified. A warning status is returned if the 1, 5, or 15 minute load averages exceed the *<wload1>*, *<wload5>*, or *<wload15>* thresholds specified.

Notes:

- This plugin was contributed by Felipe Gustavo de Almeida.
- This plugin is UNIX-specific, in that you cannot check processor loads on NT or Novell servers, etc.
- This plugin can only check processor load on the host that is doing the monitoring. You can use **rsh**, **ssh**, or similiar methods to check processor load on remote hosts, but the netsaint_statd remote perl daemon will do the job just as well and make your life easier.

HP Printer Plugin (check_hpjd)

Command Line Format:	**check_hpjd <address> [community]**
Manual Execution Example:	**check_hpjd 192.168.0.1**
Command Definition Example:	**command[check_hpdj]=/usr/local/netsaint/libexec/check_hpdj $HOSTADDRESS$ $ARG1$**

This plugin will check the status of an HP printer that has a JetDirect© card installed. This plugin will return a critical status when the printer is turned off, a warning status when it has a paper jam, is offline, out of paper, low on toner, etc. Specifying an optional [community] argument on the command line will override the default SNMP community used in the communication with the printer (public).

My guess is that you don't want to get alert emails or pages everytime a printer jams or gets turned off at the end of the day, right? If so, read the FAQ on monitoring printers.

Notes:

- This plugin works *only* on HP brand printers with JetDirect© cards installed, and it may not work on all of them.
- This plugin used the **snmpget** command distributed in the UCD-SNMP package. If you don't have

it installed on your system you will need to get it from here before you can use the plugin.

- The idea (and some code) for this plugin came from Jim Trocki's printer alert script in his *mon* program.
- The JetDirect© card must have the TCP/IP protocol stack enabled and configured. External JetDirect© devices must have TCP/IP enabled on each port they want to monitor.
- *JetDirect* is copyrighted by Hewlett-Packard. Don't sue me please. :-)

MRTG Traffic Plugin (check_mrtgtraf)

Command Line Format:	**check_mrtgtraf <log_file> <expire_minutes> <AVG|MAX> <iwl> <icl> <owl> <ocl>**
Manual Execution Example:	**check_mrtgtraf /home/httpd/html/mrtg/router1.log 10 AVG 1000000 1500000 1000000 1500000**
Command Definition Example:	**command[check_mrtgtraf]=/usr/local/netsaint/libexec/check_mrtgtraf $ARG1$ 10 AVG $ARG2$ $ARG3$ $ARG4$ $ARG5$**

This plugin will check a traffic log file generated by MRTG and generate alerts if the incoming or outgoing rates (in Bytes/sec) exceed the specified thresholds. If the newest entry in the log file is more than *<expire_minutes>* old, the plugin will return a warning level. Specifying *AVG* or *MAX* as the third argument will control whether the plugin looks at average or maximum values in the log file (default is average). If the incoming or outgoing rates exceed the *<iwl>* or *<owl>* warning thresholds, respectively, a warning status is returned. If the rates exceed the *<icl>* or *<ocl>* critical thresholds, a critical status is returned. Command line thresholds are in Bytes/sec.

Notes:

- This plugin requires MRTG to do the actual traffic rate monitoring. You can download MRTG from http://ee-staff.ethz.ch/~oetiker/webtools/mrtg/mrtg.html.
- The traffic rates reported by the plugin are slightly different from those reported by MRTG - I'll look into this.

MRTG Generic Plugin (check_mrtg)

Command Line Format:	**check_mrtg <log_file> <expire_minutes> <AVG|MAX> <column> <vwl> <vcl> <label> [rate]**
Manual Execution Example:	**check_mrtg /home/httpd/html/mrtg/router1.log 10 AVG 1 1000000 1500000 In Bytes/Sec**
Command Definition Example:	**command[check_mrtg]=/usr/local/netsaint/libexec/check_mrtg $ARG1$ 10 AVG $ARG2$ $ARG3$ $ARG4$ $ARG5$ $ARG6$**

This plugin will check a log file generated by MRTG and generate alerts if the value of the specified variable exceeds the specified thresholds. If the newest entry in the log file is more than *<expire_minutes>* old, the plugin will return a warning level. Specifying *AVG* or *MAX* as the third command line argument will control whether the plugins looks at the average or maximum value of the variable in the log file (default is average). This plugin will only check one of the two possible variables recorded by MRTG. If you want to monitor the first variable, specify **1** as the *<column>* argument. If you want to monitor the second variable, specify **2** as the argument. If the value of the specified variable exceeds the *<vcl>* threshold, a critical status is returned. If it exceeds the *<vwl>* threshold, a warning status is returned. The *<label>* argument is used in the output to identify what type of data is being monitored. Examples include **Connections**, **"User Connections"**, **"Processor Utilization"**, **"Traffic In"**, etc. The optional [rate] argument is used to give the variable value som meaning. Examples include **%**, **Packets/Sec**, **Bytes/Sec**, **"Errors Per Second"**, etc.

Notes:

- This plugin requires MRTG to do the actual monitoring. You can download MRTG from http://ee-staff.ethz.ch/~oetiker/webtools/mrtg/mrtg.html.

Process Image Size Plugin (check_vsz)

Command Line Format:	**check_vsz <wsize> <csize> [command_name]**
Manual Execution Example:	**check_vsz 100000 150000 betaprogram**
Command Definition Example:	**command[check_vsz]=/usr/local/netsaint/libexec/check_vsz $ARG1$ $ARG2$ $ARG3$**

This plugin will check for processes whose total image size (in bytes) exceeds the warning or critical thresholds given on the command line (*<wsize>* and *<csize>*, respectively). With no [command_name] specified, every command that shows up in the **ps** command is evaluated. Otherwise, only jobs with names matching the [command_name] argument are examined. This program is particularly useful if you have to run a piece of commercial software that has a potential memory leak and you want to watch its memory usage carefully.

Notes:

- This plugin was contributed by Karl DeBisschop.
- Used in conjuction with an approriate service event handler, you could use this plugin to automatically kill and restart a program which is consuming memory.

Swap Usage Plugin (check_swap)

Command Line Format:	**check_swap <wswap> <cswap>**
Manual Execution Example:	**check_swap 100000 120000**
Command Definition Example:	**command[check_swap]=/usr/local/netsaint/libexec/check_swap $ARG1$ $ARG2$**

This plugin will check all the swap partitions on the local machine and return a warning or critical status if the percent of swap usage is above the <*wswap*> or <*cswap*> thresholds.

Notes:

- This plugin was contributed by Karl DeBisschop.

Novell Server Statistics Plugin (check_nwstat)

Command Line Format:	**check_nwstat <host_address> [-p port] [-v variable] [-wv warn_value] [-cv crit_value] [-to to_sec]**
Manual Execution Example:	**check_nwstat 192.168.1.5 -v LOAD5 -wv 80 -cv 95**
Command Definition Example:	**command[check_nwstat]=/usr/local/netsaint/libexec/check_nwstat $HOSTADDRESS$ -v $ARG1$ -wv $ARG2$ -cv $ARG3$**

This plugin allows you to monitor disk usage, connections, cache buffers, and LRU sitting time on your Novell servers. The plugin obtains server information by talking to the MRTGEXT NLM (distributed with James Drews' MRTG extension - see below) on the Novell server. The default port used to communicate with the server NLM is 9999. If the value for a given variable is higher than the specified critical threshold (or possibly lower - see note below), a critical status is returned. If the value is highter than the specified warning threshold (or possibly lower - see note below), a warning status is returned. Only one variable can be checked at a time. Valid variables are listed below. A critical status is returned if the plugin cannot contact the host within the timeout period specified by the *to_sec* option (default is 10 seconds).

Variable	Description
LOAD1	1 minute load average
LOAD5	5 minute load average
LOAD15	15 minute load average
CONNS	Number of licensed connections
LTCH	Percentage of long term cache hits
CBUFF	Number of total cache buffers
CDBUFF	Number of dirty cache buffers
LRUM	LRU sitting time in minutes
VPF<volume>	Percent free space on volume <volume>
VKF<volume>	KB of free space on volume <volume>

Notes:

- Both the critical and warning thresholds are optional. If neither is specified, the plugin returns an ok status (except in the case of a communication or configuration error).
- The critical thresholds should be **lower** than the warning thresholds for volume free space, cache buffers, and LRU sitting time because a lower value for these variables is worse than a higher number!
- This plugin requires that **MRTGEXT.NLM** (distributed in James Drews' MRTG Extensions for

Netware package) be running on the Novell servers you wish to monitor. You don't have to use MRTG to use this plugin - just run MRTGEXT.NLM on your servers.

Over-CR Collector Plugin (check_overcr)

Command Line Format:	**check_overcr <host_address> [-p port] [-v variable] [-wv warn_value] [-cv crit_value] [-to to_sec]**
Manual Execution Example:	**check_overcr 192.168.1.5 -v LOAD5 -wv 80 -cv 95**
Command Definition Example:	**command[check_overcr]=/usr/local/netsaint/libexec/check_overcr $HOSTADDRESS$ -v $ARG1$ -wv $ARG2$ -cv $ARG3$**

This plugin allows you to monitor active network connections, uptime, running processes, disk usage, and processor load on remote servers. The plugin obtains server information by talking to Over-CR collector that runs on the remote server (see note below). The default port used to communicate with the Over-CR collector is 2000. If the value for a given variable is higher than the specified critical threshold (or possibly lower - see note below), a critical status is returned. If the value is highter than the specified warning threshold (or possibly lower - see note below), a warning status is returned. Only one variable can be checked at a time. Valid variables are listed below. A critical status is returned if the plugin cannot contact the host within the timeout period specified by the *to_sec* option (default is 10 seconds).

Variable	Description
LOAD1	1 minute load average
LOAD5	5 minute load average
LOAD15	15 minute load average
DPU<filesys>	Percent *used* disk space on file system **filesys**
PROC<process>	Number of running processes with a name of **process**
NET<port>	Number of active TCP/IP connection on port **port**
UPTIME	System uptime in seconds

Notes:

- Both the critical and warning thresholds are optional. If neither is specified, the plugin returns an ok status (except in the case of a communication or configuration error).
- The critical thresholds should be **lower** than the warning thresholds for UPTIME, since a lower uptime value is worse than a higher one!
- This plugin requires that Eric Molitor's Over-CR collector be running on the servers you wish to monitor.

Oracle Database Server Plugin (check_oracle)

Command Line Format:	**check_oracle <host_address>**
Manual Execution Example:	**check_oracle 192.168.0.2**

Command Definition Example:	**command[check_oracle]=/usr/local/netsaint/libexec/check_oracle $HOSTADDRESS$**

This plugin will attempt to see if an Oracle database server on the specified host can be contacted. The plugin returns a critical status if the database server cannot be reached.

Notes:

- This plugin was contributed by Jorge Sanchez
- This plugin requires the **tnsping** program distributed with Oracle SQL*Net software.
- This plugin is a shell script and does not need to be compiled

PostgresQL Database Plugin (check_pgsql)

Command Line Format:	**check_pgsql [twarn] [tcrit] [host] [port] [db] [user] [password]**
Manual Execution Example:	**check_pgsql 120 3600 192.168.0.1 5432 mydatabase**
Command Definition Example:	**command[check_pgsql]=/usr/local/netsaint/libexec/check_pgsql 120 3600 $HOSTADDRESS$ $ARG1$ $ARG2$ $ARG3$ $ARG4$**

This plugin will attempt to connect to the specified postgresQL database on the host. Connection refusals and timeouts result in a critical status. If the connection time exceeds the *tcrit* value in seconds, a critical status results. If the connection time exceeds the *twarn* value in seconds, a warning status results. All other errors result in an unknown status.

All arguments to this plugin are optional. The defaults are equivalent to **"check_pgsql 120 3600 localhost 5432"** and assume that the user that runs the plugin can connect to the database without a password.

Security Tip: If you use the [username] and [password] arguments in a command or service definition, you should take steps to ensure that this information does not end up getting displayed in the HTML pages that NetSaint generates!

Notes:

- This plugin was contributed by Karl DeBisschop
- This plugin requires that the PostgresQL libraries be installed on your machine in order to be compiled
- This plugin requires that the backend for remote machines use TCP/IP (start postmaster with the -i option)

Log File Pattern Detector Plugin (check_log)

Command Line Format:	**check_log <log_file> <old_log_file> <pattern>**

Manual Execution Example:	**check_oracle /var/log/messages /usr/local/netsaint/var/check_log.old.loginfailure 'LOGIN FAILURE'**
Command Definition Example:	**command[check_log]=/usr/local/netsaint/libexec/check_log $ARG1$ $ARG2$**

This plugin will scan a log file (specified by the *log_file* option) for a specific pattern (specified by the *pattern* option). Successive calls to the plugin script will only report **new** pattern matches in the log file, since an copy of the log file from the previous run is saved to *old_log_file*. The plugin returns a critical status if the log file cannot be located. The first time the plugin is executed it will initialize the data it needs and return with an ok status. Successive executions will return an OK status if no pattern matches are detected in the changes to the original log file. If one or more pattern matches are found, the plugin will return a CRITICAL status and print a string in the following format: *(x) last_entry*, where *x* is the total number of matches found and *last_entry* is the last matching entry from the log file.

Notes:

- This plugin is a shell script and does not need to be compiled
- This plugin is very "expensive" as far as disk space is concerned, because it keeps an old copy of the original log file for each pattern that you want to check. I am aware of this and know this should be done better - I wrote this simply as an example. A long term solution is a tie-in with a log watcher like SWATCH.

UPS Plugin (check_ups)

Command Line Format:	**check_ups <host_address> [-p port] [-u ups] [-v variable] [-wv warn_value] [-cv crit_value] [-to to_sec]**
Manual Execution Example:	**check_ups 192.168.0.3 -u mybigups -v BATTPCT -wv 80 -cv 40**
Command Definition Example:	**command[check_ups]=/usr/local/netsaint/libexec/check_ups $HOSTADDRESS$ -u $ARG1$ -v $ARG2$ -wv $ARG3$ -cv $ARG4$**

This plugin attempts to determine the status of an UPS (Uninterruptible Power Supply) on a remote host (or the local host) that is being monitored with Russel Kroll's "Smart UPS Tools" package. If the UPS is online or calibrating, the plugin will return an OK state. If the battery is on it will return a WARNING state. If the UPS is off or has a low battery the plugin will return a CRITICAL state. You may also specify a variable to check (such as temperature, utility voltage, battery load, etc.) as well as warning and critical thresholds for the value of that variable. If the remote host has multiple UPS that are being monitored, you will have to use the *ups* option to specify which UPS to check. A critical status is returned if the host cannot be contacted within *crit_time* seconds (if the -ct option is supplied) and a warning status is returned if the host cannot be contacted within *warn_time* seconds (if the -wt option is supplied). A critical status is returned if the plugin cannot contact the host within the timeout period specified by the *to_sec* option (default is 10 seconds).

Variable	Description

UTILITY	The *difference* (absolute value) between 120.0 VAC and the voltage that is being supplied to the UPS via the utility line
BATTPCT	Percent of battery charge remaining
LOADPCT	Percent load being put on the UPS by the electronic devices attached to it
TEMP	The temperature (in degrees Farenheit) of the UPS

Notes:

- If the UPS being monitored does not support one or more of the variables that the plugin monitors, those variables will not be checked.
- The critical thresholds should be **lower** than the warning thresholds for BATTPCT, since a lower battery reserve is worse than a higher one!
- This plugin requires that the UPSD daemon distributed with Russel Kroll's "Smart UPS Tools" be installed on the remote host. If you don't have the package installed on your system, you can download it from http://www.exploits.org/~rkroll/smartupstools.

SSH Plugin Executor (check_by_ssh)

Command Line Format:	**check_by_ssh <user> <host> <command>**
Manual Execution Example:	**check_by_ssh**
Command Definition Example:	**command[check_by_ssh]=/usr/local/netsaint/libexec/check_by_ssh $ARG1$ $ARG2$ $ARG3$**

This is not so much a plugin as it is a plugin wrapper. It basically allows you to execute plugins on a remote host by using SSH. The SSH username on the remote host is specified with the *user* argument, the address of the remote host is specified by the *host* argument, and the command that should be executed on the remote host is specified by the *command* argument.

Notes:

- This plugin was contributed by Karl DeBisschop
- You must have SSH installed and configured properly in order to use this plugin
- The plugins that you want to execute on the remote host must be installed on the remote host!

NetSaint Process Plugin (check_netsaint)

Command Line Format:	**check_netsaint <status_log> <expire_minutes> <process_string>**
Manual Execution Example:	**check_netsaint /usr/local/netsaint/var/status.log 5 "/usr/local/netsaint/bin/netsaint -d /usr/local/netsaint/etc/netsaint.cfg"**
Command Definition Example:	**command[check_netsaint]=/usr/local/netsaint/libexec/check_netsaint $ARG1$ $ARG2$ $ARG3$**

This plugin is used to check the status of the NetSaint process on the local host. The plugin will check the contents of the status log (specified by the *status_log* argument) and make sure that the most recent entry is no older than the number of minutes specified by the *expire_minutes* argument. If the status log is older than this value, a warning state results. The plugin will also use the **ps** command to search for a running process that matches the *process_string* argument. If the plugin cannot locate a match of the process string, it assumes that NetSaint is not running and returns a critical state.

Notes:

- This plugin can be used by the CGIs to check the status of the NetSaint process. This is done by using the process_check_command option in the CGI configuration file.
- This plugin can be used when implementing redundant monitoring hosts, although it must be executed on the remote hosts as explained in this FAQ.

NetSaint Addons

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

Index

cl_status - Console interface for viewing status of monitored services

Author:	Adam Bowen
Description:	This program is designed to run in a console and display the current status of monitored hosts and services. It uses ncurses to display as many status lines as possible based on the screen size settings. It will also make the console beep and flash if there are any problems. You can specify the rate at which the status information is refreshed from the NetSaint status log.

neat - Web-based administration interface for NetSaint

Author:	Jason Blakey
Description:	NetSaint Easy Administration Tool (NEAT) is a web administration interface for NetSaint that is written in Perl. It allows you to add/edit/delete definitions in your host configuration file and restart NetSaint upon completion of the configuration changes. Unlike nsa, it does not require a database to store your configuration data.

netsaint_mrtg - MRTG scripts for graphing NetSaint host and service status information

Author:	Richard Mayhew	
Overview:	Allows you to produce MRTG graphs of NetSaint host and service status information	
Files:	mrtghost_total.pl	- Perl script that obtains the total number of hosts that are up and down
	mrtgsvc_total.pl	- Perl script that obtains the total number of services that are up and down
	mrtgsvchost_total.pl	- Perl script that obtains the total number of services that are up and down on a particular server
	mrtgsvctyp_total.pl	- Perl script that obtains the total number of services (of a particular type) that are up and down

Description: This package includes two scripts which allow MRTG to generate graphs of host and service status totals, as reported by NetSaint. The scripts scan the NetSaint status log to determine the total number of services or hosts that have problems or are okay. Examples of how to incorporate the scripts with MRTG are provided in the **README.mrtg** file.

Notes:
- You must be running MRTG to actually make use of this package

netsaint_statd - Perl daemon and plugins for monitoring remote host information

Author: Nick Reinking

Overview: Allows you to monitor disk usage, load average, processes, and users on remote hosts.

Files:

netsaint_statd	- Perl daemon that runs on remote hosts
check_disk.pl	- Perl plugin that is executed by NetSaint to check remote host disk information [single disks]
check_all_disks.pl	- Perl plugin that is executed by Netsaint to check remote disk information. [all but ignored disks]
check_users.pl	- Perl plugin that is executed by NetSaint to check remote host user information
check_procs.pl	- Perl plugin that is executed by NetSaint to check remote host process information
check_load.pl	- Perl plugin that is executed by NetSaint to check remote host load information
Changelog	- Changes recently made to netsaint_statd
README	- Command options and arguments (for hosts.cfg)

Description: netsaint_statd is a daemon which allows a NetSaint host to get information such as process count, users, disk usage, and load information using the corresponding plugin scripts. The daemon does not process the system information in anyway. It merely collects the information and hands it back to the calling script to do with as it pleases.

The daemon script is designed in such a way as to allow for easy porting to other OSes (as long as you have Perl installed). Adding other checks should also be easy by adding the appropriate sections in the command list for netsaint_statd. Currently supported OSs are HP-UX, Linux, Solaris/SunOS, IRIX, OSF1, FreeBSD and NEXTSTEP. The only requirements for getting your OS supported are the standard UNIX utilities. Host restrictions are just a small list of IPs to listen to (or you can have it listen to everybody). netsaint_statd is designed to allow easy addition of extra remote system checks.

Notes:
- You'll have to modify the first line of code in each file to match the location of your perl binary.

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 NetSaint 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 NetSaint host. The check_nrpe plugin then passes the remote plugin's output and return code back to NetSaint 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

nrpep - Service and plugin for executing plugins on remote hosts

Author: Adam Jacob

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

Description: NetSaint Remote Plugin Executor/Perl (NRPEP) was designed as a replacemnt for the netsaint_statd and nrpe addons. Although this addon is similiar in function to nrpe, it is written in Perl and implements TripleDES encryption for the data in transit. It is also designed to run under inetd and make use of the TCP Wrappers package for access control.

Notes:

- Requires two Perl modules: *Crypt-TripleDES-0.24* and *Digest-MD5-2.09*

nsa - Web-based administration package for NetSaint

Author: Daniel Burke

Description: Daniel Burke has created this excellent addon - named "NetSaint Administrator" - to fill the need for an more user-friendly means of configuring NetSaint. This package allows you to edit your configuration data (hosts, services, contacts, timeperiods, etc.) via a web interface. Configuration data is stored in a MySQL database and written to a text file in the proper configuration file format when you're ready. The CGIs can also run NetSaint with the -v option to verify the contents of your configuration file. This is an excellent application for anyone who either hates the native config file format or just wants an easier interface for managing the configuration data.

Notes:

- You must have MySQL v2.22.25 or higher installed, Perl5 with DBI and MySQL DBD installed, and a general knowledge of how to create/delete databases and tables in MySQL in order to use this package.

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 NetSaint.

Files:	nsca	- Daemon that runs on the central NetSaint 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 NetSaint 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 NetSaint. The client can be used as a standalone program or can be integrated with remote NetSaint servers that run an ocsp command to setup a distributed monitoring environment.

Notes:

- As of the first beta version, the client and daemon use an elementary XOR operation to "encrypt" the data being passed across the network. This is obviously not very secure! This was implemented only as of proof-of-concept solution for allowing the nsca daemon to "trust" the data the client sends. A strong private-key encryption method will hopefully be incorporated into the client and daemon soon, so long as I can implement the encryption function on multiple platforms (NT, Novell, etc).

pscwatch - Watchdog daemon that ensures passive service checks are being submitted

Author: Me

Overview: Ensures that passive service checks are being submitted at regular intervals.

Description: This addon's sole purpose in life is to ensure that passive service checks are being submitted to NetSaint on a regular basis. This addon is designed to be used on a central monitoring server when setting up a distributed monitoring environment.

Determining Status and Reachability of Network Hosts

Monitoring Services on Down or Unreachable Hosts

The main purpose of NetSaint 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, NetSaint will not be able to monitor the services associated with that host.

NetSaint 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, NetSaint 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, NetSaint assumes that there is a problem with the host. In this situation NetSaint 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, NetSaint 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 NetSaint - no routers or firewalls lay between them. Figure 1 shows an example network layout. Host A is running NetSaint 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.

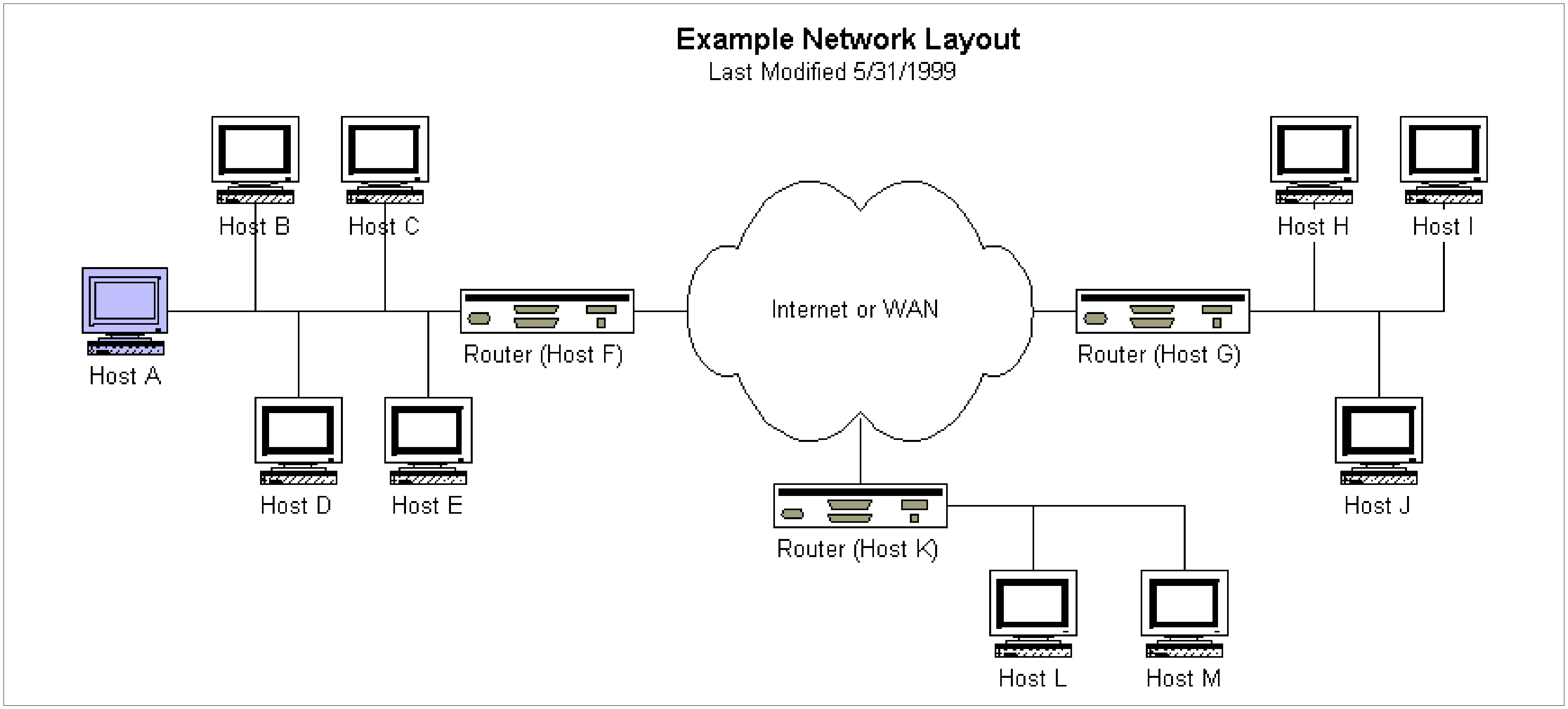
The <*parent_host*> option in the host defintion for a "local" host should be left blank, as local hosts have no depencies or "parents" - that's why they're local.

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 NetSaint 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, NetSaint assumes the host is up. If the command returns any other status level, NetSaint will assume the host is down.

Figure 1.



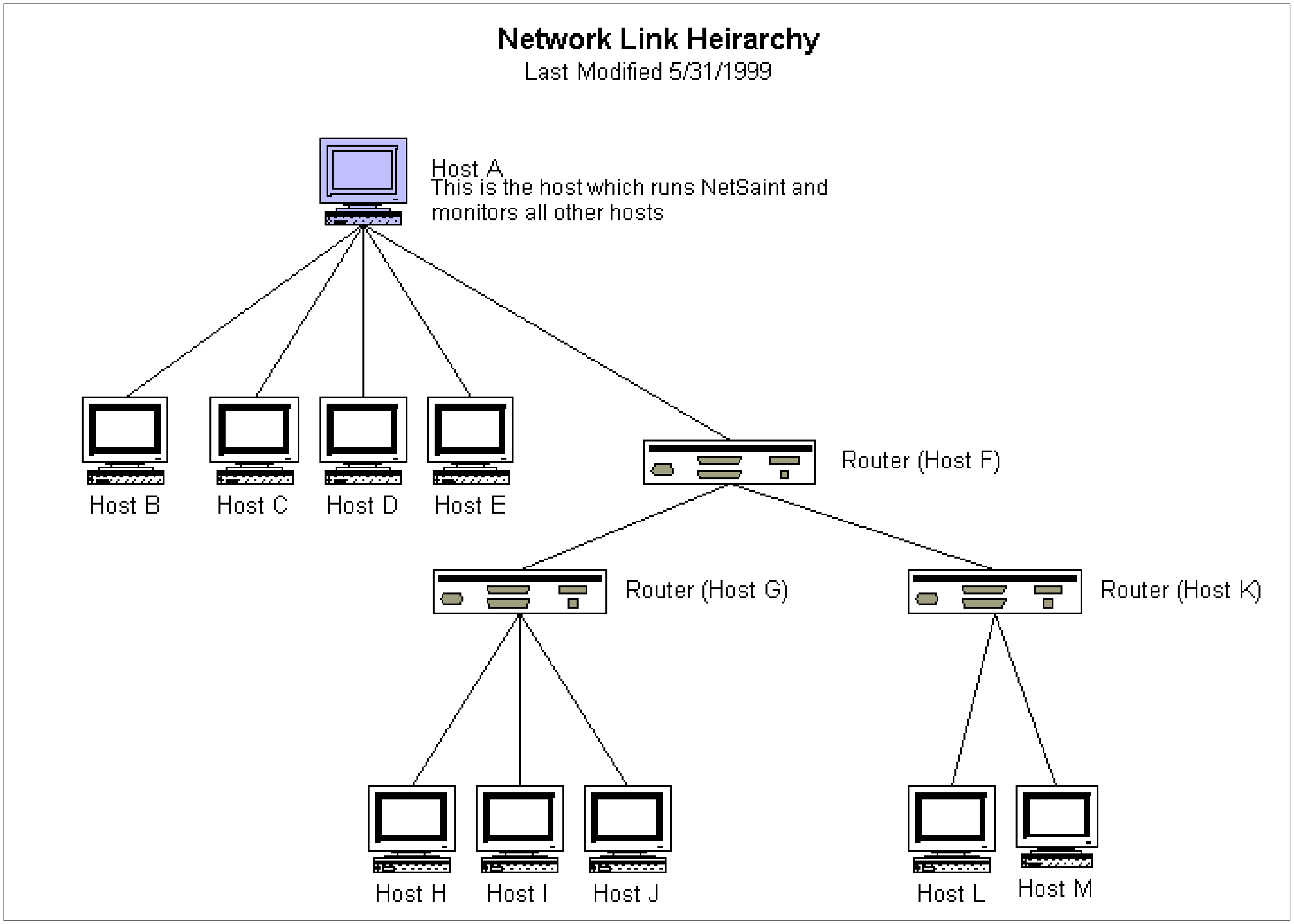
Remote Hosts

"Remote" hosts are hosts that reside on a different network segment than the host running NetSaint. 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 NetSaint.

The <**parent_host**> option in the host defintion for a "remote" host should be the short name of the host 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 NetSaint 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_host>** option introduced in 0.0.4 allows NetSaint to do this.

If a host check command for a remote host returns a non-OK state, NetSaint 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, NetSaint 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.

A logic diagram for the host check function is included below in Figure 3. It illustrates how NetSaint determines if a host is down or unreachable.

Figure 3.

State Types

Introduction

The current state of services and hosts is determined by two components: the status of the service or host and the *type* of state it is in. There are two state types in NetSaint - "soft" states and "hard" states. State types are a crucial part of NetSaint's 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, NetSaint 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_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_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**").
- NetSaint 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_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_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.

Logic Diagrams

Soft and hard states can be a little difficult to understand. The logic diagrams found here may be of some help.

Time Periods

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

Introduction

With the release 0.0.4 the notion of time periods was introduced. 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

Previous to release 0.0.4, NetSaint 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 NetSaint when the service can be checked. When NetSaint 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, NetSaint 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 <*svc_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 NetSaint is allowed to send notifications out regarding problems or recoveries for that host. When a host notification is about to get sent out, NetSaint 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 NetSaint 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, NetSaint will not send the notification out to *any* contacts. A logic diagram outlining the basic decisions NetSaint makes when sending out host notifications can be found here.

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 NetSaint is allowed to send notifications out regarding problems or recoveries for that service. When a service notification is about to get sent out, NetSaint 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 NetSaint 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, NetSaint will not send the notification out to *any* contacts. A logic diagram outlining the basic decisions NetSaint makes when sending out service notifications can be found here.

Potential Problems With Contact Notifications

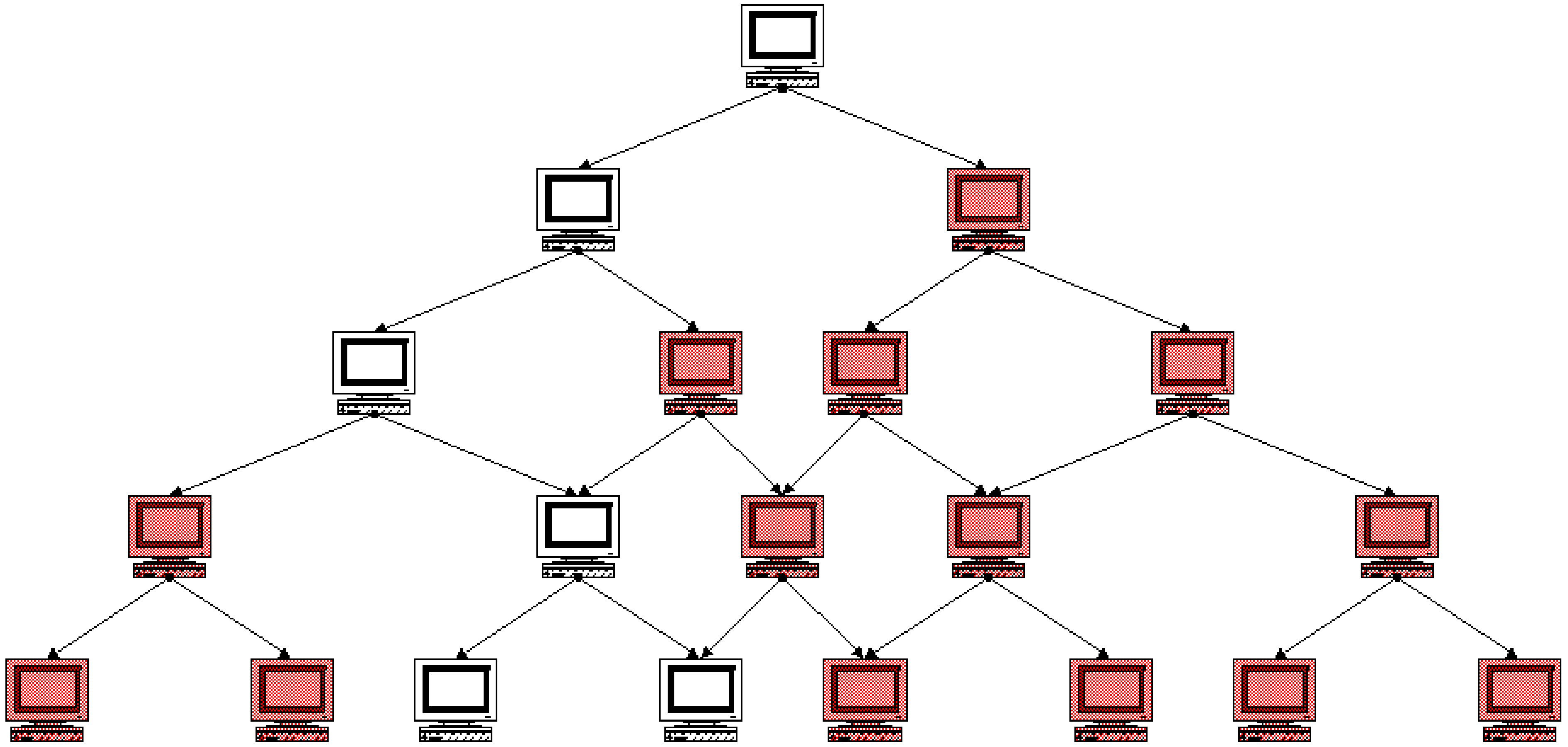
There aren't really any major problems that you'll run into with using time periods to create custom contact notification times. You do, however, need to be aware that contacts may not always be notified of a service or host problem or recovery. If the time isn't right for both the host or service notification 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 NetSaint 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.

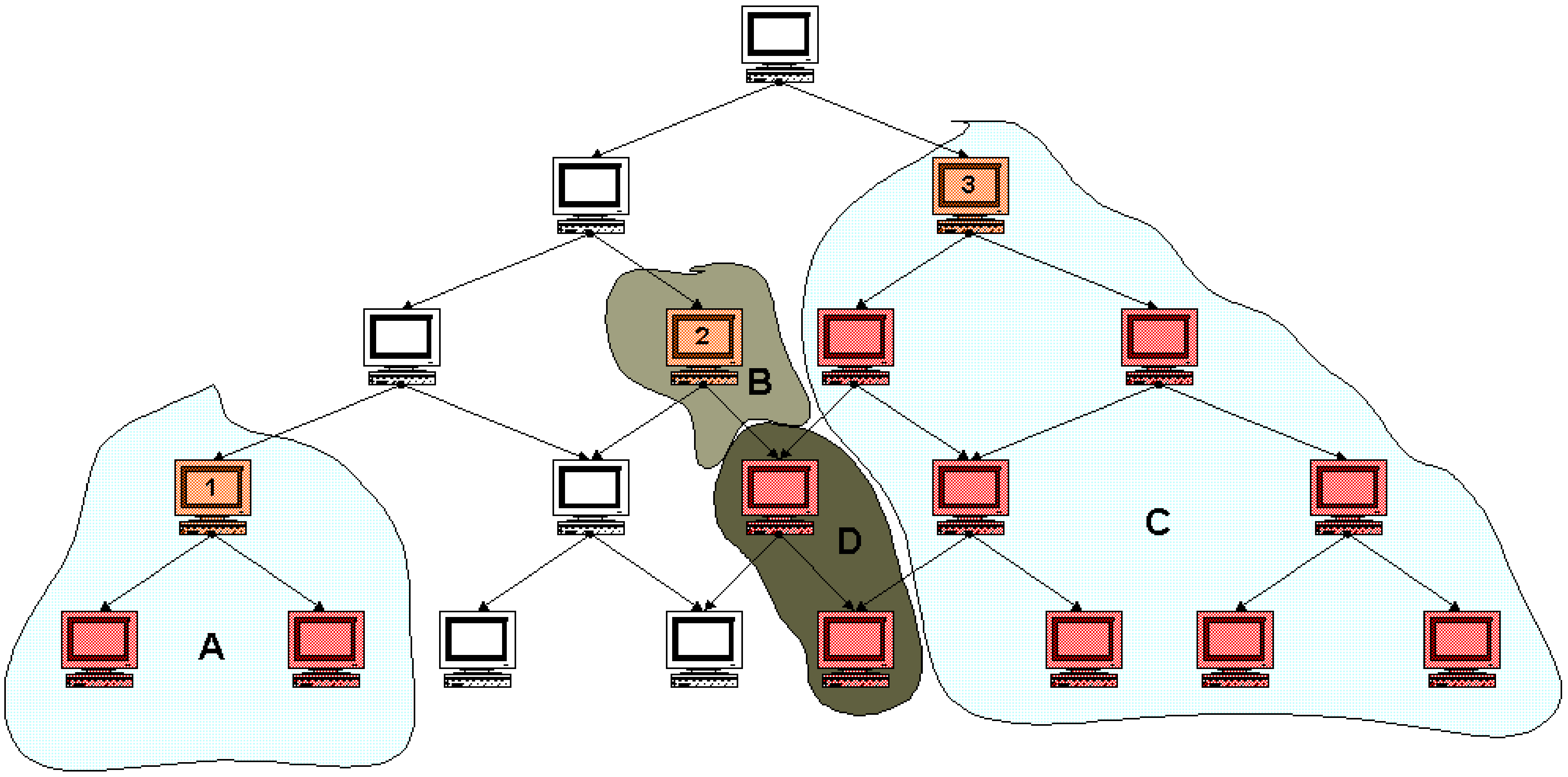
Network Outages

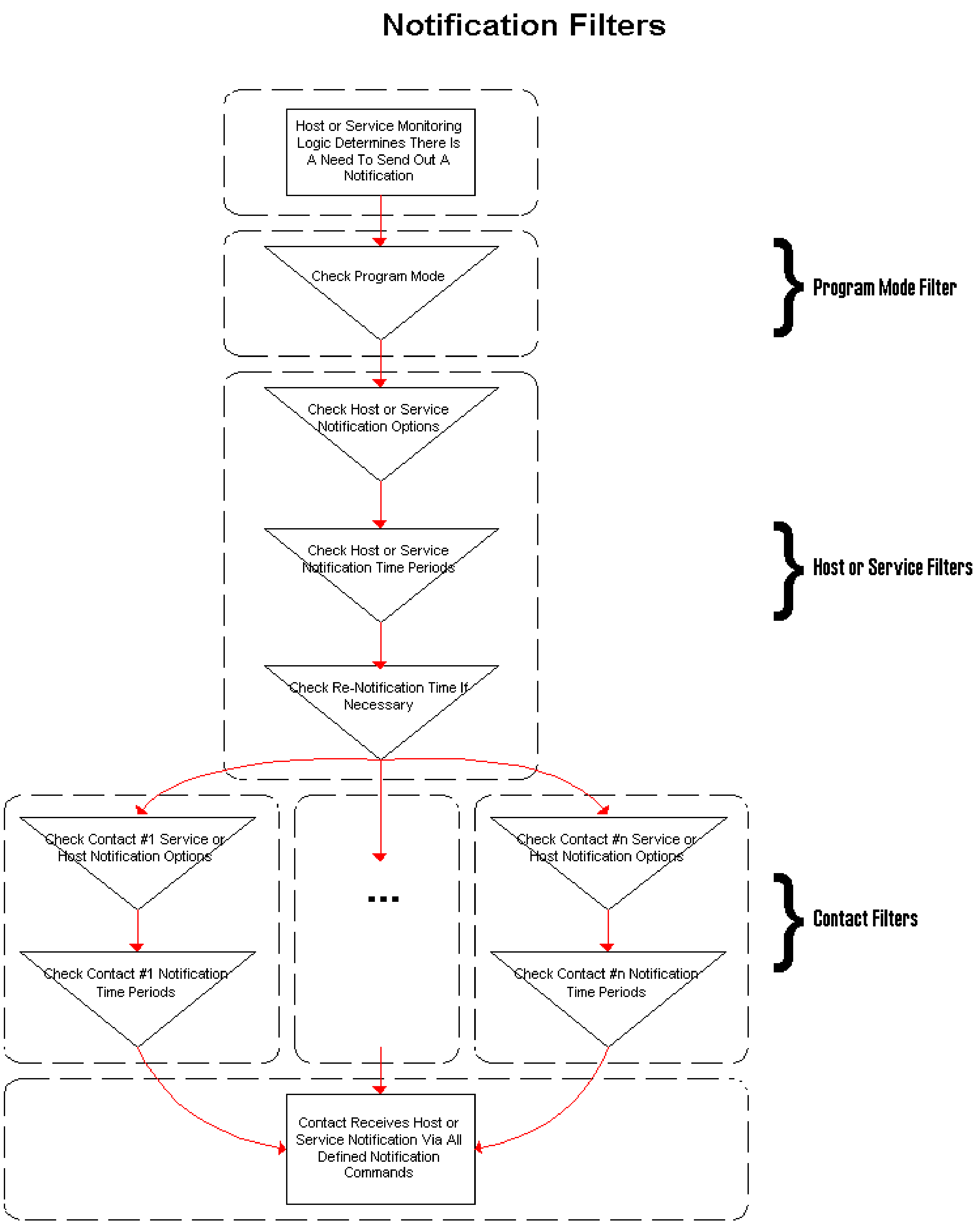
Last Modified 02/26/2000



Cause and Effect Of Network Outages

Last Modified 02/26/2000



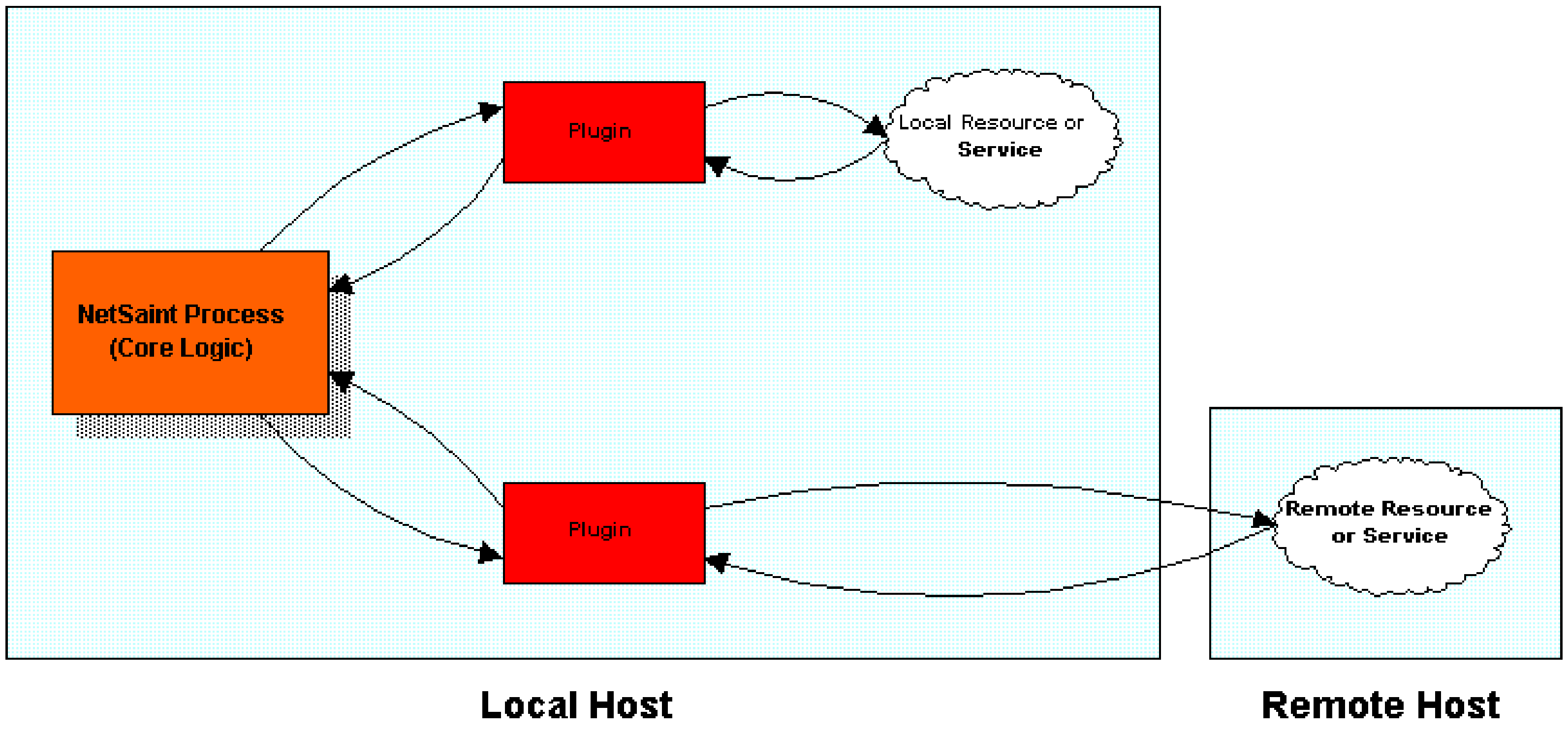

Notification Filters
Host or Service Monitoring Logic Determines There Is A Need To Send Out A Notification
Check Program Mode
Program Mode Filter
Check Host or Service Notification Options
Check Host or Service Notification Time Periods
Host or Service Filters
Check Re-Notification Time If Necessary
Check Contact #1 Service or Host Notification Options
Check Contact #1 Notification Time Periods
...
Check Contact #n Service or Host Notification Options
Check Contact #n Notification Time Periods
Contact Filters
Contact Receives Host or Service Notification Via All Defined Notification Commands

Plugin Theory

Introduction

Unlike many other monitoring tools, NetSaint does not include any internal mechanisms for checking the status of services, hosts, etc. Instead, NetSaint relies on external programs (called plugins) to do the all the dirty work. NetSaint 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 NetSaint. NetSaint 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 NetSaint. NetSaint 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 NetSaint for processing. A more complex diagram on how plugins work can be found in the documentation on passive service checks.


Plugin
Local Resource or Service
NetSaint Process (Core Logic)
Plugin
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 NetSaint. 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 NetSaint 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, NetSaint 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...

Using Plugins For Service Checks

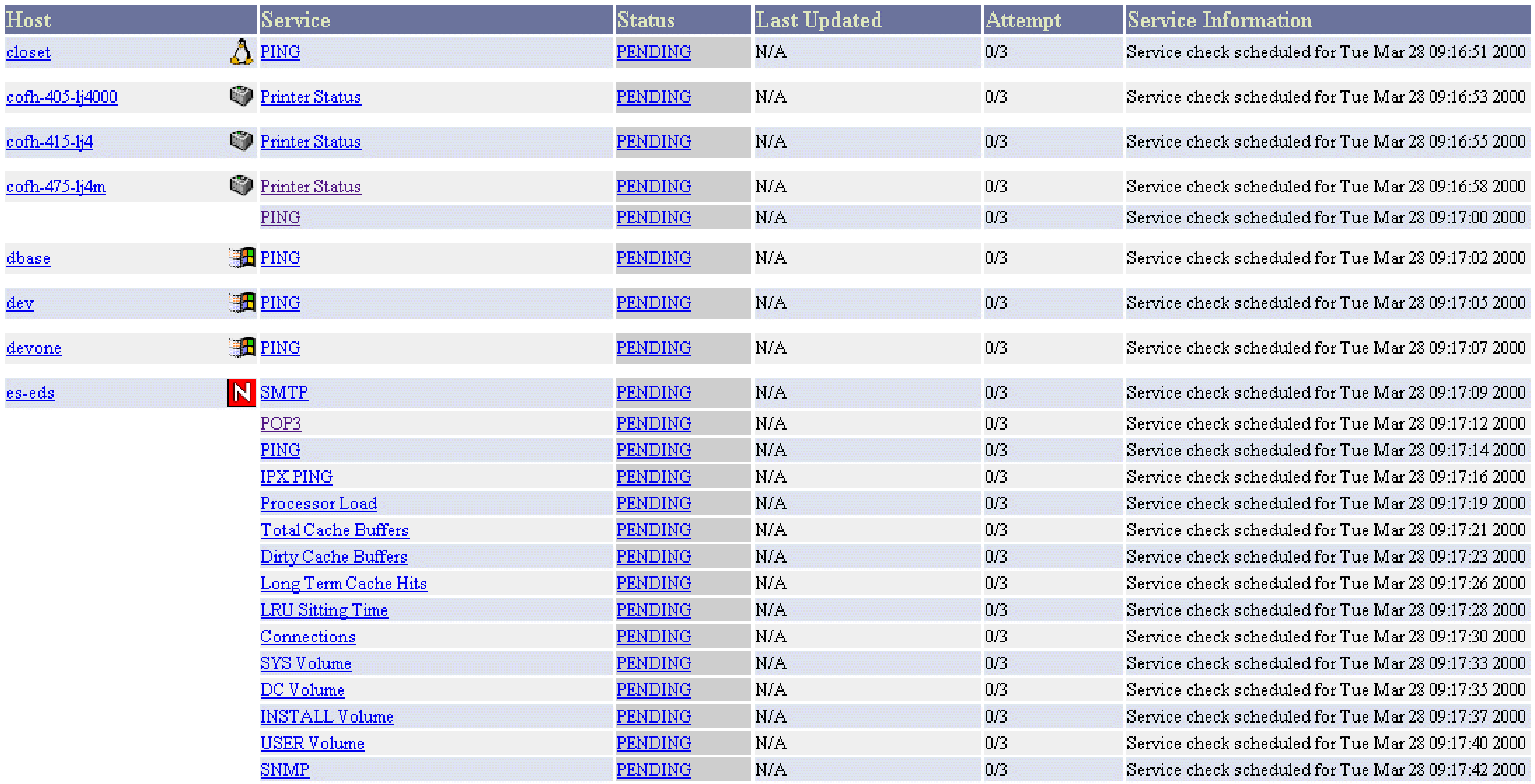
The correlation between plugins and service checks should be fairly obvious. When NetSaint needs to check the status of a particular service that you have defined, it will execute the plugin you specified in the <*check_command*> argument of the service definition. The plugin will check the status of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

Using Plugins For Host Checks

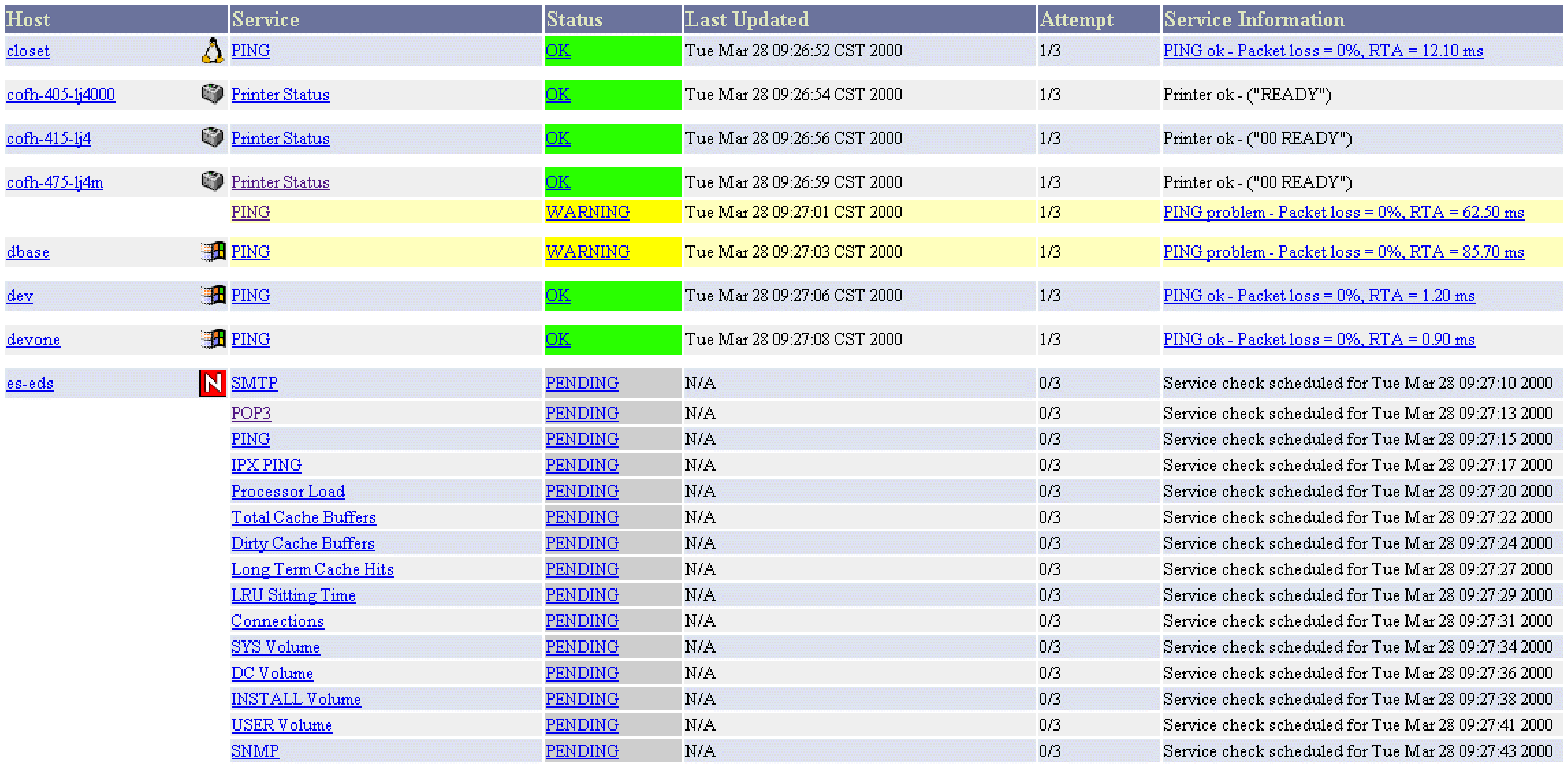
Using plugins to check the status of hosts may be a bit more difficult to understand. In each host definition you use the <*host_check_command*> argument to specify a plugin that should be executed to check the status of the host. Host checks are not performed on a regular basis - they are executed only 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, NetSaint will believe that the host is down.

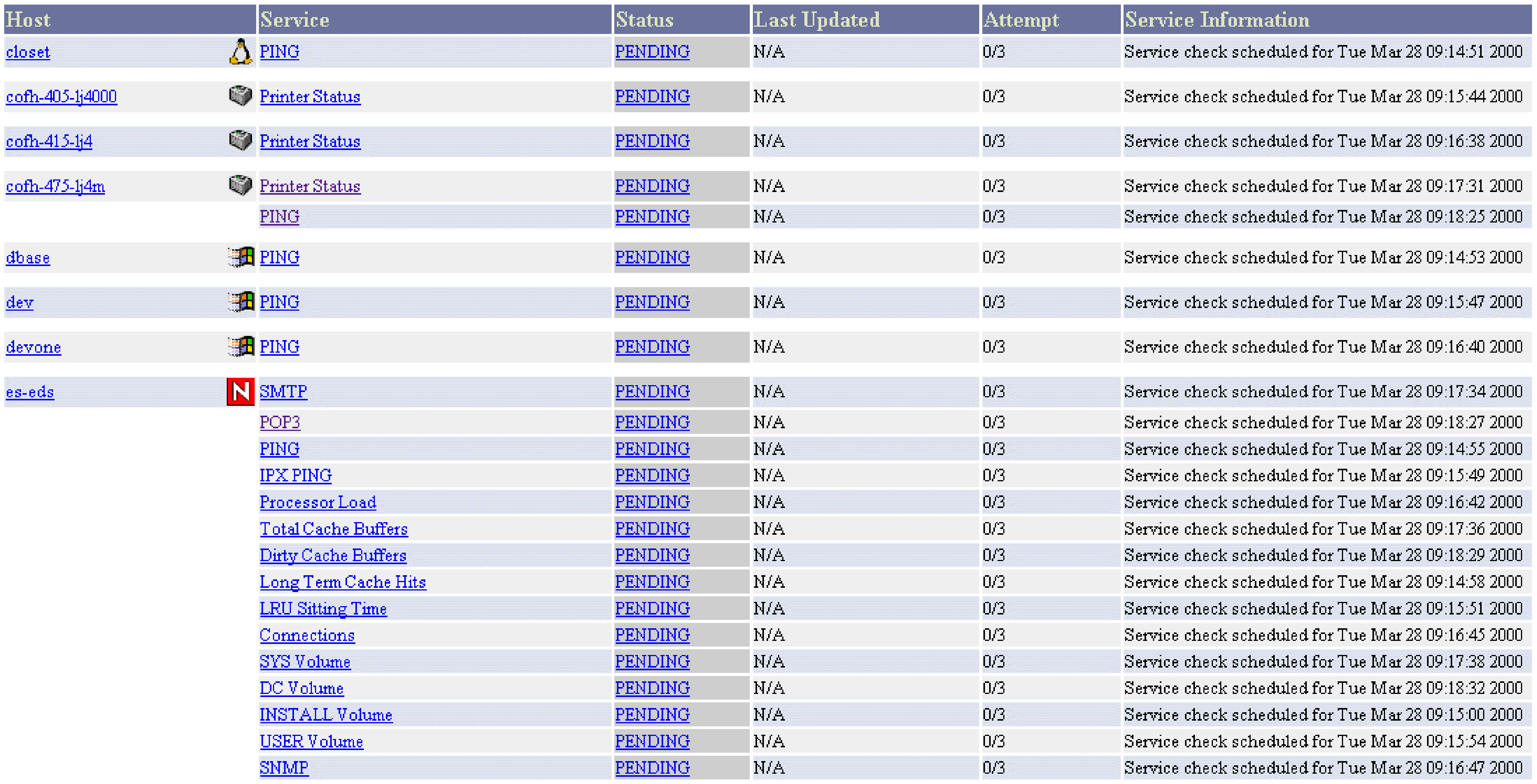
In most situations, you'll want to use a plugin which checks to see if the host can be pinged, as this is the most common method of telling whether or not a host is up. However, if you were monitoring some kind of super-fantastic toaster, you might want to use a plugin that would check to see if the 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".



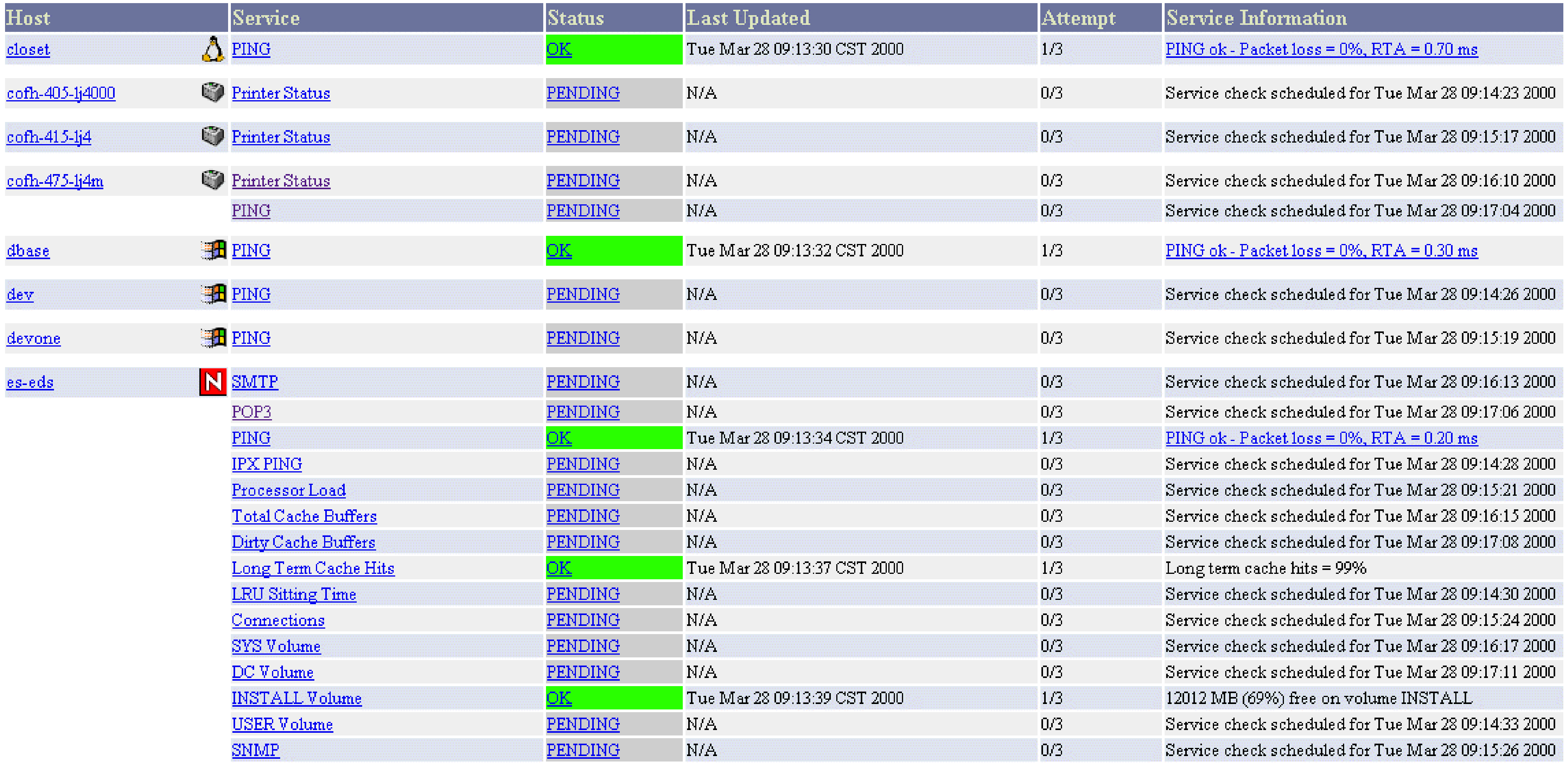
Host	Service	Status	Last Updated	Attempt	Service Information
closet	PING	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:16:51 2000
cofh-405-lj4000	Printer Status	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:16:53 2000
cofh-415-lj4	Printer Status	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:16:55 2000
cofh-475-lj4m	Printer Status	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:16:58 2000
	PING	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:17:00 2000
dbase	PING	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:17:02 2000
dev	PING	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:17:05 2000
devone	PING	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:17:07 2000
es-eds	SMTP	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:17:09 2000
	POP3	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:17:12 2000
	PING	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:17:14 2000
	IPX PING	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:17:16 2000
	Processor Load	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:17:19 2000
	Total Cache Buffers	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:17:21 2000
	Dirty Cache Buffers	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:17:23 2000
	Long Term Cache Hits	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:17:26 2000
	LRU Sitting Time	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:17:28 2000
	Connections	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:17:30 2000
	SYS Volume	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:17:33 2000
	DC Volume	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:17:35 2000
	INSTALL Volume	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:17:37 2000
	USER Volume	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:17:40 2000
	SNMP	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:17:42 2000



Host	Service	Status	Last Updated	Attempt	Service Information
closet	PING	OK	Tue Mar 28 09:26:52 CST 2000	1/3	PING ok - Packet loss = 0%, RTA = 12.10 ms
cofh-405-lj4000	Printer Status	OK	Tue Mar 28 09:26:54 CST 2000	1/3	Printer ok - ("READY")
cofh-415-lj4	Printer Status	OK	Tue Mar 28 09:26:56 CST 2000	1/3	Printer ok - ("00 READY")
cofh-475-lj4m	Printer Status	OK	Tue Mar 28 09:26:59 CST 2000	1/3	Printer ok - ("00 READY")
	PING	WARNING	Tue Mar 28 09:27:01 CST 2000	1/3	PING problem - Packet loss = 0%, RTA = 62.50 ms
dbase	PING	WARNING	Tue Mar 28 09:27:03 CST 2000	1/3	PING problem - Packet loss = 0%, RTA = 85.70 ms
dev	PING	OK	Tue Mar 28 09:27:06 CST 2000	1/3	PING ok - Packet loss = 0%, RTA = 1.20 ms
devone	PING	OK	Tue Mar 28 09:27:08 CST 2000	1/3	PING ok - Packet loss = 0%, RTA = 0.90 ms
es-eds	SMTP	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:27:10 2000
	POP3	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:27:13 2000
	PING	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:27:15 2000
	IPX PING	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:27:17 2000
	Processor Load	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:27:20 2000
	Total Cache Buffers	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:27:22 2000
	Dirty Cache Buffers	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:27:24 2000
	Long Term Cache Hits	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:27:27 2000
	LRU Sitting Time	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:27:29 2000
	Connections	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:27:31 2000
	SYS Volume	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:27:34 2000
	DC Volume	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:27:36 2000
	INSTALL Volume	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:27:38 2000
	USER Volume	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:27:41 2000
	SNMP	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:27:43 2000



Host	Service	Status	Last Updated	Attempt	Service Information
closet	PING	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:14:51 2000
cofh-405-lj4000	Printer Status	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:15:44 2000
cofh-415-lj4	Printer Status	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:16:38 2000
cofh-475-lj4m	Printer Status	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:17:31 2000
	PING	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:18:25 2000
dbase	PING	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:14:53 2000
dev	PING	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:15:47 2000
devone	PING	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:16:40 2000
es-eds	SMTP	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:17:34 2000
	POP3	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:18:27 2000
	PING	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:14:55 2000
	IPX PING	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:15:49 2000
	Processor Load	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:16:42 2000
	Total Cache Buffers	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:17:36 2000
	Dirty Cache Buffers	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:18:29 2000
	Long Term Cache Hits	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:14:58 2000
	LRU Sitting Time	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:15:51 2000
	Connections	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:16:45 2000
	SYS Volume	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:17:38 2000
	DC Volume	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:18:32 2000
	INSTALL Volume	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:15:00 2000
	USER Volume	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:15:54 2000
	SNMP	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:16:47 2000



Host	Service	Status	Last Updated	Attempt	Service Information
closet	PING	OK	Tue Mar 28 09:13:30 CST 2000	1/3	PING ok - Packet loss = 0%, RTA = 0.70 ms
cofh-405-lj4000	Printer Status	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:14:23 2000
cofh-415-lj4	Printer Status	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:15:17 2000
cofh-475-lj4m	Printer Status	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:16:10 2000
	PING	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:17:04 2000
dbase	PING	OK	Tue Mar 28 09:13:32 CST 2000	1/3	PING ok - Packet loss = 0%, RTA = 0.30 ms
dev	PING	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:14:26 2000
devone	PING	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:15:19 2000
es-eds	SMTP	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:16:13 2000
	POP3	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:17:06 2000
	PING	OK	Tue Mar 28 09:13:34 CST 2000	1/3	PING ok - Packet loss = 0%, RTA = 0.20 ms
	IPX PING	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:14:28 2000
	Processor Load	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:15:21 2000
	Total Cache Buffers	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:16:15 2000
	Dirty Cache Buffers	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:17:08 2000
	Long Term Cache Hits	OK	Tue Mar 28 09:13:37 CST 2000	1/3	Long term cache hits = 99%
	LRU Sitting Time	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:14:30 2000
	Connections	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:15:24 2000
	SYS Volume	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:16:17 2000
	DC Volume	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:17:11 2000
	INSTALL Volume	OK	Tue Mar 28 09:13:39 CST 2000	1/3	12012 MB (69%) free on volume INSTALL
	USER Volume	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:14:33 2000
	SNMP	PENDING	N/A	0/3	Service check scheduled for Tue Mar 28 09:15:26 2000

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 NetSaint. 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 NetSaint 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.

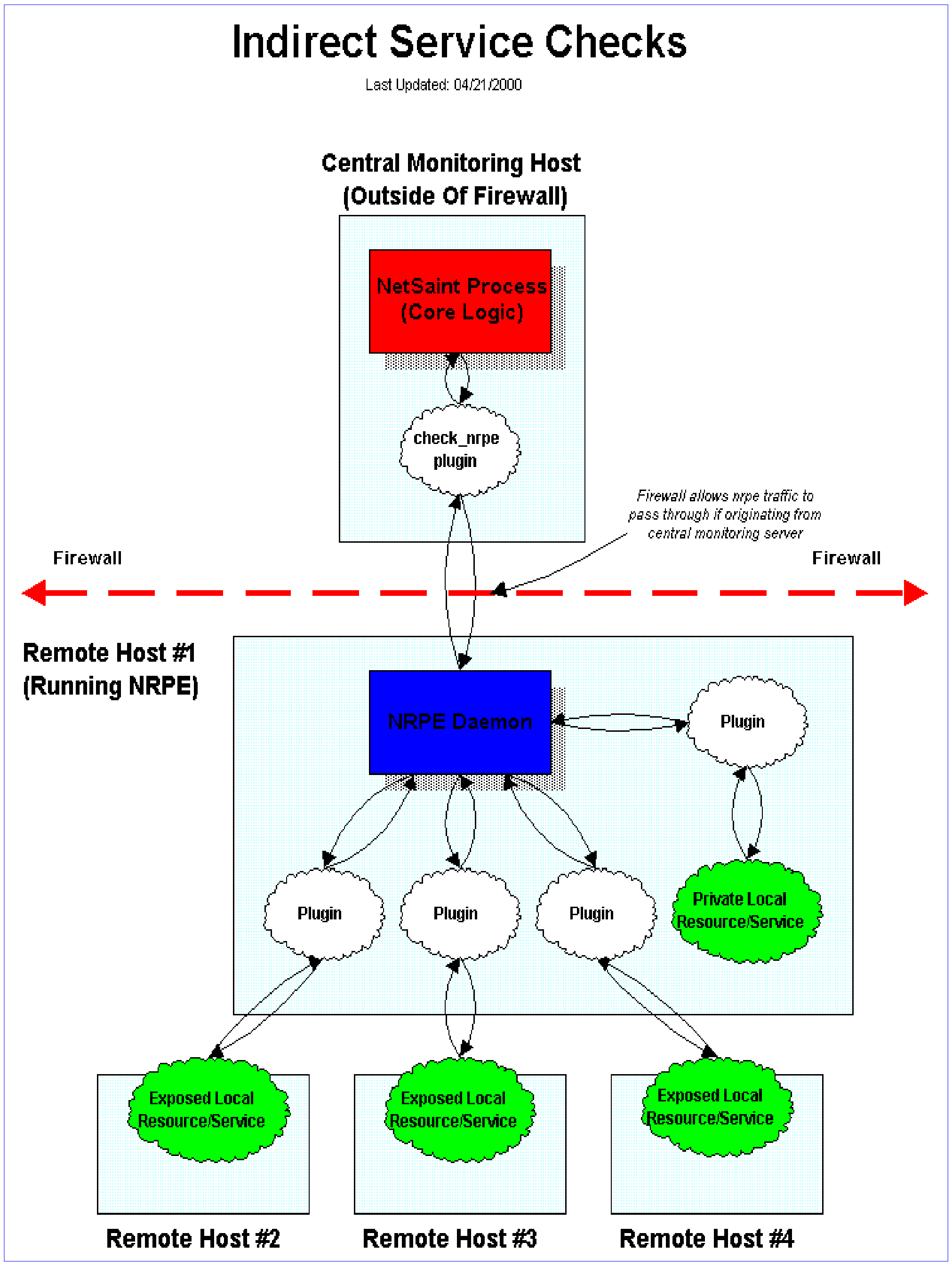
Indirect checks are useful for:

- 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. The nrpep and netsaint_statd can also be used to perform indirect checks.

Indirect Service Checks

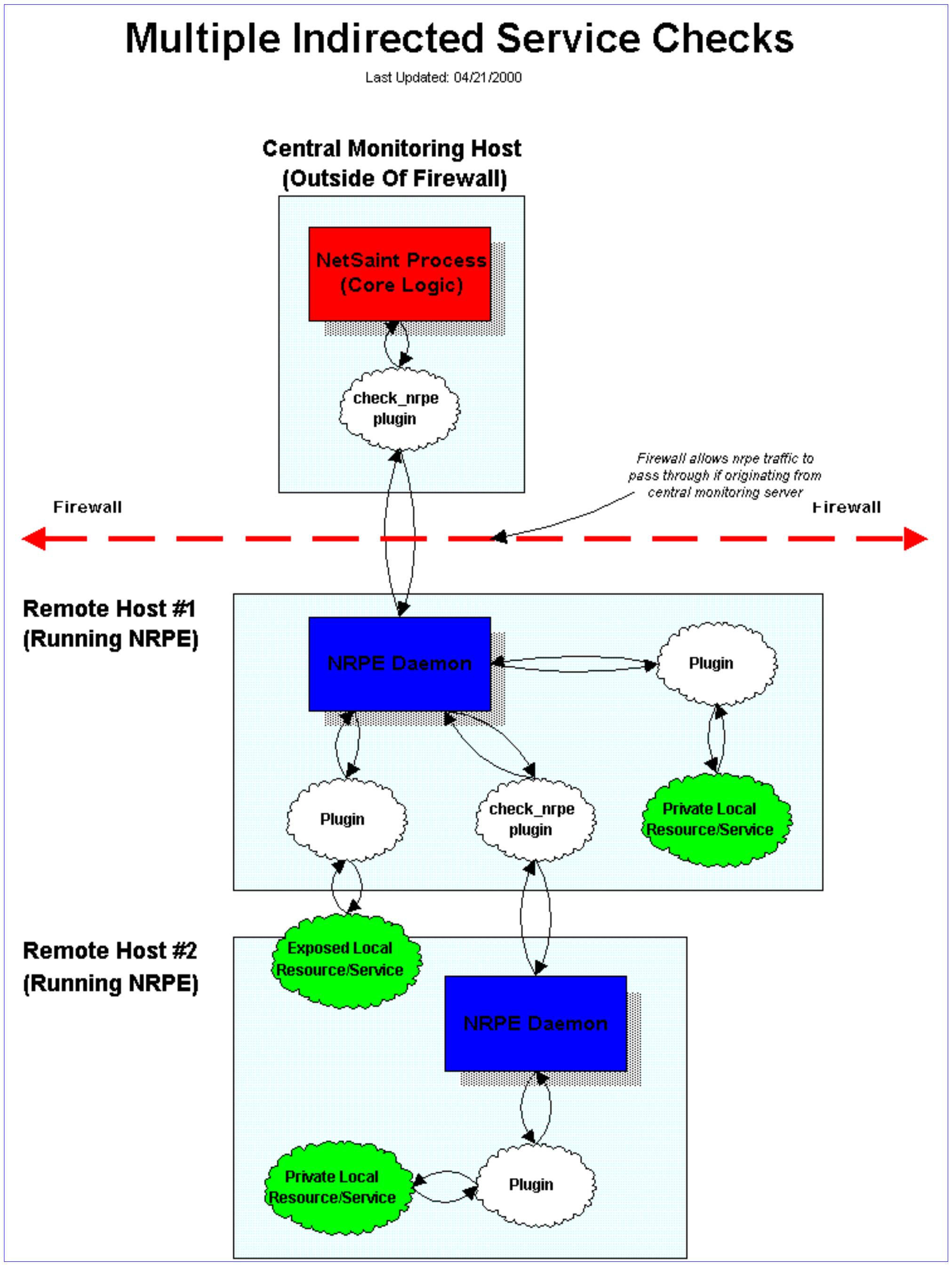
The diagram below shows how indirect service checks work. Click the image for a larger version...


Indirect Service Checks
Last Updated: 04/21/2000
Central Monitoring Host (Outside Of Firewall)
NetSaint 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 NetSaint 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 NetSaint 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.

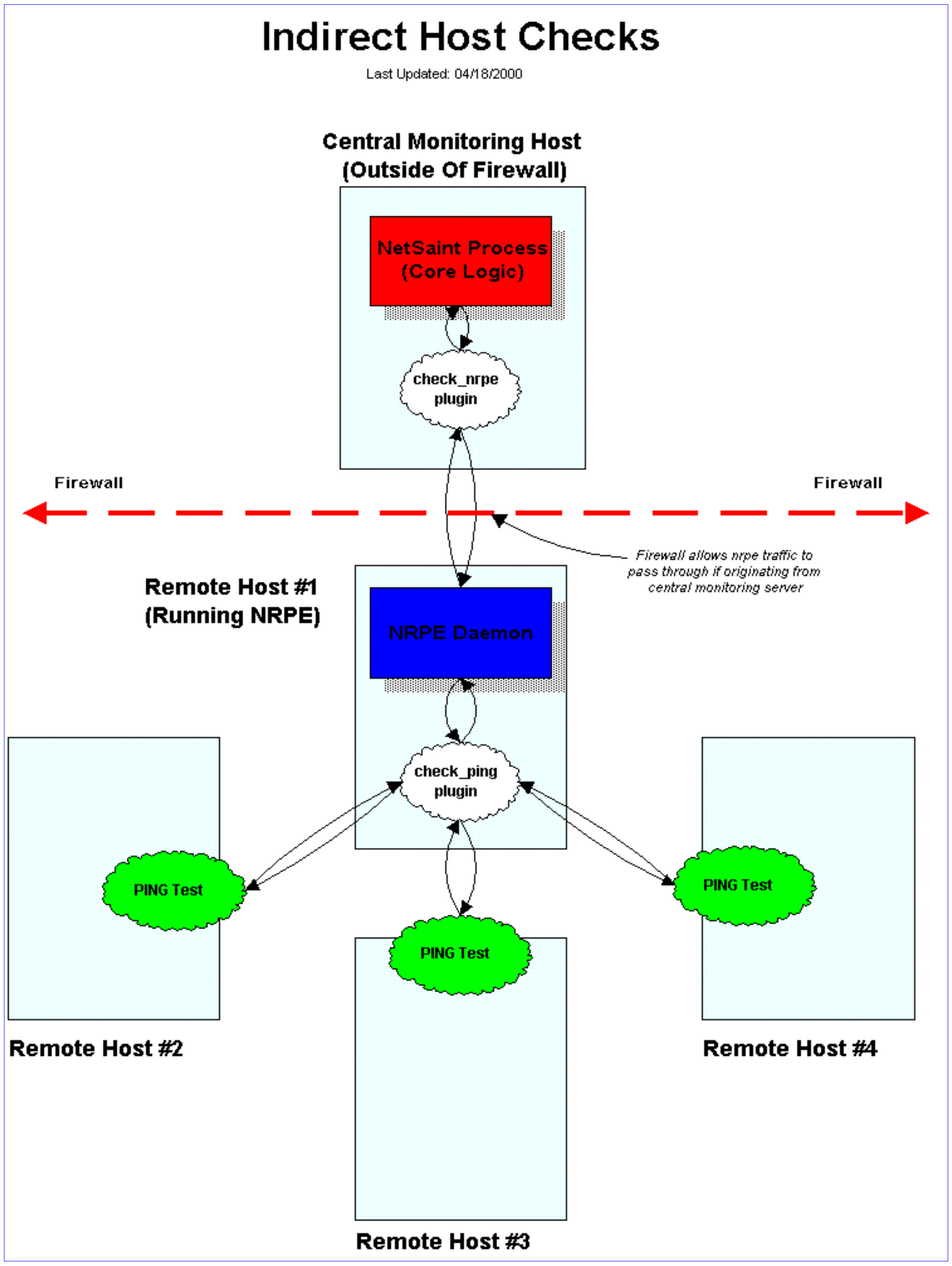
The diagram below show how multiple indirect service checks work. Notice how the *nrpe* daemon is running on hosts #1 and #2. The copy that runs on host #2 is used to allow the *nrpe* agent on host #1 to perform a check of a "private" service on host #2. "Private" services are things like process load, 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: 04/21/2000
Central Monitoring Host (Outside Of Firewall)
NetSaint 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

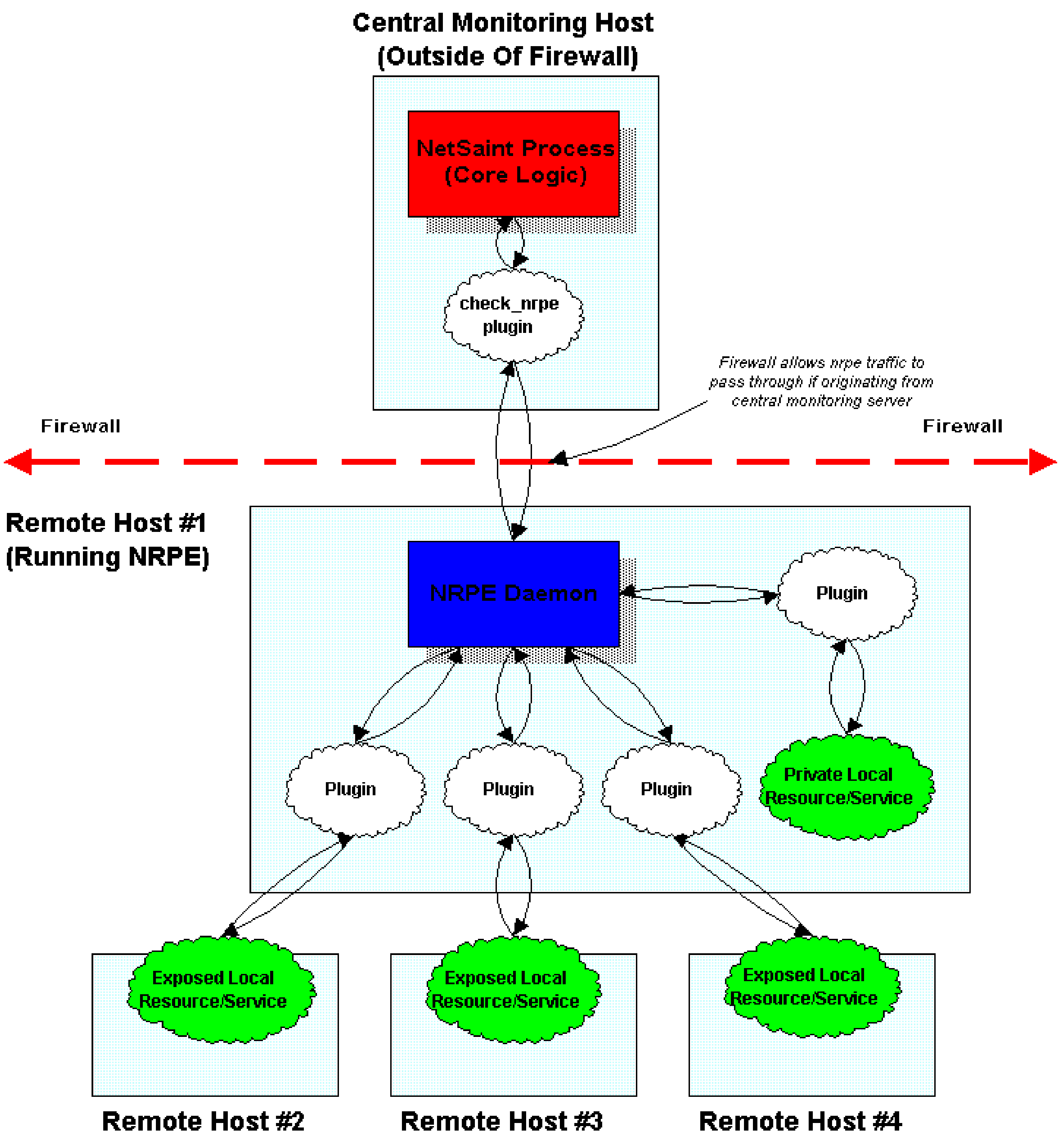
Indirect host checks work on the same principle as indirect service checks. Basically, the plugin used in the host check command asks an intermediary agent (i.e. a daemon running on a remote host) to perform the host check for it. Indirect host checks are useful when the remote hosts being monitored 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 NetSaint 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 NetSaint 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: 04/18/2000
Central Monitoring Host (Outside Of Firewall)
NetSaint 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 #4
Remote Host #3

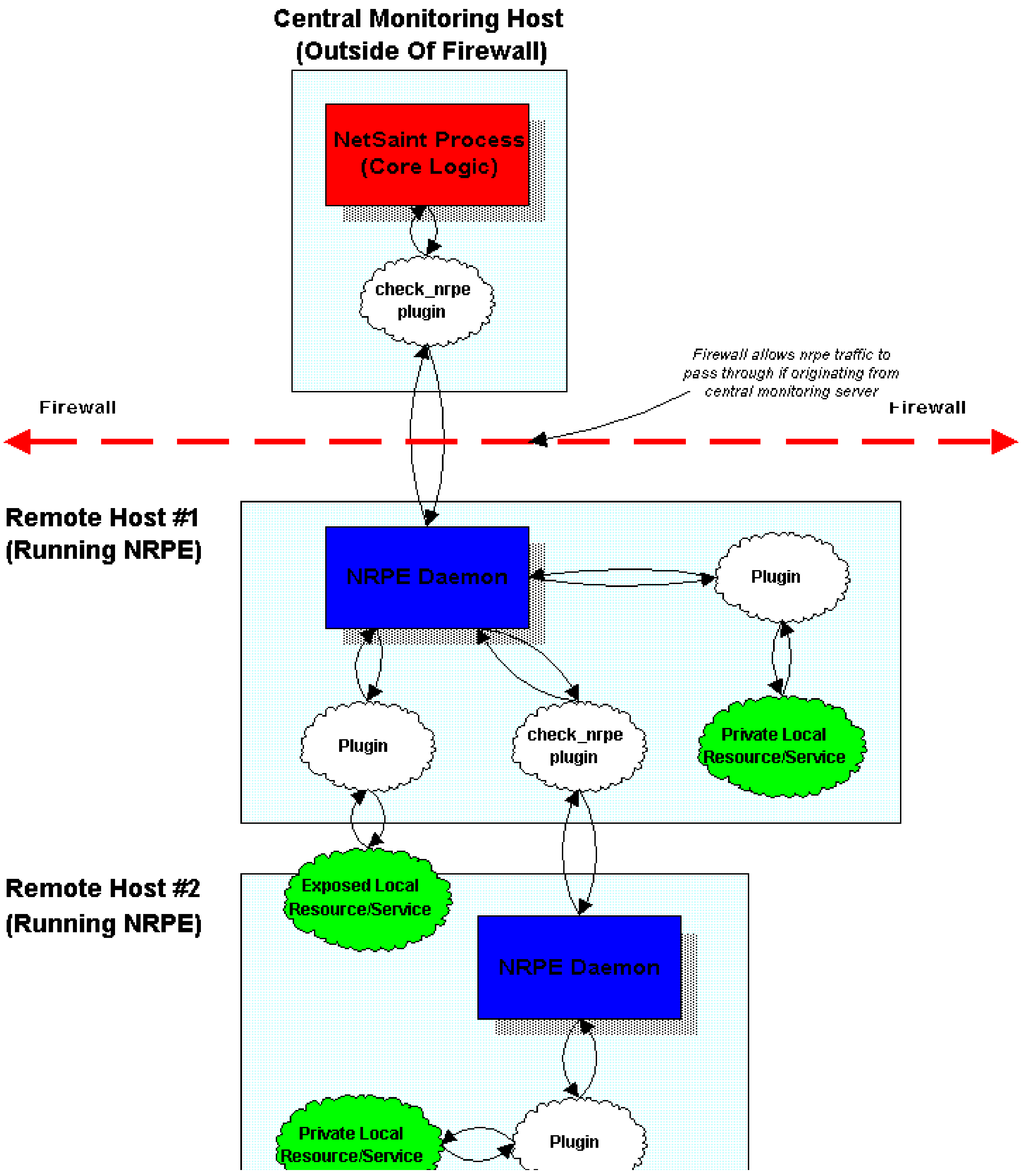
Indirect Service Checks

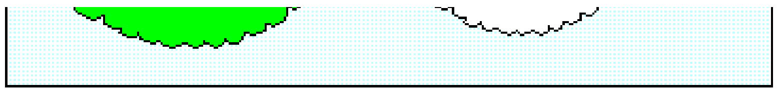
Last Updated: 04/21/2000


Central Monitoring Host
(Outside Of Firewall)
NetSaint 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

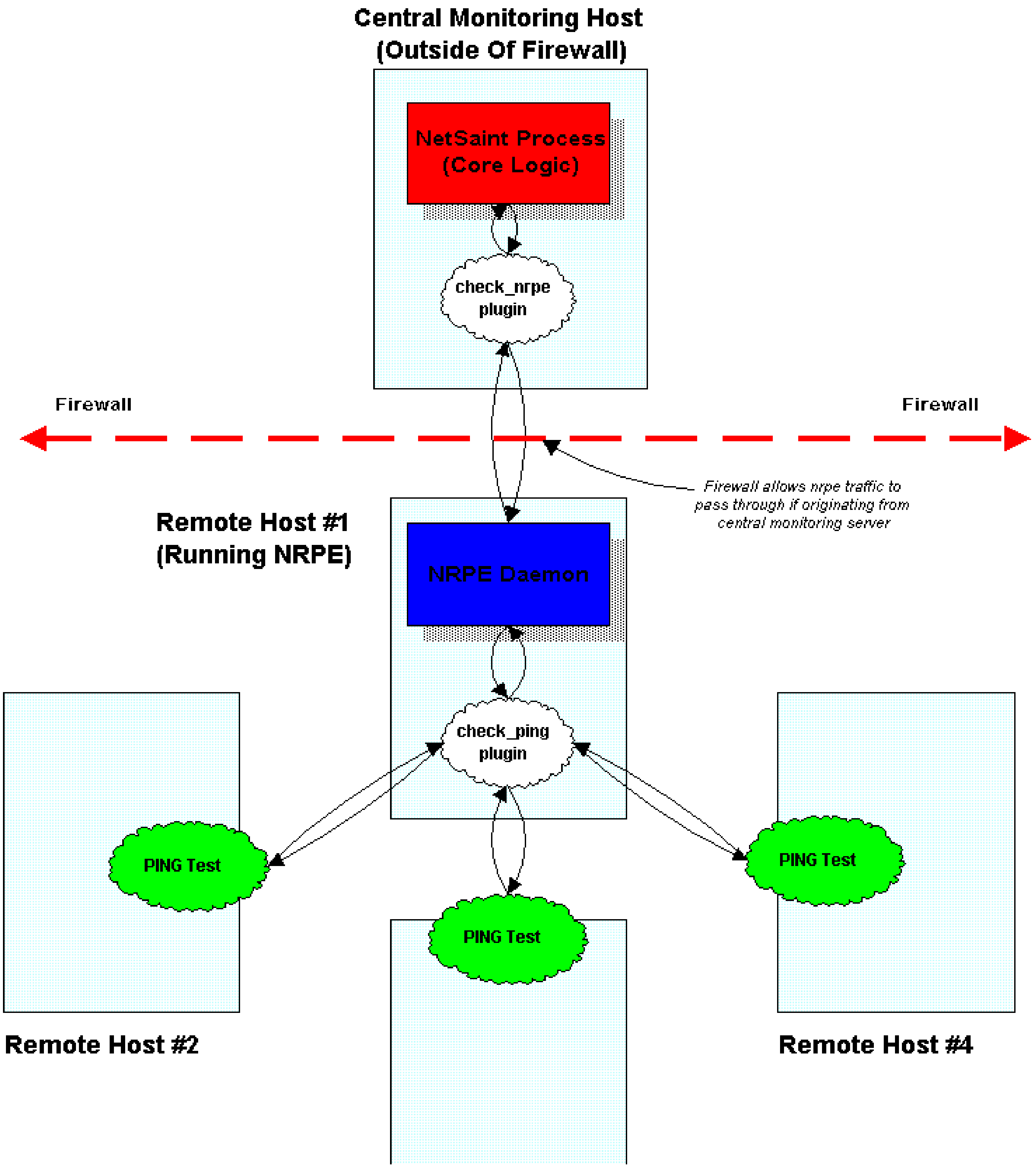
Last Updated: 04/21/2000

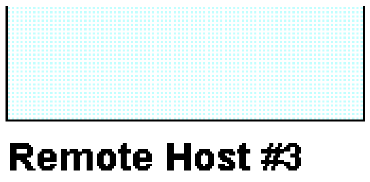




Indirect Host Checks

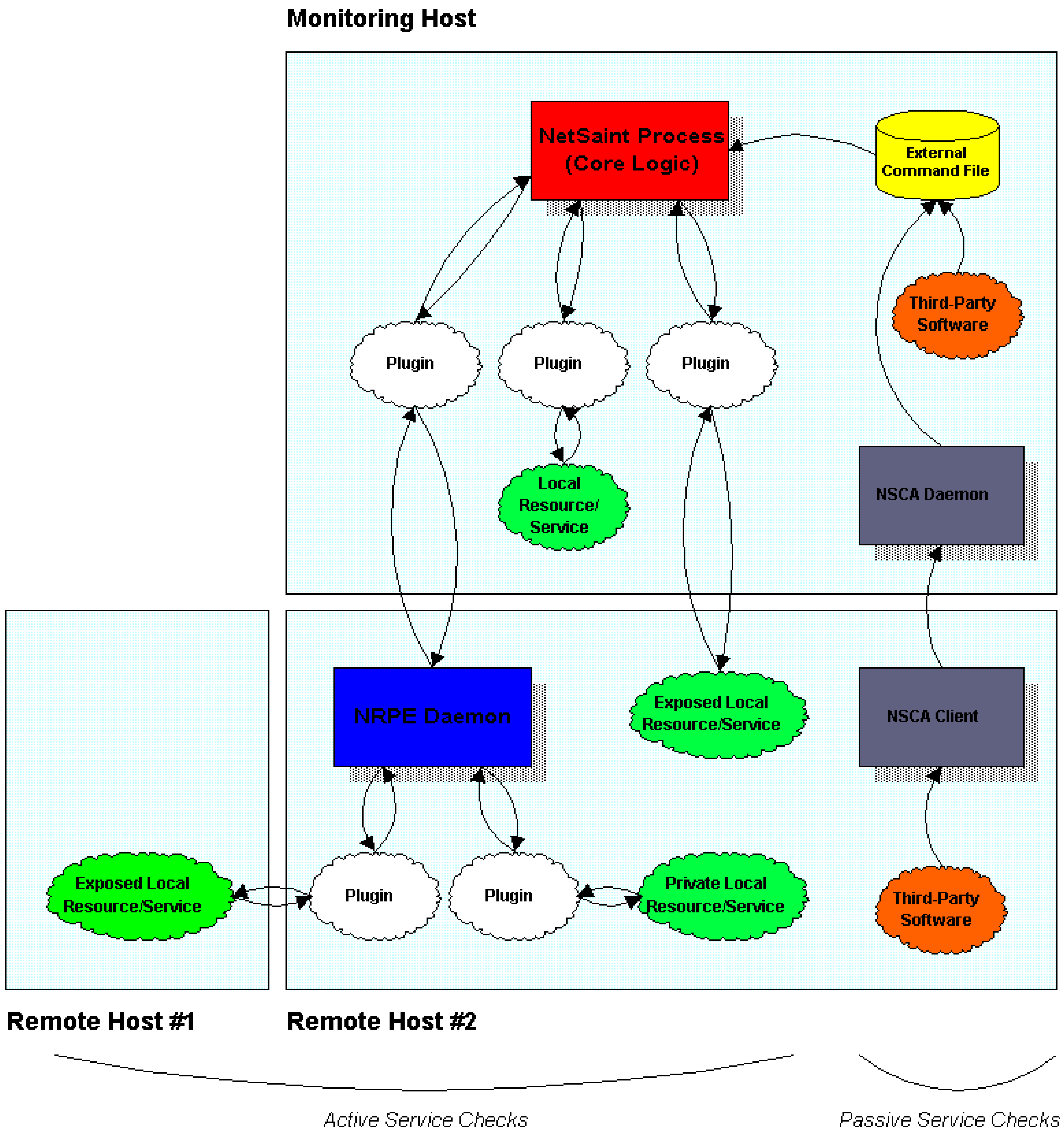
Last Updated: 04/18/2000

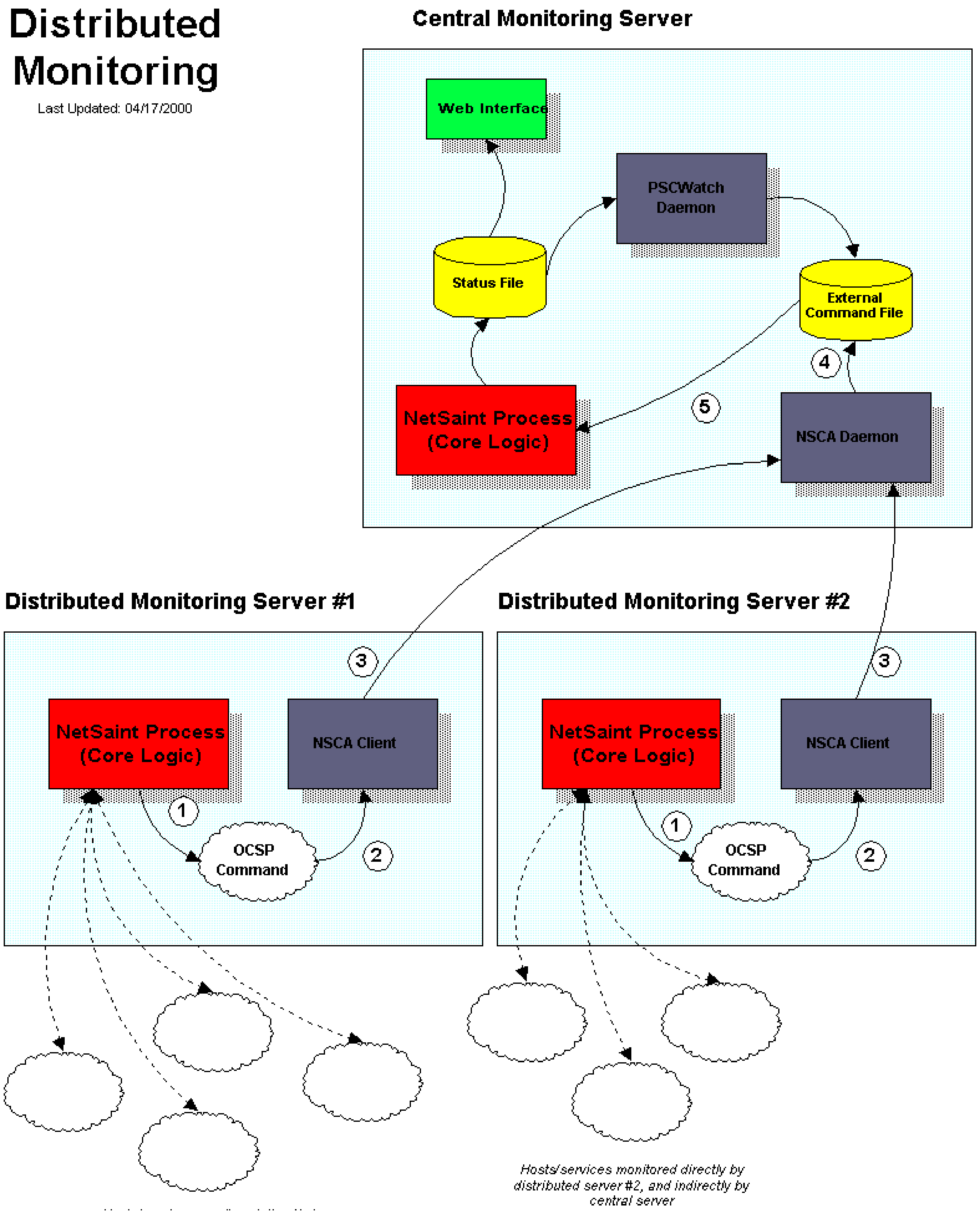


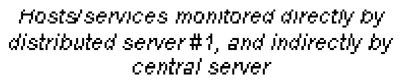

Remote Host #3

Using Active And Passive Checks Together

Last Updated: 04/18/2000




Distributed Monitoring
Last Updated: 04/17/2000
Central Monitoring Server
Web Interface
PSCWatch Daemon
Status File
External Command File
NetSaint Process (Core Logic)
NSCA Daemon
4
5
Distributed Monitoring Server #1
Distributed Monitoring Server #2
3
NetSaint Process (Core Logic)
NSCA Client
1
OCSP Command
2
3
NetSaint Process (Core Logic)
NSCA Client
1
OCSP Command
2
Hosts/services monitored directly by distributed server #2, and indirectly by central server


Hosts/services monitored directly by distributed server #1, and indirectly by central server

Status File Format

Introduction

In order to give external applications (such as the CGIs) access to the current host and service status information in NetSaint, all status information is saved to the file specified by the status_file option in the main config file. External applications can read the contents of this file to determine the current status of any monitored host or service. External applications *should not* write anything to the status file. NetSaint does not read the status file to determine current service and host information - it is simply provided as a means for third-party apps to access the internal status information in an easy manner.

File Format

The status file contains three types of entries: a program entry, one or more host status entries, and one or more service status entries. The format for each type of entry it described below.

Program Entry Format:

[<timestamp>] PROGRAM;<start_time>;<daemon_mode>;<program_mode>;<last_mode_change>;<last_command_check>;<last_log_rotation>;<executing_service_checks>;<accept_passive_service_checks>;<enable_event_handlers>;<obsess_over_services>

where...

- ***timestamp*** is the time in time_t format (seconds since UNIX epoch) that the program entry was last updated.
- ***start_time*** is the time in time_t format (seconds since UNIX epoch) that NetSaint was last (re)started.
- ***daemon_mode*** in an integer that indicates whether or not NetSaint is running as a daemon. If this value is 1, NetSaint is running in daemon mode. If this value is 0, NetSaint is running as a normal (foreground or background) process.
- ***program_mode*** a string which identifies what program mode NetSaint is currently in. If this string is "ACTIVE", NetSaint is in active mode. If this string is "STANDBY", NetSaint is in standby mode.
- ***last_mode_change*** is the time in time_t format (seconds since UNIX epoch) when the last program mode change occurred.
- ***last_command_check*** is the time in time_t format (seconds since UNIX epoch) that NetSaint last checked for external commands. A value of zero means that NetSaint has not checked for external commands since it was last (re)started.
- ***last_log_rotation*** is the time in time_t format (seconds since UNIX epoch) that NetSaint last rotated the main log file. A value of zero means that the log file has not been rotated since NetSaint was last (re)started.
- ***execute_service_checks*** in an integer that indicates whether or not NetSaint is actively executing service checks. Values: 0=checks are *not* being executed, 1=checks are being executed.
- ***accept_passive_service_checks*** in an integer that indicates whether or not NetSaint is accepting passive service checks. Values: 0=passive service checks are *not* being accepted, 1=passive checks are being accepted.
- ***enable_event_handlers*** in an integer that indicates whether or not host and service event handlers are enabled. Values: 0=event handlers are *not* enabled, 1=event handlers are enabled.
- ***obsess_over_services*** in an integer that indicates whether or not is running "obsessing" over service check results and running a obsessive service check processor command. Values: 0=Netsiant is *not* obsessing, 1=NetSaint is obsessing.

Host Status Format:

[<timestamp>] HOST; <host_name>;<state>;<last_state_change>;<problem_has_been_acknowledged>;<time_up>;<time_down>;<time_unreachable>;<last_notification>;<current_notification_number>;<notifications_enabled>;<event_handler_enabled>;<checks_enabled>;<plugin_output>

where...

- ***timestamp*** is the time in time_t format (seconds since UNIX epoch) that the host was last checked (or its current state was assumed).
- ***host_name*** is the short name of the host (as defined in the host configuration file) that the state information corresponds to.
- ***state*** is a string that indicates the current state of the host. Values include "PENDING", "UP", "DOWN", and "UNREACHABLE".
- ***last_state_change*** is the time in time_t format (seconds since UNIX epoch) that the host last experienced a hard state change.
- ***problem_has_been_acknowledged*** is an integer indicating whether or not this host problem has been acknowledged. If the host is UP, or it is DOWN or UNREACHABLE and has not been acknowledged, this is set to 0. If this host is DOWN or UNREACHABLE and the problem has been acknowledged, this is set to 1.
- ***time_up*** is the number of seconds (since monitoring began) that the host has been in an UP state.
- ***time_down*** is the number of seconds (since monitoring began) that the host has been in a DOWN state.
- ***time_unreachable*** is the number of seconds (since monitoring began) that the host has been in an UNREACHABLE state.
- ***last_notification*** is a timestamp in time_t format (number of seconds since UNIX epoch) that indicates when the last notification for this host was sent out. If no notifications have been sent out (or if the host is UP), this value is set to zero.
- ***current_notification_number*** is an integer representing the number of notifications that have been sent out about this host problem. If no notifications have been sent out since the host last changed state (of if it is in an UP state), this value is set to zero.
- ***notifications_enabled*** is an integer that indicates whether or not notifications for this host are enabled. Values: 0=notifications are *not* enabled, 1=notifications are enabled.
- ***event_handler_enabled*** is an integer that indicates whether or not the event handler for this host are enabled. Values: 0=event handler is *not* enabled, 1=event handler is enabled.
- ***checks_enabled*** is an integer that indicates whether or not checks this host are enabled. Values: 0=checks are *not* enabled, 1=checks are enabled.
- ***plugin_output*** is the output from the last host check (text)

Service Status Format:

**[<timestamp>] SERVICE;
<host_name>;<svc_description>;<state>;<current_attempt>/<max_attempts>;<state_type>;<next_check>;<check_type>;<checks_enabled>;<passive_checks_accepted>;<last_state_change>;<problem_has_been_acknowledged>;<last_hard_state>;<time_ok>;<time_unknown>;<time_warning>;<time_critical>;<last_notification>;<current_notification_number>;<notifications_enabled>;<check_latency>;<execution_time>;<plugin_output>**

where...

- ***timestamp*** is the time in time_t format (seconds since UNIX epoch) that the service was last checked.
- ***host_name*** is the short name of the host that this service is associated with.
- ***svc_description*** is the description of the service (as defined in the host configuration file) that the state information corresponds to. Together, the *host_name* and *svc_description* fields uniquely identify a service definition.
- ***state*** is string indicating the current state of the service. Values include "OK", "UNKNOWN", "WARNING", "CRITICAL", "RECOVERY", "UNREACHABLE", and "HOST DOWN". A value of "RECOVERY" indicates that the service is in an OK state, but just recovered from a non-OK state. Values of "UNREACHABLE" and "HOST DOWN" indicate that the host that the service is associated with is either down or unreachable.
- ***current_attempt*** is an integer representing the current service check attempt number. This value will be set to 1 if the host that the service is associated with is either down or unreachable.
- ***max_attempts*** is an integer representing the maximum number of check attempts for this service.
- ***state_type*** is a string indicating what type of state the service is currently in. Values include "SOFT" and "HARD".
- ***next_check*** is the time in time_t format (seconds since UNIX epoch) that the service is next scheduled to be checked.
- ***check_type*** is a string indicating what type of service check this was. Values include "ACTIVE" and "PASSIVE".
- ***checks_enabled*** is an integer representing whether or not checks for this service are enabled. Values: 0=checks are *not* enabled, 1=checks are enabled.
- ***accept_passive_checks*** is an integer representing whether or not passive checks are being accepted for this service. Values: 0=passive checks are *not* being accepted, 1=passive checks are being accepted.
- ***event_handler_enabled*** is an integer representing whether or not the event handler for this service is enabled. Values: 0=event handler is *not* enabled, 1=event handler is enabled.
- ***passive_checks_accepted*** is an integer representing whether or not passive checks are being accepted for this service. If this value is 1, they are being accepted. If this value is 0, passive checks are not being accepted.
- ***last_state_change*** is the time in time_t format (seconds since UNIX epoch) that the service last had a hard state change.
- ***problem_has_been_acknowledged*** is an integer indicating whether or not this service problem has been acknowledged. If the service is in an OK state, or it is in a non-OK state and has not been acknowledged, this is set to 0. If this service is in a non-0K state and the problem has been acknowledged, this is set to 1.
- ***last_hard_state*** is a string that indicates the last hard state of the service. Values include "OK", "UNKNOWN", "WARNING", and "CRITICAL".
- ***time_ok*** is the number of seconds that the service has been in an OK state.
- ***time_warning*** is the number of seconds that the service has been in a WARNING state.
- ***time_unknown*** is the number of seconds that the service has been in an UNKNOWN state.
- ***time_critical*** is the number of seconds that the service has been in a CRITICAL state.
- ***last_notification*** is a timestamp in time_t format (number of seconds since UNIX epoch) that indicates when the last notification for this service was sent out. If no notifications have been sent out or if the service is currently in an OK state, this value is set to zero.
- ***current_notification_number*** is an integer representing the number of notifications that have been sent out about this service problem. If no notifications have been sent out since the service last changed state (of if it is in an OK state), this value is set to zero.
- ***notifications_enabled*** is an integer that indicates whether or not notifications for this service have been enabled. Values: 0=notifications are *not* enabled, 1=notifications are enabled.
- ***check_latency*** is an integer indicating the number of seconds that the service check lagged behind its scheduled execution time (actual execution time - scheduled execution time = latency)
- ***execution_time*** is an integer indicating the number of seconds that this service check took to execute
- ***plugin_output* is the output from the last service check (text)**

Comment File Format

Introduction

In order to help share information among administrators, techs, etc., NetSaint allows comments to be added to all hosts and services that are being monitored. The comments are stored in the file specified by the comment_file directive in the main configuration file.

It should be noted that NetSaint "cleans" the comment file each time it restarts. During the cleaning process, NetSaint will remove all comments that are not marked as being persistent or that do not correspond to valid hosts or services that you have defined, and it will re-number all comment IDs.

Adding Comments

If you wish to use or write an external application that adds comments to hosts or services, you should *not* write comments directly to the comment file. Instead, use the ***ADD_SVC_COMMENT*** and ***ADD_HOST_COMMENT*** external commands. The commands should be written to the external command file. NetSaint will periodically scan the external command file and process any commands it finds in there.

Deleting Comments

Similiarly, if you want to delete one or more comments from the command file, use the ***DEL_SVC_COMMENT***, ***DEL_HOST_COMMENT***, ***DEL_ALL_SVC_COMMENTS***, or ***DEL_ALL_HOST_COMMENTS*** external commands. Do *not* modify the contents of the comment file yourself!

File Format

The comment file contains two types of entries: host comments and service comments. The format for each type of comment it described below.

Host Comment Format:

[<timestamp>] HOST_COMMENT;<id>;<host_name>;<persistent>;<author>;<comment>

where...

- *timestamp* is the time in time_t format (seconds since UNIX epoch) that the comment was entered by the user.
- *id* is a comment identification number which is unique among other host and service comments. This number is generated by NetSaint and cannot be specified by the user.
- *host_name* is the short name of the host (as defined in the host configuration file) that the comment is associated with.
- *persistent* is a flag which indicated whether the comment is persistent or not. Persistent comments survive program restarts, while non-persistent comments are deleted when NetSaint is restarted. A value of 0 indicates that the comment is non-persistent, while a value of 1 indicates that it is

persistent.

- *author* is a text field that contains the name of the person who entered the comment.
- *comment* is a text field that contains the actual comment.

Service Comment Format:

[<timestamp>] SERVICE_COMMENT;<id>;<host_name>;<svc_description>;<persistent>;<author>;<comment>

where...

- *timestamp* is the time in time_t format (seconds since UNIX epoch) that the comment was entered by the user.
- *id* is a comment identification number which is unique among other host and service comments. This number is generated by NetSaint and cannot be specified by the user.
- *host_name* is the short name of the host that the service is associated with.
- *svc_description*is the description of the service (as defined in the host configuration file) that the comment is associated with. Together the *host_name* and *svc_description* uniquely identiry a particular service.
- *persistent* is a flag which indicated whether the comment is persistent or not. Persistent comments survive program restarts, while non-persistent comments are deleted when NetSaint is restarted. A value of 0 indicates that the comment is non-persistent, while a value of 1 indicates that it is persistent.
- *author* is a text field that contains the name of the person who entered the comment.
- *comment* is a text field that contains the actual comment.

State Retention File Format

Introduction

In order to preserve host and service state information (current status, state time statistics, etc.) between program restarts, users can opt to enable the state retention feature by using the retain_state_information option in the main config file. If this option is enabled, state retention information is stored in the file specified by the state_retention_file directive in the main configuration file. Immediately before shutting down (or restarting) NetSaint will write all current host and service state information to the retention file. Upong restarting, NetSaint will read the information stored in the retention file, initialize host and service information, and delete the file.

At any time while NetSaint is running, you can have it save service and host state information, by using the *SAVE_STATE_INFORMATION* external command. You can also force NetSaint to read in previously save state information by using the *READ_STATE_INFORMATION* command, although this is not recommend, as the current state information that NetSaint has will be replaced with whatever is stored in the state retention file.

It should be noted that NetSaint will only save state information for service and hosts that have been checked at the time the file is written. Also, NetSaint will only save the last hard state for the host or service.

File Format

The state retention file contains four types of entries: a creation timestamp, program state information, host state information and service state information. The format for each type of entry it described below.

Creation Time Format:

CREATED: <timestamp>

where...

- *timestamp* is the time in time_t format (seconds since UNIX epoch) that the state information was saved.

Program Information Format:

PROGRAM: <program_mode>;<execute_service_checks>;<accept_passive_service_checks>;<enable_event_handlers>;<obsess_over_services>

where...

- *program_mode* is an integer that represents the last program mode that NetSaint was in. Values: 0=standby mode, 1=active mode.
- *execute_service_checks* is an integer indicating whether or not service checks were being executed when NetSaint was running. Values: 0=checks were *not* being executed, 1=checks were being executed.
- *accept_passive_service_checks* is an integer indicating whether or not passive service checks were being accepted when NetSaint was running. Values: 0=passive checks were *not* being accepted, 1=passive checks were being accepted.
- *enable_event_handlers* is an integer indicating whether or not host and service event handlers were enabled when NetSaint was running. Values: 0=event handlers were *not* enabled, 1=event handlers were enabled.
- *obsess_over_services* is an integer indicating whether or not NetSaint was obsessing over service checks when it was running. Values: 0=NetSaint was *not* obsessing, 1=NetSaint was obsessing.

Host Information Format:

HOST: <host_name>;<state>;<last_check>;<checks_enabled>;<time_up>;<time_down>;<time_unreachable>;<last_notification>;<current_notification_number>;<current_notification_number>;<notifications_enabled>;<event_handler_enabled>;<problem_has_been_acknowledged>;<plugin_output>

where...

- *host_name* is the short name of the host (as defined in the host configuration file) that the state information corresponds to.
- *state* is an integer corresponding to the state of the host (UP, DOWN, or UNREACHABLE). See the *base/netsaint.h* file for the integer values of different states.
- *last_check* is a timestamp in time_t format (number of seconds since UNIX epoch) that indicates when the host status was last checked.
- *checks_enabled* is an integer indicating whether or not checks of this host have been enabled. Values: 0=checks have been disabled, 1=checks are enabled.
- *time_up* is the number of seconds that the host has been in an UP state.
- *time_down* is the number of seconds that the host has been in a DOWN state.
- *time_unreachable* is the number of seconds that the host has been in an UNREACHABLE state.
- *last_notification* is a timestamp in time_t format (number of seconds since UNIX epoch) that indicates when the last notification for this host was sent out. If no notifications have been sent out, this value is set to zero.
- *current_notification_number* is an integer representing the number of notifications that have been sent out about this host problem. If no notifications have been sent out since the host last changed state (of if it is in an UP state), this value is set to zero.
- *notifications_enabled* is an integer that indicates whether or not notifications for this host have been enabled. Values: 0=notifications have been disabled, 1=notifications are enabled.
- *event_handler_enabled* is an integer indicating whether or not the event handler for this host has been enabled. Value: 0=event handler has been disabled, 1=event handler is enabled.
- *problem_has_been_acknowledged* is an integer indicating whether or not this host problem has been acknowledged. If the host is UP, or it is DOWN or UNREACHABLE and has not been acknowledged, this is set to 0. If this host is DOWN or UNREACHABLE and the problem has been acknowledged, this is set to 1.
- *plugin_output* is the output from the last host check (text)

Service Information Format:

SERVICE: <host_name>;<svc_description>;<state>;<last_check>;<time_ok>;<time_warning>;<time_unknown>;<time_critical>;<last_notification>;<current_notification_number>;<notifications_enabled>;<checks_enabled>;<accept_passive_checks>;<event_handler_enabled>;<problem_has_been_acknowledged>;<plugin_output>

where...

- *host_name* is the short name of the host that this service is associated with.
- *svc_description* is the description of the service (as defined in the host configuration file) that the state information corresponds to. Together, the *host_name* and *svc_description* fields uniquely identify a service definition.
- *state* is an integer corresponding to the state of the state (OK, WARNING, UNKNOWN, or CRITICAL). See the *base/netsaint.h* file for the exact values of different states.
- *last_check* is a timestamp in time_t format (number of seconds since UNIX epoch) that indicates when the service status was last checked.
- *time_ok* is the number of seconds that the service has been in an OK state.
- *time_warning* is the number of seconds that the service has been in a WARNING state.
- *time_unknown* is the number of seconds that the service has been in an UNKNOWN state.
- *time_critical* is the number of seconds that the service has been in a CRITICAL state.
- *last_notification* is a timestamp in time_t format (number of seconds since UNIX epoch) that indicates when the last notification for this service was sent out. If no notifications have been sent out, this value is set to zero.
- *current_notification_number* is an integer representing the number of notifications that have been sent out about this host problem. If no notifications have been sent out since the host last changed state (of if it is in an UP state), this value is set to zero.
- *notifications_enabled* is an integer that indicates whether or not notifications for this service have been enabled. Values: 0=notifications have been disabled, 1=notifications are enabled.
- *checks_enabled* is an integer that indicates whether or not checks of this service have been enabled. Values: 0=checks have been disabled, 1=checks are enabled.
- *accept_passive_checks* is an integer representing whether or not passive checks are being accepted for this service. If this value is 1, they are being accepted. If this value is 0, passive checks are not being accepted.
- *event_handler_enabled* is an integer indicating whether or not the event handler for this service has been enabled. Value: 0=event handler has been disabled, 1=event handler is enabled.
- *problem_has_been_acknowledged* is an integer indicating whether or not this service problem has been acknowledged. If the service is in an OK state, or it is in a non-OK state and has not been acknowledged, this is set to 0. If this service is in a non-0K state and the problem has been acknowledged, this is set to 1.
- *plugin_output* is the output from the last service check (text)

Fun Stuff

Have a little too much free time on your hands? Well, instead of playing Quake , you could try out some of the following things that you can do with NetSaint...

Create a virtual network assistant that speaks!

The Lowdown

By utilizing event handlers and some speech software, you can have NetSaint talk to you and tell you whats wrong with your network.

Completely Scientific Ratings

Funness Rating:	100%
Ability To Impress Co-Workers Rating:	100%
Usefulness Rating:	30%
Wise Use Of System Resources Rating:	5%

The Upsides

- It gives immediate audio feedback on the status of your network, which is quite useful if you're in the server room working on other things
- It will impress your boss and co-workers...

The Downsides

- Its a bit of a waste of system resources, so its not really fit for production machines
- If you're in the server room alone on a weekend or at night, having a machine start talking can scare the living daylights out of you. It has happened to me before...

Give Me Details!

First off, you need speech software installed on your system. I would recommend using the Festival Speech Synthesis System developed by The Centre for Speech Technology Research at the University of Edinburgh. This package provides the basic framework needed for converting text into spoken word. In order to use Featival in a practical manner you'll also need to install speechd. The speechd software implements /dev/speech and will queue all text written to it for processing by the Festival sound system. Once you've installed the speech software and *tested* it to make sure it works properly, you can move on to the next step...

In order to make NetSaint report system status via the speech software you'll have to write some event handlers. If you want audio alerts for only certain hosts or services, you'll have to define event handlers in the appropriate host and service definitions. If you want audio alerts for everything, you can just use the global_host_event_handler and global_service_event_handler definitions in the main configuration file. I've chosen to use global event handlers to make things easier to implement.

The global event handler definitions in my main configuration file look like this...

```
global_host_event_handler=global-hst-event-handler

global_service_event_handler=global-svc-event-handler
```

The command definitions for my global event handlers look like this...

```
command[global-hst-event-handler]=/usr/local/netsaint/libexec/eventhandlers/hst_event_handler $HOST NAME$
"$HOSTALIAS$" $HOSTSTATE$ $STATETYPE$ $HOSTATTEMPT$

command[global-svc-event-handler]=/usr/local/netsaint/libexec/eventhandlers/svc_event_handler $HOSTNAME$
"$HOSTALIAS$" "$SERVICEDESC$" $SERVICESTATE$ $STATETYPE$ $SERVICEATTEMPT$
```

So what do the event handler scripts look like? The event handlers I use are listed below. A copy of these scripts is available in the *eventhandlers/* subdirectory of the distribution. You may have to modify things to work on your system...

Global Host Event Handler (hst_event_handler)

```
#!/bin/sh

#########################################
# NetSaint Global Host Event Handler
#
# Arguments:
#
# $1 = host short name
# $2 = host alias (long name)
# $3 = state
# $4 = state type (HARD or SOFT)
# $5 = current attempt number
#
#########################################

echocmd="/bin/echo"
festivalcmd="/dev/speech"

case $4 in
    HARD)
        case $3 in
            UP)
                $echocmd "Good news!  Host $2 has RECOVERED!" > $festivalcmd
                ;;
            DOWN)
                $echocmd "Attention...  Host $2 is $3.  This is a critical state. Please check host
status!" > $festivalcmd
                ;;
            UNREACHABLE)
                $echocmd "Attention...  Host $2 is $3.  This is a critical state.  Please check
network connectivity!" > $festivalcmd
                ;;
        esac
        ;;
    SOFT)
        case $3 in
            UP)
                ;;
            DOWN)
                $echocmd "Attention... Host $2 is in a $4 $3 state.  Attempt number $5" >
$festivalcmd
                ;;
            UNREACHABLE)
                $echocmd "Attention... Host $2 is in a $4 $3 state.  Attempt number $5" >
$festivalcmd
                ;;
        esac
        ;;
esac

exit 0
```

Global Service Event Handler (svc_event_handler)

```
#!/bin/sh

#########################################
# NetSaint Global Service Event Handler
#
# Arguments:
#
# $1 = host short name
# $2 = host alias (long name)
# $3 = service description
# $4 = state
# $5 = state type (HARD or SOFT)
# $6 = current attempt number
#
#########################################

echocmd="/bin/echo"
festivalcmd="/dev/speech"

case $5 in
    HARD)
        case $4 in
            OK)
                $echocmd "Good news!  Service $3 on $2 has RECOVERED!" > $festivalcmd
                ;;
            CRITICAL)
                $echocmd "Attention...  Service $3 on $2 is in a $4 state.  Please check service!" 
> $festivalcmd
                ;;
            WARNING)
                $echocmd "Attention...  Service $3 on $2 is in a $4 state.  Please check service!" 
> $festivalcmd
                ;;
            UNKNOWN)
                $echocmd "Attention...  Service $3 on $2 is in a $4 state.  Please check service!" 
> $festivalcmd
                ;;
        esac
        ;;
    SOFT)
        case $4 in
            OK)
                ;;
            CRITICAL)
                $echocmd "Attention.  Service $3 on $2 is in a $5 $4 state." > $festivalcmd
                ;;
            WARNING)
                $echocmd "Attention.  Service $3 on $2 is in a $5 $4 state." > $festivalcmd
                ;;
            UNKNOWN)
                $echocmd "Attention.  Service $3 on $2 is in a $5 $4 state." > $festivalcmd
                ;;
        esac
        ;;
esac

exit 0
```

That's it! Fire up NetSaint and listen to it tell you about your problems...