CHAPTER 8

Distributed Monitoring, Redundancy, and Failover

One of the important components of any enterprise monitoring application or tool is its ability to function in a disaster or high-availability situation. This is especially important for an enterprise management tool that is monitoring your hosts and services and warning you when issues arise or administrator intervention is required. If your enterprise-monitoring solution fails, you will have no warning or notifications of other problems in your environment. Often this includes being unable to provide you with warning or notification of the disaster or outage that has taken out both your enterprise-monitoring system and your production environment.

Also important for an enterprise-monitoring tool is the ability to function in a distributed model that allows the monitoring of assets in remote locations. This is especially true where monitoring of these assets is not possible from a central location due to issues with network visibility or network controls such as intervening firewalls. In this instance the ability to deploy distributed servers and send back the results of this monitoring to a centralized management server is important. Nagios has the ability to do this kind of distributed monitoring and to operate in redundant and failover modes. I'll address how to achieve both in this chapter.

Distributed Monitoring

Distributed monitoring allows you to offload the monitoring of your hosts and services onto multiple systems. This is designed to overcome two obstacles. First, it allows you to overcome performance limitations if you have numerous hosts and services and allows you to spread the monitoring load over multiple servers. Second, it allows you to monitor hosts and services in remote or segmented parts of your environment, such as in a remote geographical location or behind a firewall.

In a nondistributed environment, the Nagios server needs to be able to connect to all monitored hosts and services using the plug-in or mechanism you are using to monitor them and return the results of the check. For example, if you are monitoring a website the Nagios server needs to be able to make an HTTP connection to the website and receive the response. If there is a firewall between the Nagios server and the website that blocks this HTTP traffic, it is not possible to check this service.

To overcome this, Nagios uses another server, a distributed server, on the other side of the firewall to perform the check and then sends the results back to a central Nagios server. The distributed server uses service obsession and a special tool called NSCA (Nagios Service Check

Acceptor) to send these results to the central server.[1] Thus, you only need to open one hole in your firewall for the Nagios check results rather than all the possible protocol types required to monitor a collection of hosts and services. The central server receives these check results as passive checks and updates the status of the hosts and services based on this.[2]

As I mentioned, to facilitate the sending of these results there is a tool called NSCA, developed by the author of Nagios. This tool has two components: a plug-in that sends check results and a daemon that runs on your central Nagios server. The daemon accepts the transmitted results and submits them as passive check results using external commands to the Nagios server. This means you only need to have the NSCA traffic between your central and distributed servers. This limits both the bandwidth used and the number of ports you might need to open on your firewall or firewalls.

So how does this work in more detail? Well, you can see a distributed monitoring model in Figure 8-1.

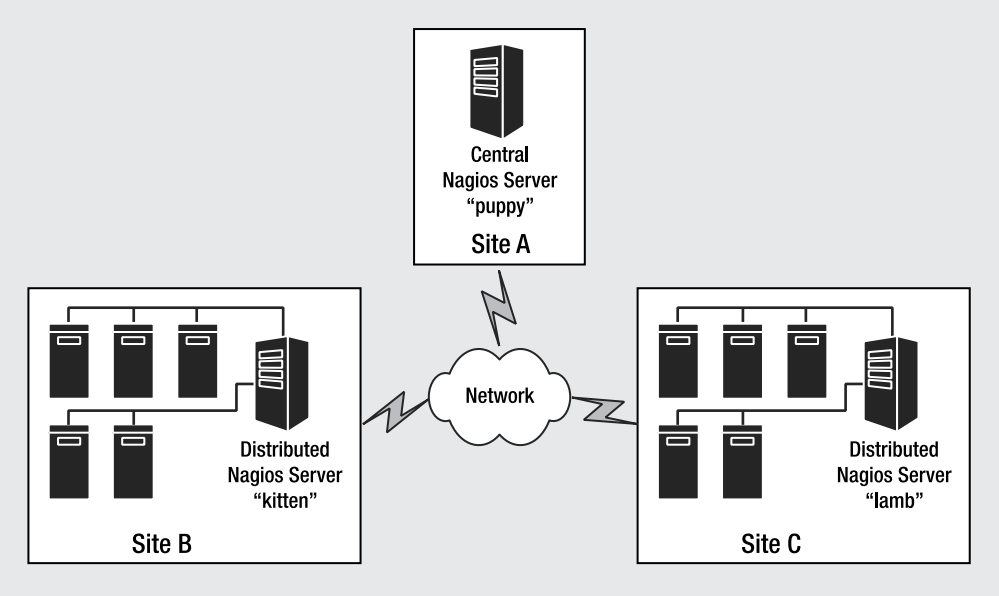

Figure 8-1. *Distributed monitoring model*

First, let's look at the architecture of our distributed environment. The major pieces of the solution are the central server and the distributed servers. In Figure 8-1 you can see our central server, `puppy`, located in Site A, and two distributed servers, `kitten` and `lamb`, located in Sites B and C. LAN or WAN connections would link these sites and potentially might have firewalls installed between them. If you had firewalls installed, traffic from the NSCA tool on the distributed servers containing the check results would need to be able to pass through these

1. I discussed service obsession in Chapter 2.
2. See Chapter 2 for a discussion of passive service checks.

firewalls. By default, the NSCA tool uses TCP port 5667 for this traffic. I'll cover installing and configuring NSCA to both send and receive check results later in this chapter.

■ **Note** The sites in Figure 8-1 could also represent segmented sections of an internal network. The same design principles apply to both models.

The central server, `puppy`, located in Site A in Figure 8-1, is the collation point for all your check results. It generally would not perform any active checks itself. Or potentially it may perform checks for hosts and services local to it, for example, in the same site or network segment. As it is the collation point, the central server must contain object definitions for all hosts and services in your environment. This includes all hosts and services defined on your distributed servers, in this case, the `kitten` and `lamb` servers. If you do not add a host or service being monitored by one of your distributed servers to your central server, when the central server receives the check result it will discard the result since it has no knowledge of that host or service.

The distributed servers, `kitten` and `lamb`, perform the actual checks on your hosts and services. They then use the NSCA tool to send the results to the `puppy` server, where they are processed as passive check results and the status of the host or service updated. Thus, the distributed servers only need to have the object definitions for those hosts and services that you require them to monitor.

Your distributed servers are usually bare-bones installations of Nagios that only contains the Nagios server and any plug-ins required to perform checks. You should not need to install the web console. You also do not need to configure any notifications. The central server will perform all the notifications required based on the results it receives from the distributed servers.

Distributed Server Configuration

Let's start by configuring our distributed servers. As I have already mentioned, these can be a bare-bones installation of Nagios and you do not generally need to install the web console and hence a web server on these servers. So first install Nagios on the server. You can follow the instructions in Chapter 1 for this. Then define all the hosts and services that this server will be required to monitor.[3] Remember, you only need to define those hosts and services that this specific distributed server is going to monitor.

You also do not need to configure any notifications. Notifications will be performed by the central server. So you should set the `enable_notifications` directive in the `nagios.cfg` configuration file on the distributed server to `0` like so:

```
enable_notifications=0
```

---

3. See Chapters 1 and 2 for further information on how to do this.

■ **Caution** If you have state retention enabled (using the `retain_state_information` and `use_retained_program_state` directives) as I discussed in Chapter 2, Nagios will ignore how this directive is set in the `nagios.cfg` configuration file in favor of what is contained in the state retention file. So if you have program state retention enabled on your distributed server and at some point enable notifications, remember that this setting will be kept through program restarts and the setting in the `nagios.cfg` file will be ignored. This can lead to confusion as to why your distributed server is sending notifications.

Installing NSCA

Next, we need to install the NSCA tool to allow the distributed server to send the check results to the central server. The NSCA package does have one prerequisite, the `libmcrypt` library. This library enables you to use encryption when sending results between your distributed and central servers. This is a very important security feature, and I recommend you install the library. You can find the `libmcrypt` library at `http://mcrypt.sourceforge.net/` or you can download it as an RPM from either `www.ottolander.nl/opensource/mcrypt/mcrypt.html` or `http://dag.wieers.com/packages/libmcrypt/`. The installation process for the `libmcrypt` library is very simple. Download the source package (I have chosen the Australian Sourceforge mirror but you should choose the mirror near you). Then configure, make, and install it like so:

```
kitten# wget http://optusnet.dl.sourceforge.net/sourceforge/mcrypt/➥
libmcrypt-2.5.7.tar.gz
kitten# tar -zxf libmcrypt-2.5.7.tar,gz
kitten# cd libmcrypt-2.5.7
kitten# ./configure
kitten# make
kitten# make install
```

Once you have `libmcrypt` installed, you can start the NSCA installation process. The NSCA tool is available from the Nagios Exchange website at `www.nagiosexchange.org/Communication.41.0.html?&tx_netnagext_pi1[p_view]=140` or via Sourceforge at `http://prdownloads.sourceforge.net/nagios/`. At the time of this writing the latest version of NSCA was 2.4.

Download and unpack the package on your distributed server like so:

```
kitten# wget http://optusnet.dl.sourceforge.net/sourceforge/nagios/nsca-2.4.tar.gz
kitten# tar -zxf nsca-2.4.tar.gz
```

Next change into the NSCA source package directory and configure the package using the `configure` script. There are some configuration options you can see if you run the `configure` script with the `--help` option (see Table 8-1), but generally you will not need these for compiling the send tool on the distributed server.

Table 8-1. *NSCA configure Options*

Option	Default	Description
`--with-nsca-user=`*user*	`nagios`	Specifies the username to run the NSCA daemon as
`--with-nsca-grp=`*group*	`nagios`	Specifies the group name to run the NSCA daemon as
`--with-nsca-port=`*port*	`5667`	Specifies the TCP port number that the NSCA tool uses

The first two options specify which user to run the NSCA daemon as; I recommend you use the same user and group that the Nagios server runs as. Both default to a setting of `nagios`. The third option controls which port NSCA will use to send and receive results. This defaults to TCP port 5667, and I recommend you leave this as the default unless your environment requires you define another port number. You cannot specify a privileged port number, that is, a port number below 1024.

You can then configure NSCA and run the `make` option to compile it:

```
kitten# ./configure
kitten# make all
```

■ **Tip** You can also find an RPM of the NCSA tool at `http://dries.studentenweb.org/rpm/packages/nagios-nsca/info.html`.

When you want to install the NSCA tool, you need to manually install the required tool and configuration files. The compile process creates two binaries, `nsca` and `send_nsca`, located in the `src` directory beneath the NSCA package directory. The `nsca` binary is the NSCA daemon that runs on the central server and receives the check results. The `send_nsca` binary does the actual sending of the check results and is executed on the distributed server. There are also two configuration files, `nsca.cfg` and `send_nsca.cfg`, located in the root of the NSCA package directory. They are for the NSCA daemon and sending program, respectively. For the distributed server, you only need the `send_nsca` and `send_nsca.cfg` files. I recommend you install the `send_nsca` binary to the `bin` directory in your default Nagios directory structure, as I'm going to define it in a command shortly. This directory is `/usr/local/nagios/bin` by default. I suggest you place the `send_nsca.cfg` file into your Nagios `etc` directory, by default `/usr/local/nagios/etc`.

Configuring send_nsca

Once you have installed NSCA you need to configure it, both in terms of configuring it to the Nagios server and configuring the `send_nsca` program itself. Let's start with the `send_nsca` program. Its configuration is held in the `send_nsca.cfg` configuration file we placed in the `/usr/local/nagios/etc/` directory earlier. Example 8-1 shows a sample configuration.

Example 8-1. *send_nsca.cfg configuration File*

```
password=password
encryption_method=3
```

Tip You can add comments to the `send_nsca.cfg` configuration file by prefixing the line with a # symbol.

There are only two options in the `send_nsca.cfg` configuration file. Both of these options contain values that must be identical to their equivalent options in the `nsca.cfg` configuration file that will be located on the central server. The first option, `password`, specifies the password you will use to encrypt any data between the distributed and central servers. Therefore, it must be the same on both servers. You should replace the value *password* with a password of your choice. I recommend you chose a nondictionary word that includes special characters and is at least eight characters long.

Caution I don't want people to read this option, so I'm going to secure this file to prevent casual users from seeing the password.

The next option, `encryption_method`, specifies what form of encryption you will use to encrypt the transmission of your check results. A variety of methods are available, ranging from no encryption, which I strongly recommend you don't use, to 3DES and Blowfish. Table 8-2 contains a list of the values you can use for this option and the type of encryption they represent.

Table 8-2. *NSCA Encryption Methods*

Value	Type of Encryption
0	None
1	Simple XOR (obfuscates but does not encrypt)
2	DES
3	Triple DES
4	CAST-128
5	CAST-256
6	xTEA
7	3WAY
8	Blowfish
9	Twofish
10	LOKI97
11	RC2
12	ARCFOUR
14	RIJNDAEL-128
15	RIJNDAEL-192
16	RIJNDAEL-256
19	WAKE

Value	Type of Encryption
20	SERPENT
22	Enigma (Crypt)
23	GOST
24	SAFER64
25	SAFER128
26	SAFER+

This is not a book about encryption, and I will not make any judgments about which encryption method is best. Obviously, the more complicated the encryption method, the greater overhead placed on your servers to encrypt and decrypt the transmissions. Generally speaking, though, the volume of data is quite small and most hosts will absorb the overhead required for more complex encryption. Additionally, it depends on the level of risk you perceive about the data being sent. I'll discuss this in more detail when I explain the security architecture and model of the NSCA daemon in the "Central Server Configuration" section.

Personally I use Triple DES (option `3`). It is secure and is present on most distributions and systems. To be able to communicate with the distributed servers, the central server must use the same encryption method; I'll also show you how to do that when I look at configuring that server in the "Central Server Configuration" section.

Once you've configured the `send_nsca.cfg` file, you need to secure that file's ownership and permissions, as shown in Example 8-2.

Example 8-2. *Securing the send_nsca.cfg File*

```
kitten# chown nagios:nagios /usr/local/nagios/etc/send_nsca.cfg
kitten# chmod 0640 /usr/local/nagios/etc/send_nsca.cfg
```

In Example 8-2, I've changed the ownership of the configuration file to the `nagios` user and group (which I've used for the Nagios server on the distributed servers). I've also changed the permissions of the file to `0640` to only allow the `nagios` user to read the file.

■ **Note** I'll look at the exact functioning of the `send_nsca` binary later in this section when I define how to send the check results to the central server.

Now we've configured the NSCA sending program, we need to define this program to Nagios and configure the server to send its check results to the central server. To do this, let's use host and service obsession. Host and service obsession allow you to specify commands that will run after each host or service check. In this case, the commands will send the check results to the central server to allow the hosts and services status to be updated.

To do this you need to turn on two directives in the `nagios.cfg` file, specify a command to send service results and a command to send host results, and define those commands to some additional directives in the configuration file. The two directives you need to turn on are

obsess_over_services and obsess_over_hosts, which turn on obsession for services and hosts, respectively:

```
obsess_over_services=1
obsess_over_hosts=1
```

Next, you need to define the commands that will be executed when a check is completed. These are defined in the ocsp_command and ochp_command directives; also in the nagios.cfg configuration file:

```
ocsp_command=send_service_check
ochp_command=send_host_check
```

■ **Note** These directives may not be present in your nagios.cfg configuration file if you have used the sample configuration that comes with Nagios, and you may need to add them.

Sending Service Check Results

Now, the commands defined in these directives need to be defined to Nagios. We do this by adding them as command object definitions. You will need to create two commands: one for host check results and one for service check results. Let's first look at the service check results command, which is defined in the ocsp_command directive in Example 8-3.

Example 8-3. *ocsp_command directive Command Definition*

```
define command{
             command_name     send_service_check
             command_line     /usr/local/nagios/libexec/send_service_check ➥
$HOSTNAME$ '$SERVICEDESC$' $SERVICESTATEID$ '$SERVICEOUTPUT$'
             }
```

Let's break down the command defined in Example 8-3. I've defined a command called send_service_check (which is the same name I used in the ocsp_command directive in the nagios.cfg configuration file). The command executes a shell script (which I'll show you next) also called send_service_check. I'm passing a number of macro values to the shell script, which will in turn pass these to the send_nsca program. These are the hostname that the service is running on represented by the $HOSTNAME$ macro. Next is the description of the service, which is defined in the service_description directive in the service definition object, and represented here by the $SERVICEDESC$ macro. Next comes the $SERVICESTATEID$ macro, which contains the state of the service as returned by the service check. In the case of services, these are OK, WARNING, CRITICAL, and UNKNOWN. But rather than being displayed by name, the service states in the $SERVICESTATEID$ macro are represented by the return code values listed in Table 8-3.

Table 8-3. *Numeric Representation of Service States*

Value	State
0	OK
1	WARNNG
2	CRITICAL
3	UNKNOWN

You should use these numeric return codes because the `send_nsca` program requires the return code rather than the name of the status when submitting check results. I'll demonstrate this in the shell script I've created to submit the results that you can see in Example 8-5.

Finally, I've specified the `$SERVICEOUTPUT$` macro. This macro contains the output of the service check result.

■ **Note** You will see I've placed two macros, `$SERVICEDESC$` and `$SERVICEOUTPUT$`, in quotes. This is to ensure that if they contain multiword data that it is passed to the shell script cleanly.

Now that we have our command definition, we need to write the `send_service_check` shell script to use the `send_nsca` program to submit the results to the central server. To do this, you must understand how the `send_nsca` program works. The program looks very similar to a standard Nagios plug-in; Example 8-4 demonstrates a sample of how it might be run from the command line.

Example 8-4. *The send_nsca Program*

```
kitten# ./send_nsca -H 10.0.0.1 -c /usr/local/nagios/etc/send_nsca.cfg ➥
owlet local_disk 2 'Connection refused by host'
```

Example 8-4 executes the `send_nsca` program from the command line. It has a number of options. The first is the `-H` option, which specifies the IP address or hostname of the central server the check result is being sent to. In this case, it is `10.0.0.1`. The next option, `-c`, specifies the location of the `send_nsca.cfg` configuration file. Finally, I've specified the actual check results: the hostname of the host the service is running on, the service description, the return code, and the output of the service check. By default each of these result values needs to be separated by a tab character.

There are also some additional options available for the `send_nsca` program, as shown in Table 8-4.

Table 8-4. *send_nsca Options*

Option	Description
`-p` *port*	The TCP port number on the central server where the NSCA daemon is running. Defaults to port 5667.
`-to` *seconds*	Connection timeout. Defaults to 10 seconds.
`-d` *delimiter*	The delimiter character for the check results. Defaults to the tab character.

The first option, `-p`, in Table 8-4 allows you to override the default port of 5667 on the central server. The second option, `-to`, lets you specify a different value for a connection timeout to the central server. The default is 10 seconds. The last option, `-d`, allows you to specify the delimiter that will be used between each item of the check results. By default, this is a tab character. You can override this with another character such as `,` or a `;` symbol like so: `-d ","` or `-d ";"`.

Now that you know how the `send_nsca` program works, let's use it in the shell script that will send the check results to the central server. Example 8-5 shows a sample shell script that will perform this function.

Example 8-5. *send_service_check Script*

```
#!/bin/sh

# Arguments:
# $1 = Hostname of the host (using the $HOSTNAME$ macro)
# $2 = Service Description of the service (using the $SERVICEDESC$ macro)
# $3 = Service Status ID of the service (using the $SERVICESTATUSID$ macro)
# $4 = Output of the service check (using the $SERVICEOUTPUT$ macro)

/bin/echo "$1","$2","$3","$4" | /usr/local/nagios/bin/send_nsca -H ip_address ➥
-c /usr/local/nagios/etc/send_nsca.cfg -d ","
```

In Example 8-5, you can see a very simple shell script that is designed to echo the check results, each separated by a `,` symbol (for which I've overridden the default delimiter symbol using the option `-d ","`), to the `send_nsca` program and from there to the central server. You would need to replace the *ip_address* value with the IP address or hostname of the central Nagios server that is running the NSCA daemon.

AN ALTERNATIVE TO THE SEND_NSCA SHELL SCRIPT

We don't precisely need the shell script. We could also just reference the `send_nsca` program in the command object definition and pass the macros directly to it, but I prefer using a shell script as it allows me to potentially add a data cleansing or manipulation stage to the sending process. An example of this alternative object definition might look like this:

```
define command{
              command_name                          send_service_check
              command_line /usr/local/nagios/bin/send_nsca -H ip_address ➥
-c /usr/local/nagios/etc/send_nsca.cfg -d "," $HOSTNAME$,'$SERVICEDESC$',➥
$SERVICESTATEID$,'$SERVICEOUTPUT$'
              }
```

You would need to replace the *ip_address* value with the IP address or hostname of your central Nagios server.

Sending Host Check Results

Now in order to send host check results to the central server, we'll need to create another command, in our case the `send_host_check` command, as defined in the `ochp_command` directive in the `nagios.cfg` configuration file. I've shown a sample of this command object definition in Example 8-6.

Example 8-6. *ochp_command Directive Command Definition*

```
define command{
             command_name     send_host_check
             command_line     /usr/local/nagios/libexec/send_host_check ➥
$HOSTNAME$ $HOSTSTATEID$ '$HOSTOUTPUT$'
             }
```

■ **Note** You will notice the `send_host_check` command is very similar to the `send_service_check` command except that I've used different macros, which I'll explain later.

Let's break down the command defined in Example 8-6. I've defined a command called `send_host_check` (which is the same name I used in the `ochp_command` directive in the `nagios.cfg` configuration file). The command executes a shell script also called `send_host_check`. I'm passing a number of macro values to the shell script, which will in turn pass these to the `send_nsca` program. These are the hostname of the host whose check results I'm sending; this is represented by the `$HOSTNAME$` macro. Next comes the `$HOSTSTATEID$` macro, which contains the state of the host as returned by the host check. In the case of services, these are `UP`, `DOWN`, and `UNREACHABLE`. But rather than being displayed by name, the status types in the `$HOSTSTATEID$` macro are represented by the return code values, as you can see in Table 8-5.

Table 8-5. *Numeric Representation of Host States*

Value	State
0	UP
1	DOWN
2	UNREACHABLE

You need to use these numeric return codes because the `send_nsca` program requires the return code rather than the name of the status when submitting check results.

Finally, I've specified the `$HOSTOUTPUT$` macro. This macro contains the output of the host check result.

■ **Note** You will see that I've placed the `$HOSTOUTPUT$` macro in quotes. This is to ensure that if it contains multiword data it will be passed to the shell script cleanly.

Now that we have our command definition, we need to write the `send_host_check` shell script to use the `send_nsca` program to submit the results to the central server. It is again very similar to the `send_service_check` shell script, as you can see in Example 8-7.

Example 8-7. *send_host_check Script*

```
#!/bin/sh

# Arguments:
# $1 = Hostname of the host (using the $HOSTNAME$ macro)
# $2 = Host Status ID of the host (using the $HOSTSTATUSID$ macro)
# $3 = Output of the host check (using the $HOSTOUTPUT$ macro)

/bin/echo "$1","$2","$3" | /usr/local/nagios/bin/send_nsca -H ip_address ➥
-c /usr/local/nagios/etc/send_nsca.cfg -d ","
```

In Example 8-7 you can see a very simple shell script that is designed to echo the check results, each separated by a `,` symbol (for which I've overridden the default delimiter symbol using the option `-d ","`), to the `send_nsca` program and hence from there to the central server. You would need to replace the *`ip_address`* value with the IP address or hostname of the central Nagios server that is running the NSCA daemon.

Distributed Servers Final Steps

Once you've defined the required directives in the `nagios.cfg` file, configured NSCA, and created the required commands and shell scripts, you've completed the configuration of the distributed server to send the results of services and hosts to the central server. Let's quickly walk through the process of sending a service check to the central server to ensure we understand what is happening.

1. The distributed server executes a service check (or a host check).
2. After the check is completed, the command defined in either the `ocsp_command` or `ochp_command` directive executes (depending on whether it is a service or host check, respectively).
3. The command pipes the results of the check results to the `send_nsca` program.
4. The `send_nsca` command sends the check results to the central server.

Now that we have stepped through the events on the distributed Nagios server, let's see what happens on the central server and how it is configured.

Central Server Configuration

The central Nagios server should be configured much like a stand-alone Nagios server as it performs the majority of the same functions. Unlike with the distributed server, I recommend you install the web console. You will also have to ensure that all hosts and services defined on your distributed servers are also defined on your central server. The host and service definitions need to be essentially identical to those on the distributed server or servers.

Unlike with the distributed server, you will also need to enable notifications so that the central server can send any notifications generated. This is done by ensuring the `enable_notifications` directive in the `nagios.cfg` file is set to `1`:

```
enable_notifications=1
```

There are also several other options you need to have enabled on your central server. First, your central server needs to be allowed to receive passive checks from both hosts and services. This means the `accept_passive_service_checks` and `accept_passive_host_checks` directives in the `nagios.cfg` file both need to be set to `1`:

```
accept_passive_service_checks=1
accept_passive_host_checks=1
```

Second, the `check_external_commands` directive must be set to `1` to tell the Nagios server to check the external command file for commands to be processed:

```
check_external_commands=1
```

If this is not on, the submitted check results will not be processed.

Third, and last, you need to determine how your central server will handle active checks of hosts and services. There are a few scenarios here to consider. In the first instance, your central server may only be processing check results from distributed servers and not be performing any checks of hosts or services local to it. In this case, you should turn off all active host and service checking using the `execute_service_checks` and `execute_host_checks` directives in the `nagios.cfg` file. Setting these both to `0` will stop the central server from executing service checks:

```
execute_service_checks=0
execute_host_checks=0
```

In the second instance, your central servers may also be checking some local hosts and services. In this case, you should have the `execute_service_checks` and `execute_host_checks` directives set to `1` to enable active checks for these local hosts and services. To stop the central server from checking hosts and services that are being handled by the distributed server or servers, you need to set the `active_checks_enabled` directive in the object definition of each of the hosts or services being checked via distributed servers to `0`. This will ensure that the central server will not perform active checks of them.[4] Instead, the central server will rely on the submitted passive check results to maintain the status of these hosts and services.

■ **Caution** Using passive checks alone can be problematic, and I'll discuss this and some potential solutions in the upcoming section, "Distributed Monitoring and Freshness."

---

4. It will still schedule checks for them but will not execute those checks.

Installing the NSCA Daemon

You will also need to install the NSCA daemon. Follow the instructions in the "Installing NSCA" section, including adding the `libmcrypt` prerequisite, to do this. This same process will produce the required `nsca` binary and `nsca.cfg` configuration file. The `nsca` binary is the NSCA daemon that runs on the central server and receives the check results. The `nsca.cfg` configuration file controls the NSCA daemon. For the central server we need both files. I recommend you install the `nsca` binary to the `bin` directory in your default Nagios directory structure. This directory is `/usr/local/nagios/bin` by default. I suggest you place the `nsca.cfg` file in your Nagios `etc` directory, by default, `/usr/local/nagios/etc`.

■ **Tip** When you compile the NSCA package, I recommend you specify the same user and group name as your Nagios server for the daemon to run as. This is because the daemon is required to write to your external command file, and this is generally the easiest way to provide this access.

Configuring the NSCA Daemon

Once you have installed NSCA daemon, you need to configure it. The NSCA daemon works by listening for check results being sent via the `send_nsca` command on a remote server. It then submits these check results to the external command file to be processed by the Nagios server. Its configuration is held in the `nsca.cfg` configuration file you placed in the `/usr/local/nagios/etc/` directory. I've shown a sample configuration in Example 8-8.

Example 8-8. *nsca.cfg Configuration File*

```
server_port=5667
server_address=10.0.0.1
allowed_hosts=127.0.0.1,10.0.0.10,10.0.0.20,10.0.0.30
nsca_user=nagios
nsca_group=nagios
debug=0
command_file=/usr/local/nagios/var/rw/nagios.cmd
alternate_dump_file=/usr/local/nagios/var/rw/nsca.dump
aggregate_writes=0
append_to_file=0
max_packet_age=30
password=password
decryption_method=3
```

There are a number of options in the `nsca.cfg` configuration file. I've listed them all in Table 8-6 and will explain their function in more detail next.

Table 8-6. *nsca.cfg Options*

Option	Description
`server_port=port`	Specifies the TCP port the `nsca` daemon should run on. Defaults to 5667.
`server_address=ip_address`	Contains the IP address the `nsca` daemon should bind to. Defaults to all addresses.
`allowed_hosts=hosts`	Contains IP addresses of the hosts that are allowed to connect to the central server.
`nsca_user=user`	Specifies the name of the user the `nsca` daemon should run as.
`nsca_group=group`	Specifies the name of the group the `nsca` daemon should run as.
`debug=0 | 1`	Turns on debug function.
`command_file=file`	Specifies the location of the external command file.
`alternate_dump_file=file`	Specifies the location of an alternative command file.
`aggregate_writes=0 | 1`	Turns on aggregate writes.
`append_to_file=0 | 1`	Specifies whether to open the command file for writing or appending.
`max_packet_age=age`	Specifies the maximum packet age in seconds.
`password=password`	Contains the password used between central and distributed servers.
`decryption_method=method`	Specifies the decryption method used.

■ **Tip** You can add comments to the `nsca.cfg` configuration file by prefixing the line with a # symbol.

The first two options in Table 8-6, `server_port` and `server_address`, specify the port number and IP address that the daemon will bind to. This defaults to all addresses on the host on TCP port 5667.

■ **Tip** For the `server_port` value, you cannot specify a privileged port (i.e., a port number lower than 1024).

The next option, `allowed_hosts`, requires that you specify the IP addresses of the distributed servers that are allowed to connect to this central server. The daemon only performs minor checking of these source addresses and thus these can be easily spoofed. The developer of Nagios recommends running the daemon under `inetd` or `xinetd` (and instructions on how to do this are provided in the `README` file in the NSCA source package). Personally I prefer not to use either and instead use a host-based firewall to lock down the daemon to the IP addresses of the distributed servers. Thus, I specify the allowed hosts in the `allowed_hosts`

option and add iptables rules only allowing traffic from the required distributed servers to connect to the daemon on the central server via port 5667. A typical rule is shown here:

```
puppy# iptables -A INPUT -p tcp --dport 5667 -s 10.0.0.10 -j ACCEPT
```

The iptables rule on this line would only allow incoming connections to TCP port 5667 from IP address 10.0.0.10. You could specify additional rules for each distributed server.

The next two options, nsca_user and nsca_group, control the user and group that the daemon will run as. I recommend using the same user and group that the Nagios server runs as to ensure the daemon is able to write to the external command file.

The debug option allows you to turn on some debugging for the daemon. This results in debug information being outputted to syslog. I suggest when you're first testing the daemon you turn this on, but it is not required during general operation. Setting it to 1 enables debugging and 0 disables it.

The command_file and alternative_dump_file options specify the location of your external command file and an alternative destination for check results if the command file is unavailable. The command_file option would normally default to /usr/local/nagios/var/rw/nagios.cmd. As the command file is a named pipe, it only exists while the Nagios daemon is running. Therefore, if Nagios is down and check results are received, they are lost unless an alternate destination is specified. You can specify this alternate destination with the alternative_dump_file option. The results of the checks will be written into this file in the form of external commands. Many people check for this file as part of the Nagios init script or start-up process and dump the contents of the file into the external command file and then purge the file. This ensures any checks potentially received while Nagios was down are still processed. Alternatively, you could ignore checks received while the Nagios server is down by not specifying this option.

The aggregate_writes option specifies that if incoming connections contain multiple check results, such as a batch dump of results from a distributed server, you want to enable aggregate writing to the command file. Generally speaking, unless you are batching results you should leave this option as 0.

The append_to_file option specifies whether the daemon should open the command file for writing or appending. Unless you have a specific need (and I generally have never seen one), this should remain set to open for writing by specifying its value as 0.

Using the max_packet_age option is an additional method of ensuring the security of your incoming check results. This option should be set to the longest time period in seconds that your distributed servers are likely to take to send check results in. Any check results received in a longer time period will be discarded as a potential "replay" attack.[5] You cannot specify a value longer than 900 seconds, or 15 minutes. Ensure you set a high enough value for this age to accommodate any network latency issues you might have. This is especially important for check results being sent over the Internet where there might be delays.

Both of the password and decryption_method options contain values that must be identical to their equivalent options, the password and encryption_method options, in the send_nsca.cfg configuration file that is located on the distributed server.

The first option, password, specifies the password you will use to encrypt and decrypt any data between the distributed and central servers. Hence, it must be the same on both servers.

5. See http://en.wikipedia.org/wiki/Replay_attack.

I recommend you chose a nondictionary word that includes special characters and is at least eight characters long.

■ **Caution** I don't want people to view the password contained in this option, so I'm going to secure this file to prevent casual users from seeing the password.

The next option, `decryption_method`, must match the value specified in the `encryption_method` option specified on the distributed servers, in our case, `3`, which represents Triple DES encryption.

So why encrypt your check results? Well, in the case of the NSCA package, encryption provides both authentication and transmission security for the check results. The first function, authentication, enhances the daemon's ability to ensure that the source of the data is a trusted system. I've already recommended using `iptables` rules in conjunction with the `allowed_hosts` option (or you may prefer `inetd` or `xinetd`). If the password and encryption method on both the distributed and central servers are identical, the daemon assumes that the check result is valid and accepts it. This prevents an attacker from either maliciously submitting false check data or, worse, submitting malicious external commands via the daemon that could adversely impact your Nagios server or the host it is running on. The second function, transmission security, ensures that no one can sniff out your check results and use any data in them to provide some advantage when attacking your organization.

This is a very simplified explanation of how the NSCA package uses encryption and how it attempts to ensure the security and authenticity of connections. For a complete explanation, see the `SECURITY` file in the NSCA source package.

■ **Caution** If you are transmitting check results via the Internet, this is especially important. Do not transmit your results over an untrusted network without a level of authentication and encryption that you feel comfortable with.

Lastly, once you've configured the `nsca.cfg` file you need to secure that file's ownership and permissions. Example 8-9 shows how.

Example 8-9. *Securing the nsca.cfg File*

```
kitten# chown nagios:nagios /usr/local/nagios/etc/nsca.cfg
kitten# chmod 0640 /usr/local/nagios/etc/nsca.cfg
```

Notice that I've changed the ownership of the configuration file to the `nagios` user and group (which I've used to run the Nagios server process on the central server). I've also changed the permissions of the file to `0640` to only allow the `nagios` user to read the file.

Starting the NSCA Daemon

Finally, once you've configured NSCA, you need to start the daemon. The daemon is controlled by the `nsca` binary. The binary has only two options: the `-c` option that specifies the location of the `nsca.cfg` configuration file and the mode you wish to run it in. The possible modes are `--inetd`, `--daemon`, and `--single`. The `--inetd` mode is used if you run the NSCA daemon from within `inetd` or `xinted`. The `--daemon` and `--single` modes run the NSCA daemon as a standalone daemon. The `--daemon` mode runs it as a multiprocess daemon, which I recommend for servers that have a heavy workload. The `--single` mode runs the daemon as a single process, which is better suited to a low-volume environment. If you do not specify a mode, the NSCA daemon defaults to the `--single` mode. Here's an example of how to start the daemon:

```
puppy# /usr/local/nagios/bin/nsca -c /usr/local/nagios/etc/nsca.cfg --single
```

■ **Tip** Also in the NSCA package is an `init` script you can use to configure the NSCA daemon to start automatically when your host starts up. I recommend you modify this to suit your environment and use it to start and stop the NSCA daemon.

Distributed Monitoring and Freshness

There is one remaining major issue we need to consider with distributed monitoring. This issue concerns the veracity of the data being gathered from the distributed servers. The distributed monitoring solution relies on passive check results to perform its monitoring. I discussed passive check results in Chapter 2 and explained that this means Nagios simply has to assume that the source of these check results is accurate and up-to-date. It has no way of validating, as it normally would with active checks, that the incoming data represents the true state of the hosts and services being monitored.

There is, however, a way to enhance the value of these checks so you can be a bit more confident that they represent the true state of your assets. You do this using the concept of "freshness" checking.[6] Freshness checks monitor the age of the received check results, and if they exceed a set threshold, Nagios will trigger an active check of the device. This active check is conducted even if active checks are disabled for the host, service, or the entire server.

So how do you configure this? Well, let's have a quick refresher. In your `nagios.cfg` configuration file are four relevant directives that must be set. I've shown them in Example 8-10.

Example 8-10. *Freshness Directives in nagios.cfg*

```
check_service_freshness=1
service_freshness_check_interval=60
check_host_freshness=1
host_freshness_check_interval=60
```

---

6. I also discussed this concept in Chapter 2.

The check_service_freshness and check_host_freshness need to be set to 1 if you wish to enable service and host freshness checking for the server. The service_freshness_check_interval and host_freshness_check_interval specify how often in seconds the Nagios server will check the freshness of check results.

In our host and service definitions are three relevant directives: check_freshness, freshness_threshold, and check_command. The first directive, check_freshness, needs to be set to 1 for all hosts and services for which freshness checking is enabled. The freshness_threshold specifies the time period in seconds for which a check result is fresh. Any longer than this time period, and the check result will be declared stale. The freshness check will then be executed. This takes the form of the Nagios server executing the command contained in the check_command directive. On the following lines, I've specified an example service that uses freshness checking:

```
define service{
              host_name                       owlet
              service_description             http
              active_checks_enabled           0
              passive_checks_enabled          1
              check_freshness                 1
              freshness_threshold             300
              check_command                   check_http
              …
              }
```

In the previous service, the http service on the owlet host will not be actively checked. Instead, Nagios will rely on passive service check results to update the status of the service. Freshness checking is also enabled, and if the last passive check results are older than 300 seconds (5 minutes), Nagios will execute an active service check using the command check_http.

This model is fine if your central server is able to actually see the distributed hosts and services in your environment. But what if, as is common, one of the reasons you are using distributed monitoring is that you don't have network visibility of the host or service being checked? This means that, if you execute the check_http command and the Nagios server can't see the service, the check will time out. This response is of little value to us.

So what can you do to get a notification that there is potentially a problem with this service? Well, instead of using a check command that requires visibility of the host or service, configure a command that returns a status code and an error message just like a normal plug-in. Let's look at an example. In the following lines, I've defined a command called check_stale:

```
define command{
              command_name              check_stale
              command_line              $USER1$/check_dummy $ARG1$ $ARG2$
              }
```

The check_stale command uses a special plug-in called check_dummy, which comes with the Nagios plug-in package. The check_dummy plug-in returns a status and some optional text based on what is submitted to the $ARG1$ and $ARG2$ macros. On the next line, I've defined a potential check_command directive that would use this check_stale command:

```
define service{
            ….
            check_command         check_stale!2!'This service is stale'
            …
            }
```

The check_stale command has two arguments, 2 and the text 'This service is stale'. The 2 represents the return code for the status we wish the check_dummy plug-in to return (I discussed these numeric representations of service status in Table 8-3). The text 'This service is stale' represents the output we want from the plug-in. The check_dummy plug-in then returns the status of the service, based on us submitting the value 2, as CRITICAL and as the output of the plug-in as the text 'This service is stale'. As the check result is CRITICAL, the service would change into the CRITICAL status and generally, if configured, a notification would be generated. The service is then in a notification cycle until the problem is acknowledged or a more recent passive check result is received that changes the status of the service to an OK status.

■ **Tip** You could also write a script to produce the same results. You'll find an example of this way of handling this situation in the Nagios documentation at http://nagios.sourceforge.net/docs/2_0/freshness.html.

Central Servers Final Steps

Once we've defined the required directives in the nagios.cfg configuration file, configured the NSCA daemon, and duplicated the services and hosts defined on the distributed server (potentially editing them to configure how active checks and freshness are handled), we've completed the configuration of the central server. The central server should now be configured to receive the results of services and hosts from the distributed server or servers. Let's quickly walk through the process of sending and receiving a service check to ensure we understand what is happening:

1. The distributed server executes a service check (or a host check).
2. After the check is completed, the command defined in either the ocsp_command or ochp_command directive executes (depending on whether it is a service or host check, respectively).
3. The command pipes the results of the check results to the send_nsca program.
4. The send_nsca command sends the check results to the central server.
5. The NSCA daemon on the central server receives the check results and verifies the password and encryption method are valid. If they are not, the result is discarded.
6. The check result is submitted to the external command file.
7. The check result is processed and the status of the host or service is updated.

Your distributed monitoring configuration is now complete and you should be able to begin monitoring your hosts and services.

KEEPING YOUR CONFIGURATION SYNCRONIZED

In all the models I've described in this chapter, you generally need to keep your object definitions either exactly, or in the case of the distributed model, approximately synchronized between multiple Nagios servers. There are a number of ways to do this. I recommend you look at several potential tools. These range from standard Unix applications such as `rsync` to file replication tools such as Unison (`www.cis.upenn.edu/~bcpierce/unison/`) or cfengine (`www.cfengine.org/`).

I recommend that the two key attributes of any solution you chose be speed and security. This is especially true if you need to conduct file-synchronization activities over an Internet connection. Focus on a solution that will quickly and accurately replicate the required configuration files. A solution may potentially be required to only partially replicate a file or make more selective changes in the style of `diff` or `sdiff` where differences exist in the directives defined between central and distributed servers or master and slave servers. You should also consider a solution that allows you to tunnel or encrypt your file replication. This could consist of a scripted replication using a tool like `sftp` or another form of replication encapsulated in an SSL tunnel such as a Stunnel or HTTPS tunnel. Some of the available tools like Unison can be tunneled via `ssh`.

Redundancy and Failover

With Nagios there are two recommended methods for providing redundancy and failover. The first method is simple but, as a result of that simplicity, only protects against a limited type of failures. In this first method, you have two Nagios servers: one a master and one a slave. Both servers actively monitor the same hosts and services, but only the master server sends out notifications for any events on your hosts and services. In the event that master server is down or the Nagios process on the master server fails, the slave server takes over sending notifications. This is called redundant monitoring.

There are issues with this method, though. Principally, it doubles the bandwidth and performance requirements for your monitoring by duplicating your check processes across two servers. This is a rather inelegant way to provide this redundancy and resilience.

To address this issue, another method of implementation is possible. In this method, there is also a master and a slave server. The master server actively monitors all the hosts and services. The slave server contains an identical set of host and services definitions to the master server. But, unlike the redundant solution, the slave server does not actively monitor the hosts and services. Indeed, active checks and notifications are disabled on the slave server. Instead, using the NRPE tool, the slave server monitors the status of the Nagios process on the master server. If the slave server detects that the master server Nagios process has failed, it takes over the monitoring. Active checks and notifications are enabled using external commands, and the slave server takes over from the master server.

Of course, this means that the slave server does not have the current status of all the hosts and services being monitored. To overcome this, you must also configure host and service obsession on the master server and use the NSCA tool to send the check results to the slave server. This method of implementation is called failover monitoring.

You can see a high-level diagram of a failover monitoring model in Figure 8-2.

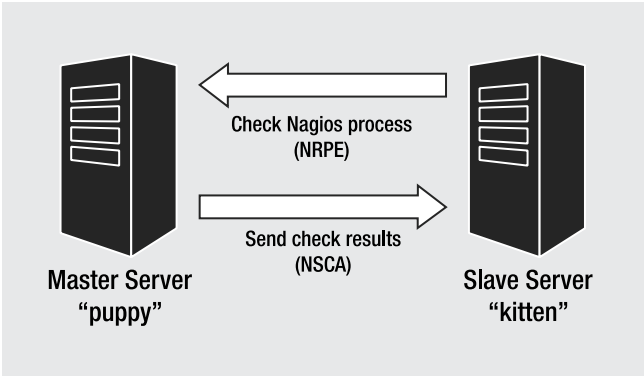

Figure 8-2. *Failover monitoring model*

This second method of implementation is far more elegant and allows considerably more flexible and less resource-intensive failover between Nagios servers. Therefore, I'll only cover this method of failover in this chapter.

■ **Note** If you are interested in implementing the first method of redundancy, many of the same steps described in this section are also required. You can see more by reading the Nagios documentation at `http://nagios.sourceforge.net/docs/2_0/redundancy.html`.

Configuring the Master Server

So where do we start? Well, first we need to configure our master server. I set up a normal Nagios server, generally including the web console. I configured all the required hosts and services that I wish to monitor. I also enabled active checks of all these hosts and services and configured any required notifications.

This process involves turning on active host and service checking using the `execute_service_checks` and `execute_host_checks` directives in the `nagios.cfg` configuration file. Setting these both to `1` will ensure the master server executes service checks:

```
execute_service_checks=1
execute_host_checks=1
```

You should also enable notifications using the `enable_notifications` directive like so:

```
enable_notifications=1
```

Configuring the NRPE Daemon

Next you need to configure the `nrpe` daemon on the master server to allow the slave server to check the status of the Nagios process.[7] Let's quickly step through installing the `nrpe` daemon on the master server. Download the `nrpe` daemon and unpack it like so:

```
puppy# wget http://prdownloads.sourceforge.net/nagios/nrpe-2.0.tar.gz
puppy# tar -zxf nrpe-2.0.tar.gz
puppy# cd nrpe-2.0
```

To run the `nrpe` daemon on the master, you need a user and group. I recommend you use the same user and group that the Nagios process runs as, in our example the user and group `nagios`. Otherwise, you can create a user and group for the `nrpe` daemon.

We can run the `configure` script for the `nrpe` daemon like so:

```
puppy# ./configure --enable-ssl
```

Using the `--enable-ssl configure` option, we configured native SSL/TSL support. This will be used to encrypt and secure the connections between the master and slave servers. You will need to have OpenSSL installed for this functionality to be enabled.[8] I strongly recommend for security reasons you enable this functionality. If you do not enable it, an intruder could either eavesdrop or subvert the connections between your Nagios server and the remote host.

Next I recommend you use a host firewall and lock down the source and destination of all traffic on port 5666 (or the port you intend to use for NRPE). Let's look at a quick example using `iptables`:

```
puppy# iptables -A INPUT -p tcp -m tcp --dport 5666 -s 10.0.0.15 -j ACCEPT
puppy# iptables -A OUTPUT -p tcp -m tcp --sport 5666 -d 10.0.0.15 -j ACCEPT
```

On these lines I've locked down the NRPE traffic entering and leaving the remote host on port 5666 to only that traffic from and destined to host `10.0.0.15`. This means that the slave server at `10.0.0.15` is the only host on our network that can send the `puppy` host NRPE traffic. You can obviously vary this to suit your environment.

Once you have configured the NRPE package, you need to `make` it:

```
puppy# make
```

The NRPE package does not have an automatic installation script. Once the `make` process is complete, you will need to manually install the required files. I normally copy the `nrpe` daemon binary into the `/usr/local/nagios/bin` directory. As you can see on the following line, the `nrpe` daemon is created in the `src` directory in the NRPE package:

```
puppy# cp src/nrpe /usr/local/nagios/bin
```

I also copy the `nrpe.cfg` configuration file. It is created in the root directory of the NRPE package. I usually place this in the `/usr/local/nagios/etc` directory:

```
puppy# cp nrpe.cfg /usr/local/nagios/etc
```

---

7. I discussed the `nrpe` daemon in Chapter 5.
8. You'll need at least version OpenSSL 0.9.7a or a more recent version.

In order for the nrpe daemon to work, you also need to change the ownership of the nrpe.cfg configuration file to the user and group that is running the nrpe daemon. This allows the daemon to read the file. I'll do this on the following line:

```
puppy# chown nagios:nagios /usr/local/nagios/etc/nrpe.cfg
```

You should change its ownership to the user and group you have specified that nrpe should run as, in our case nagios and nagios.

We also need one plug-in, the check_nagios plug-in, to allow the slave server to perform checks of the Nagios process on the master server to detect when it stops. This plug-in is installed as part of the standard Nagios plug-in package.

Next we need to configure the nrpe.cfg file. Example 8-11 shows a typical nrpe.cfg configuration file.

Example 8-11. *Master Server nrpe.cfg Configuration File*

```
server_port=5666
server_address=10.0.0.1
allowed_hosts=127.0.0.1,10.0.0.15
nrpe_user=nagios
nrpe_group=nagios
dont_blame_nrpe=0
debug=0
command_timeout=60
command[check_nagios]=/usr/local/nagios/libexec/check_nagios ➥
-e 1 -F /usr/local/nagios/var/nagios.log -C /usr/local/nagios/bin/nagios
```

Let's look at each of the options in Example 8-11. The first two options, server_port and server_address, allow you to specify the port and address the nrpe daemon will listen on. This defaults to all interfaces on port 5666.

The next option, allowed_hosts, specifies the IP addresses that are allowed to contact the nrpe daemon and transmit command requests. The loopback address, 127.0.0.1, is included by default, and you need to specify the IP addresses of the slave server that needs to be able to connect to the nrpe daemon. You should use commas to separate the IP addresses you're specifying.

The next two options specify the user and group that the nrpe daemon will run as. In this case I've used nagios and nagios.

The dont_blame_nrpe option is used to allow checks via NRPE to be submitted with arguments. For checks of the Nagios process, I don't require that arguments be allowed so I've set this option to 0.

The command_timeout is the period of time NRPE waits to see if a command completes before failing. The default of 60 seconds should be sufficient for most purposes.

The last option is the command option, which allows you to specify commands that the nrpe daemon will execute. For checks of the Nagios process from the slave server, I only require one command that executes the check_nagios plug-in. Let's quickly look at how that plug-in works. The check_nagios plug-in checks two variables: whether the Nagios process is active and running and the age of the status log file, nagios.log. You can see a command-line execution of the plug-in in Example 8-12.

Example 8-12. *Command-Line Execution of the check_nagios Plug-in*

```
puppy# /usr/local/nagios/libexec/check_nagios -e 1 -F ➥
/usr/local/nagios/var/nagios.log -C /usr/local/nagios/bin/nagios
Nagios ok: located 1 process, status log updated 3 seconds ago
```

The first option, `-e`, specifies after what period in minutes the age of the log file is considered stale. I've specified 1 minute. The second option, `-F`, tells the plug-in the location of the `nagios.log` log file. By default this is `/usr/local/nagios/var/nagios.log`. The last option, `-C`, specifies the location of the Nagios binary. By default this would be `/usr/local/nagios/bin/nagios`.

As you can see from the command-line execution in Example 8-12, it returns a status message saying Nagios is running and reports the age of the `nagios.log` status log file in terms of its last update. If the Nagios process was not running, it would return the following error message:

```
Could not locate a running Nagios process!
```

When I'll configure the slave server I'll use the results of this plug-in check to configure the slave server to take over from the master. I'll also discuss further how I might configure the `check_nagios` plug-in.

Finally, we need to decide how the `nrpe` daemon will run. Running the `nrpe` daemon locally is a simple process and you can see on the following line:

```
puppy# /ustr/local/nagios/bin/nrpe -c /usr/local/nagios/etc/nrpe.cfg -d
Aug 10 21:07:37 puppy nrpe[18728]: Starting up daemon
```

On the previous lines, you can see I've specified two options to the `nrpe` daemon: `-c` and `-d`. The `-c` option specifies the location of the `nrpe.cfg` configuration file. The `-d` option tells the `nrpe` daemon to run as a stand-alone daemon.

■ **Tip** Also contained in the root directory of the NRPE source package are a series of `init` files for a number of different platforms. You can use these to automatically start, stop or restart the `nrpe` daemon. I recommend you use these `init` scripts for the `nrpe` daemon.

Configuring NSCA

Once the `nrpe` daemon is configured and running, we need to configure service obsession and NSCA to send passive check results to the slave server. To do this, follow the steps in the "Distributed Server Configuration" section earlier in this chapter. The master server performs the same function as a distributed server sending the check results, but instead of sending to a central server, it sends to the slave server. The slave server thus acts as the central server in the distributed monitoring model.

■ **Tip** So what if you need to send results both as a distributed server and to a redundant or failover server? Well, there is nothing to stop you from configuring host and service obsession to run a shell script that sends the check results to two servers instead of just one.

Configuring the Slave Server

To configure the slave server, you need to install Nagios much like you installed it on the master server. I also recommend installing the web console as the slave server might have to take over monitoring of your environment. You configure all the required hosts and services that the master server is monitoring. You also disable active checks of all these hosts and services and notifications.

This means you should disable active host and service checking using the `execute_service_checks` and `execute_host_checks` directives in the `nagios.cfg` configuration file. Setting these both to `0` will ensure the slave server does not execute host and service checks.

```
execute_service_checks=0
execute_host_checks=0
```

You should also disenable notifications using the `enable_notifications` directive like so:

```
enable_notifications=0
```

You also need to ensure you have a few directives in the `nagios.cfg` configuration file turned on. The following list of directives should be set to `1`:

```
accept_passive_service_checks=1
accept_passive_host_checks=1
check_external_commands=1
```

This ensures that the slave server will receive passive host and service checks using NSCA to check the status of the hosts and services being monitored. It also means that external commands will be checked, which will be important when I demonstrate how to monitor for and initiate the failover process.

Configuring NRPE

On the slave server let's use the `check_nrpe` plug-in to connect to the master server and check the status of the Nagios server process. This will allow the slave server to determine when it should take over monitoring of the environment. The first step in doing this is installing NRPE. Rather than the `nrpe` daemon that we required for the master server, let's only require the `check_nrpe` plug-in.

Let's quickly step through installing the `check_nrpe` plug-in on the slave server. Download the NRPE package and unpack it like so:

```
kitten# wget http://prdownloads.sourceforge.net/nagios/nrpe-2.0.tar.gz
kitten# tar -zxf nrpe-2.0.tar.gz
kitten# cd nrpe-2.0
```

Then run the `configure` script for the NRPE package like so:

```
kitten# ./configure --enable-ssl
```

As you can see, using the `--enable-ssl` `configure` option I've enabled native SSL/TSL support as I have on the master server. You must enable this option on both the master and slave servers if you wish to use SSL connections between them. As with the master server, you must also have OpenSSL installed on the slave server.[9]

After compilation you need to install the `check_nrpe` plug-in, located in the `src` directory of the NRPE package, into your plug-in directory. I've done this on the following line:

```
kitten# cp src/check_nrpe /usr/local/nagios/libexec
```

The `check_nrpe` plug-in executes commands defined in the `nrpe.cfg` configuration file on the master server. It does this by connecting to the `nrpe` daemon on TCP port 5666. In Example 8-11 I've defined a sample `nrpe.cfg` configuration file with the following command in it:

```
command[check_nagios]=/usr/local/nagios/libexec/check_nagios -e 1 -F ➥
 /usr/local/nagios/var/nagios.log -C /usr/local/nagios/bin/nagios
```

The `nrpe` daemon on the master server executes the `check_nagios` plug-in and returns the results of that check to the `check_nrpe` plug-in on the slave server. On the following lines, you can see this same check executed from the command line of the slave server:

```
kitten# ./check_nrpe -H 10.0.0.1 -c check_nagios
Nagios ok: located 1 process, status log updated 9 seconds ago
```

So how do we use this check to allow the slave server to determine when it should take over monitoring? I usually enclose this check into a shell script, monitor the exit status of the check, and initiate the failover process depending on the exit status. I then execute the script in a `cron` job. I've included the typical script I use in Example 8-13.

Example 8-13. *Nagios Process Monitor Script*

```
#!/bin/sh

cmd_file=/usr/local/nagios/var/rw/nagios.cmd

/usr/local/nagios/libexec/check_nrpe -H 10.0.0.1 -c check_nagios
return_code=$?

case "$return_code" in
            '0')
             TIME=`date +%s`
             echo "[$TIME] STOP_EXECUTING_SVC_CHECKS" >> $cmd_file
             echo "[$TIME] STOP_EXECUTING_HOST_CHECKS" >> $cmd_file
             echo "[$TIME] DISABLE_NOTIFICATIONS" >> $cmd_file
```

9. Again, you'll need at least version OpenSSL 0.9.7a or a more recent version.

```
        ;;
      '2')
        TIME=`date +%s`
        echo "[$TIME] START_EXECUTING_SVC_CHECKS" >> $cmd_file
        echo "[$TIME] START_EXECUTING_HOST_CHECKS" >> $cmd_file
        echo "[$TIME] ENABLE_NOTIFICATIONS" >> $cmd_file
        ;;
esac
exit 0
```

You can see in Example 8-13 that this is a very simple script. First, I define the location of the external command file, to which I am going to submit the commands that will trigger the failover process. In my default Nagios installation, this file is `/usr/local/nagios/var/rw/nagios.cmd`. Next, I execute the check of the Nagios process using the `check_nrpe` plug-in. This plug-in will return a standard Unix exit status, and I assign that status to a variable called `return_code`. I then use a `case` statement to evaluate the exit status and perform actions based on its value.

If the script returns an exit status of `0`, this indicates that the Nagios process is functioning and that the master server remains the active and primary server. This status triggers the submission of three external commands: `STOP_EXECUTING_SVC_CHECKS`, `STOP_EXECUTING_HOST_CHECKS`, and `DISABLE_NOTIFICATIONS`.[10] These commands stop active service and host checks and disable notifications on the slave server.

The submission of these commands has two purposes. They are first a safety net to ensure that the slave server does not accidentally have active checks or notifications enabled. If this occurs when the master server is still available, these commands will disable those functions. Second, this creates an automated process of reversing the failover process. If the master server becomes available again, the slave server will stop active checks and notifications.

This is not a perfect model for reverting from the failed-over state. This is because if a failover has been initiated, the process of recovering the failover can be quite complicated. First, depending on how long the slave server took over from the master server, the check results on the master server may be out of date. The master server will also not know about any notifications generated, any performance or statistical data you are collecting, or the like. For example, there will be gaps in your availability reporting on the master server. Unfortunately, little can be done to alleviate these issues.

The next `case` statement is initiated when the exit status of the check is `2`. This indicates that the Nagios process is not running, or that the check has failed or cannot be completed. In this case, three different external commands are submitted: `START_EXECUTING_SVC_CHECKS`, `START_EXECUTING_HOST_CHECKS`, and `ENABLE_NOTIFICATIONS`. These commands initiate the failover process and tell the slave server to take over active checks and notifications from the master server.

We then configure this shell script to be executed by the `cron` daemon on a regular basis. I recommend an interval of 1 to 5 minutes depending on how often you wish to check the status of your master server.

```
0-59/5 * * * * /usr/local/nagios/libexec/test_failover >/dev/null 2>&1
```

10. I discussed how to manually submit external commands in Chapter 7.

On the previous line you can see a `crontab` entry for a check at five-minute intervals (the `>/dev/null 2>&1` makes the `cron` job quiet and prevents the output from being generated).

■ **Tip** Remember, if you have a scheduled outage on your master server such as a reboot and it occurs during a check of the master service by the slave server, your slave server will take over monitoring. You may wish to disable checking of the master server during scheduled downtime.

Installing NSCA

Next we need to install the NSCA daemon to receive the passive check results from the master server. To do this follow the steps in the "Installing the NSCA Daemon" and "Configuring the NSCA Daemon" sections earlier in this chapter. You should also consider enabling freshness checking, as discussed in the "Distributed Monitoring and Freshness" section, to ensure the status information of your hosts and services is up to date.

Failover Process

Once you've set up both the master and slave servers, you should now have a functioning failover solution. Just to remind you how this will work, let's step through the process and interaction that occurs between the master and slave:

1. The master server monitors all the required hosts and services.
2. The master server uses host and service obsession and the `send_nsca` plug-in to send the results of checks to the slave server.
3. The slave server performs no checks itself but receives the results of the checks conducted on the master server using the NSCA daemon and applies these check results as passive host or service checks. This ensures the slave server has up-to-date status information for your hosts and services.
4. The slave server checks the status of the Nagios process using the `check_nrpe` plug-in. It connects to the `nrpe` daemon on the master server, executes the `check_nagios` plug-in, and returns the status of the Nagios process.
5. If the slave server receives an `OK` check result from the `check_nagios` plug-in on the master server, it submits external commands on the slave server disabling active checks and notifications. Checks of the Nagios process continue normally.
6. If the slave server receives a non-`OK` check result from the `check_nagios` plug-in on the master server, it assumes the server is unavailable. The process then submits external commands on the slave server enabling active checks and notifications. This indicates that the master server has failed over to the slave server.
7. If the master server becomes available again, the process will reverse the failover and the master server will become the active monitoring and notifying server. The slave server will return to the stand-by mode.

Checkpoint

- If you are using NSCA or NRPE, enable encryption to ensure that your communications are not compromised or intercepted, and that malicious commands are not submitted to your Nagios servers.
- If you are using distributed monitoring, consider adding freshness checking on your central server to ensure that your passive check results are up-to-date and accurate.
- You will need to keep your configurations objects synchronized between your Nagios servers. For both distributed monitoring and failover, the Nagios servers involved must have the appropriate object definitions defined on them.
- Be aware that if you fail over to your slave server any monitoring done by that server, notifications generated, or statistics gathered will not be replicated on the master server. Additionally, if there is a long period of failover, when your master server takes over again it might take some time for that server to check all the hosts and services and become up-to-date.

Resources

These are a list of sites that I have referred to in this chapter as well as other sites where you can seek further information.

Sites

- NSCA: `www.nagiosexchange.org/Communication.41.0.html?&tx_netnagext_pi1[p_view]=140`
- NSCA RPM: `http://dries.studentenweb.org/rpm/packages/nagios-nsca/info.html`
- MCrypt: `http://mcrypt.sourceforge.net/`
- MCrypt RPMs: `www.ottolander.nl/opensource/mcrypt/mcrypt.html` or `http://dag.wieers.com/packages/libmcrypt/`