Wolfgang Barth

Nagios

System and Network Monitoring

Munich

NO STARCH
PRESS

San Francisco

Chapter 7

Testing Local Resources

The plugins introduced in this chapter from the basis range of the **nagios-plugins** package test local resources that do not have their own network protocol and therefore cannot be easily queried over the network. They must therefore be locally installed on the computer to be tested. Such plugins on the Nagios server can test only the server itself—with command and service definitions as described in Chapter 6.

To perform such local tests from a central Nagios server on remote hosts, you require further utilities: the plugins are started via a secure shell, or you use the *Nagios Remote Plugin Executor* (NRPE). Using the secure shell is described in Chapter 9 from page 157, and Chapter 10 (page 165) is devoted to NRPE.

The definition of command and service depends on the choice of mechanism. If you want to test for free hard drive capacity with the **check_by_ssh** plugin installed on the Nagios server, which remotely calls **check_disk** on the target server (see Section 7, page 133), then a special command definition is required for this, which differs

somewhat from the definitions given in Chapter 6 (page 85). What command and service definitions for remotely executed local plugins look like is described in the aforementioned chapters on NRPE and SSH.

For the remote query of some local resources you can also use SNMP (see Chapter 11 from page 177), but the checks are then restricted to the capabilities of the SNMP daemon used. Local plugins are usually more flexible here and provide more options for querying.

7.1 Free Hard Drive Capacity

The question of when the hard drive(s) of a computer may threaten to overflow is answered by the **check_disk** plugin, which in version 1.4 includes considerably more functions than its predecessor:

-w *limit* / **--warning=***limit*
: The plugin will give a warning if the free hard drive capacity drops below this limit, expressed as a percentage or as an integer. If you specify percentage, the percent sign **%** must also be included; floating-point decimals such as **12.5%** are possible. Integer values in kBytes are demanded by version 1.3.x, but by version 1.4 in MBytes (in each case without a unit abbreviation). The unit can also be influenced with **-k**, **-k**, and **-u**.

-c *limit* / **--critical=***limit*
: If the free hard drive capacity level falls below this as a percentage or integer (see **-w**), **check_disk** displays the CRITICAL status. The critical limit must be smaller than the warning limit.

-p *path_or_partition* / **--path=***path* or **--partition=***partition*
: This specifies the root directory in file systems or the physical device in partitions (e.g., **/dev/sda5**). From version 1.4 **-p** can be called multiple times. If the path is not specified, the plugin tests all file systems (see also **-x** and **-X**).

-e / **--errors-only**
: With this switch, the plugin shows only the file systems or partitions that are in a WARNING or CRITICAL state.

-k / **--kilobytes** (from 1.4)
: With this switch, limit values given as whole numbers with **-c** and **-w** are to be interpreted as kBytes.

-m / **--megabytes** (from 1.4)
: With this switch, whole number limit values with **-c** and **-w** are interpreted by the plugin as MBytes (the default). Caution: in version 1.3.x, **-m** has a completely different meaning!!

-m / --mountpoint (1.3.x)
: Normally **check_disk** in version 1.3.x will return the physical device (e.g., **/dev/sda5**). **-m** ensures that the file system path (e.g., **/usr**) is named instead.

-M / --mountpoint (from 1.4)
: From version 1.4 on, **check_disk** by default displays the file system path (e.g., **/usr**). With **-M** you are told instead what physical device (e.g., **/dev/sda5**) is involved.

-t *timeout* / --timeout=*timeout*
: After *timeout* seconds have expired the plugin stops the test and returns the CRITICAL status. The default is **10** seconds.

-u *unit* / --units=*unit* (from 1.4)
: In what unit do you specify integer limit values? **kB**, **MB**, **GB** and **TB** are all possible.

-x *path* / --exclude_device=*path*
: This switch excludes the mount point specified as *path* from the test. This option may be used several times in a plugin command.

-X *fs_typ* / --exclude-type=*fs_typ* (from 1.4)
: This switch excludes a specific file system type from the test. It is given the same abbreviation as in the **-t** option of the **mount** command. In this way *fs_type* can take the values **ext3**, **reiserfs**, or **proc**, for example (see also **man 8 mount**). This option can be used several times in a plugin command.

-C / --clear (from 1.4)
: From version 1.4 on, **-p** can be used multiple times. If you want to test several file systems at the same time, but using different limit values, **-C** can be used to delete old limit values that have been set:

```
-w 10% -c 5% -p / -p /usr -C -w 500 -c 100 -p /var
```

: The order is important here: the limit values are valid for the file system details until they are reset with **-C**. Then new limits must be set with **-w** and **-c**.

The plugin versions 1.3.1 (above example) and 1.4 differ not only in their options, but also in their output. Performance data are missing from the latter (see Chapter 17 from page 313):

```
user@linux:nagios/libexec$ ./check_disk -w 10% -c 5% -p /usr
DISK CRITICAL [87000 kB (5%) free on /usr]
user@linux:nagios/libexec$ ./check_disk -w 10% -c 5% -p /
DISK OK - free space: / 710 MB (74%);| /=247MB;861;909;0;957
```

These can be extracted from the Nagios log files and prepared in graphic form.

The following example functions only with version 1.4:

```
user@linux:nagios/libexec$ ./check_disk -w 10% -c 5% -p / -p /usr \
    -p /var -C -w 5% -c 3% -p /net/emil1/a -p /net/emil1/c -e
DISK WARNING - free space: /net/emil1/c 915 MB (5%);| /=146MB;458;483;0;
509 /usr=1280MB;3633;3835;0;4037 /var=2452MB;3633;3835;0;4037 /net/emil1
/a=1211MB;21593;22048;0;22730 /net/emil1/c=17584MB;17574;17944;0;18499
```

Everything is in order on the file system **/**, **/usr**, and **/var**, since more space is available on them—as can be seen from the performance data—than the limit value of 10 percent (for a warning), and certainly more than 5 percent (for the critical status). The file systems **/net/emil1/a** and **/net/emil1/c** encompass significantly larger ranges of data, which is why the limit values are set lower, after the previous ones have been deleted with **-C**.

-e ensures that Nagios shows only the file systems that really display an error status. In fact the output of the plugin *before* the **|** sign, with **/net/emil1/c**, only displays one single file system. The performance information after the pipe can only be seen on the command line—it contains all file systems tested, as before. This is slightly confusing, because a Nagios plugin restricts its output to a single line, which has been line wrapped here for this printed version.

7.2 Utilization of the Swap Space

The **check_swap** plugin tests the locally available swap space. Here there are again fundamental differences between versions 1.3.x and 1.4:

-w *limit* | --warning=*limit*
: The warning limit can be specified as a percentage or as an integer, as with **check_disk**, but the integer value is specified in *bytes*, not in kBytes!

  In version 1.3.x the percentage specification refers to used, and not free, swap space. If at least 10 percent should remain free, you must specify **-c 10%** in version 1.4, but **-c 90%** in version 1.3. The integer specification, however, refers to the remaining free space for both versions.

-c *limit* | --critical=*limit*
: Critical limit, similar to the warning limit. If a percentage is specified, versions 1.3.x and 1.4 differ, as in the **-w** option.

-a | --allswaps
: Tests the threshold values for each swap partition individually.

The following example tests to see whether at least half of the swap space is available. If there is less than 20 percent free swap space, the plugin should return a critical status. The output is from plugin version 1.4, and after the | sign the program again provides performance data, which is logged by Nagios but not displayed in the message on the Web interface:

```
user@linux:nagios/libexec$  ./check_swap -w 50% -c 20%
swap OK: 100% free (3906 MB out of 3906 MB) |swap=3906MB;1953;781;0;3906
```

7.3 Testing the System Load

The load on a system can be seen from the number of simultaneously running processes, which is tested by the **check_load** plugin. With the help of the **uptime** program, it determines the average value for the last minute, the last five minutes, and the last 15 minutes. **uptime** displays these values in this sequence after the keyword **load average**:

```
user@linux:~$  uptime
 16:33:35 up  7:05, 18 users,  load average: 1.87, 1.38, 0.74
```

check_load has only two options (the two limit values), but these can be specified in two different ways:

-w *limit* | --warning=*limit*
: This option specifies the warning limit either as a simple floating-point decimal (**5.0**) or as a comma-separated triplet containing three-floating point decimals (**10.0,8.0,5.0**).

  In the first case, the limit specified applies to all three average values. The plugin issues a warning if (at least) one of these is exceeded. In the second case the triplet allows the limit value to be specified separately for each average value. Here as well, **check_load** issues a warning as soon as one of the average values exceeds the limit defined for it.

-c *limit* | --critical=*limit*
: This specifies the critical limit in the same way as **-w** specifies the warning limit. These critical limit values should be higher than the values for **-w**.

In the following example Nagios would raise the alarm if more than 15 processes were active on average in the last minute, if more than 10 were active on average in the last five minutes, or if eight were active on average in the last 15 minutes. There is a warning for average values of ten, eight, or five processes:

```
user@linux:local/libexec$ ./check_load -w 10.0,8.0,5.0 -c 15.0,10.0,8.0
OK - load average: 1.93, 0.95, 0.50| load1=1.930000;10.000000;15.000000;
0.000000 load5=0.950000;8.000000;10.000000;0.000000 load15=0.500000;
5.000000;8.000000;0.000000
```

7.4 Monitoring Processes

The **check_procs** plugin monitors processes according to various criteria. Usually it is used to monitor the running processes of just one single program. Here the upper and lower limits can also be specified.

nmbd, for example, the name service of Samba, always runs as a daemon with two processes. A larger number of **nmbd** entries in the process table is always a sure sign of a problem; it is commonly encountered, especially in older Samba versions.

Services such as Nagios itself should only have one main process. This can be seen by the fact that its parent process has the process ID **1**, marking it is a child of the **init** process. It was often the case, in the development phase of Nagios 2.0, that several such processes were active in parallel after a failed restart or reload, which led to undesirable side effects. You can test to see whether there really is just one single Nagios main process active, as follows:

```
nagios@linux:nagios/libexec$ ./check_procs -c 1:1 -C nagios -p 1
PROCS OK: 1 process with command name 'nagios', PPID = 1
```

The program to be monitored is called **nagios** (option **-C**), and its parent process should have the ID **1** (option **-p**). Exactly one Nagios process must be running, no more and no less; otherwise the plugin will issue a CRITICAL status. This is specified as a range: **-c 1:1**.

Another example: between one and four simultaneous processes of the OpenLDAP replication service **slurpd** should be active:

```
nagios@linux:nagios/libexec$ ./check_procs -w 1:4 -c 1:7 -C slurpd
PROCS OK: 1 process with command name 'slurpd'
```

If the actual process number lies between **1** and **4**, the plugin returns OK, as is the case here. If it finds between five and seven processes, however, a warning will be given. Outside this range, **check_procs** categorizes the status as CRITICAL. This is the case here if there are either no processes running at all, or more than seven running.

Instead of the number of processes of the same program, you can also monitor the CPU load caused by it, its use of memory, or even the CPU runtime used. **check_procs** has the following options:

-w *start*:*end* | --warning=*start*:*end*

The plugin issues a warning if the actual values lie *outside* the range specified by the start and end value. Without further details, it assumes that it should count processes: **-w 2:10** means that **check_procs** gives a warning if it finds less than two or more than ten processes.

If you omit one of the two limit values, zero applies as the lower value, or infinite as the upper limit. This means that the range **:10** is identical to **0:10**; **10:** describes any number larger than or equal to 10. If you just enter a single whole number instead of a range, this represents the maximum. The entry **5** therefore stands for **0:5**.

If you swap the maximum and minimum, the plugin will give a warning if the actual value lies *within* the range, so for **-w 10:5** this will be if the value is 5, 6, 7, 8, 9 or 10. You may always specify only one interval.

-c *start*:*end* | --critical=*start*:*end*

This specifies the critical range, in the same way as for the warning limit.

-m *type* | --metric=*type* (from version 1.4)

This switch selects one of the following metrics for the test:

PROCS: number of processes (the default if no specific type is given)

VSZ: the virtual size of a process in the memory (*virtual memory size*), consisting of the main memory space that the process uses exclusively, plus that of the shared libraries used. These only take up memory space once, even if they are used by several different processes. The specification is given in bytes.

RSS: the proportion of main memory in bytes that the process actually uses for itself (*Resident Set Size*), that is, **VSZ** minus the shared memory.

CPU: CPU usage in percent. The plugin here checks the CPU usage for each individual process for morning and critical limits. If one of the processes exceeds the warning limit, Nagios will issue a warning. In the text output the plugin also shows how many processes have exceeded the warning or critical limit.

ELAPSED: The overall time that has passed since the process was started.

-s *flags* | --state=*flags*

This restricts the test to processes with the specified status flag.[1] The plugin in the following example gives a warning if there is more than one zombie process (status flag: **Z**):

[1] The following states are possible in Linux: **D** (uninterruptible waiting, usually a *Disk Wait*), **R** (running process), **S** (wait status), **T** (process halted), **W** (paging, only up to kernel 2.4), **X** (a finished, killed process), and **Z** (zombie). Further information is provided by **man ps**.

```
nagios/libexec@linux: $ ./check_procs -w 1 -c 5 -s Z
PROCS OK: 0 processes with STATE = Z
```

Things become critical here if more than five zombies "block up" the process table. Several states can be queried at the same time by by adding individual flags together, as in **-s DSZ**. Now Nagios cancels the processes that are in at least one of the states mentioned.

-p *ppid* | --ppid=*ppid*
: This switch restricts the test to processes whose parent processes have the *parent process ID* (*ppid*). The only PPIDs that are known from the beginning, and that do not change, are 0 (started by the kernel, and usually only concerns the init process) and 1 (the init process itself).

-P *pcpu* | --pcpu=*pcpu* (from version 1.4)
: This option filters processes according to the percentage of CPU they use:

```
nagios/libexec@linux: $ ./check_procs -w 1 -c 5 -P 10
PROCS OK: 1 process with PCPU >= 10,00
```

The plugin in this example takes into account only processes which have at least a ten percent share of CPU usage. As long as there is just one such process (**-w 1**), it returns OK. If there are between two and five such processes, the return value is a WARNING. With at least six processes, each with a CPU usage of at least ten percent, things get critical.

-r *rss* | --rss=*rss* (from version 1.4)
: This option filters out processes that occupy at least *rss* bytes of main memory. It is used like **-P**.

-z *vsz* | --vsz=*vsz* (from version 1.4)
: This option filters out processes whose VSZ (see above) is at least *vsz* bytes. It is used like **-P**.

-u *user* | --user=*user*
: This option filters out processes that belong to the specified user (see example below).

-a "*string*" | --argument-array="*string*"
: This option filters out commands whose argument list contains *string*. **-a .tex**, for example, refers to all processes that work with ***.tex files**; **-a -v** to all processess that are called with the -v flag.

-C *command* | --command=*command*
: This causes the process list to be searched for the specified command name. *Command* must exactly match the command specified, without a path (see example below).

-t *timeout* | --timeout=*timeout*
: After *timeout* seconds have expired, the plugin stops the test and returns the CRITICAL status. The default is 10 seconds.

The following example checks to see whether exactly one process called master is running on a mail server on which the Cyrus Imapd is installed. No process is just as much an error as more than one process:

```
user@linux:nagios/libexec$ ./check_procs -w 1:1 -c 1:1 -C master
CRITICAL - 2 processes running with command name master
```

The first attempt returns two processes, although only a single Cyrus Master process is running. The reason can be found if you run ps:

```
user@linux:~$ ps -fC master
UID        PID PPID  C STIME TTY          TIME CMD
cyrus      431     1  0  2004 ?        00:00:28 /usr/lib/cyrus/bin/master
root      1042     1  0  2004 ?        00:00:57 /usr/lib/postfix/master
```

The Postfix mail service also has a process with the same name. To keep an eye just on the master process of the Imapd, the search is additionally restricted to processes running with the permissions of the user cyrus:

```
user@linux:nagios/libexec$ ./check_procs -w 1:1 -c 1:1 -C master -u \
    cyrus
OK - 1 processes running with command name master, UID = 96 (cyrus)
```

7.5 Checking Log Files

Monitoring log files is not really part of the concept of Nagios. On the one hand, the syslog daemon notices critical events there immediately, so that an error status can be correctly determined. But if the error status continues, this cannot be seen in the log file in most cases.

Correspondingly the plugins described here can determine only whether other, new entries on error events are added. In order to communicate information on a continuing error behavior to Nagios via a log file, the service monitored must log the error status regularly—at least at the same intervals as Nagios reads the log file—and repeatedly. Otherwise the plugin will alternate between returning an error status, and then an OK status, depending on whether the (continuing) error has in the meantime turned up in the log or not.

Under no circumstances may Nagios repeat its test. The parameter max_check_attempts (see page 45) must have the value 1. Otherwise Nagios would first assign

the error status as a soft state, would repeat the test, and would almost always arrive at an OK, since it only takes into account new entries during repeat tests. **max_check_attempts = 1** ensures that Nagios diagnoses a hard state after the first test.

For events that log an error just once, Nagios has *volatile services*, described in Section 14.5.2 from page 257. For services defined in this way, the system treats every error status as if it was occurring for the first time (causing a message to be sent each time, for example). Such services must be reset manually to the OK status. How this is done is described in Section 14.5.3 from page 258.

7.5.1 The standard plugin check_log

With **check_log**, Nagios provides a simple plugin for monitoring log files. It creates a copy of the tested log file each time it is run. If the log file has changed since the previous call, **check_log** searches the newly added data for simple text patterns. The plugin does not have any longer options and just has the states OK and CRITICAL:

-F *logfile*
: This is the name and path of the log file to be tested. It must be readable for the user **nagios**.

-O *oldlog*
: This is the name and path of the log file copy. The plugin just examines the difference between *oldlog* and *logfile* when it is run. Afterwards it copies the current log file to *oldlog*. *oldlog* must contain the absolute path and be readable for the user **nagios**.

-q *query*
: This is the pattern searched for in examining the log file. Not found means OK; a match returns the CRITICAL status.

It is recommended that you generally do not use messages of the type *recovery notification* (OK after an error state).

An OK in a repeated test just means that no new error in events have occurred since the last test. The **notification_options** parameter (see page 46) in the service definition should therefore not contain an **r**.

The following command examines the file **/var/log/auth** for failed logins:

```
nagios@linux:local/libexec$  ./check_log -F /var/log/auth \
    -O /tmp/check_log.badlogin -q "authentication failure"
(1) < Jan  1 18:47:56 swobspace su[22893]: (pam_unix) authentication
failure; logname=wob uid=200 euid=0 tty=pts/8 ruser=wob rhost= user=root
```

This produces one hit. The plugin does not show its return value in the text, but it can be displayed in the shell with **echo $?**. In the example, a **2** for CRITICAL is returned.

If you examine the log file for several different events, you must specify a separate *oldlog* for each log file:

```
./check_log -F /var/log/messages -O /tmp/check_log.pluto -q "pluto"
./check_log -F /var/log/messages -O /tmp/check_log.ntpd -q "ntpd"
```

Even if you are searching in the same original log file, you cannot avoid using two different **oldlogs**: otherwise **check_log** would not work correctly.

7.5.2 The modern variation: check_logs.pl

As an alternative, The Nagios Exchange[2] provides a completely new plugin for monitoring log files. **check_logs.pl** represents a further development of the Perl plugin **check_log2.pl**, which is included in the **contrib** directory for Nagios plugins but is not installed automatically.

check_logs.pl can examine several log files simultaneously for events, in contrast to **check_log** and **check_log2.pl**. It requires a configuration file to do this.

It does have a simple command line mode, but this functions only if you specify a single log file and a single regular expression simultaneously. But the really interesting feature of **check_logs.pl** is that you can perform several examinations in one go. This is why we will not spend any more time describing the command line mode.

Initially we create a configuration file with roughly the following contents, preferably in the directory **/etc/nagios**:

```
# /etc/nagios/check_logs.cfg
$seek_file_template='/var/nagios/$log_file.check_log.seek';

@log_files = (
   {'file_name' => '/var/log/messages',
    'reg_exp' =>'ntpd',
   },
   {'file_name' => '/var/log/warn',
    'reg_exp' =>'(named|dhcpd)',
   },
);
1;
```

The Perl variable **$seek_file_template** contains the path to the file in which the plugin saves the current position of the last search. **check_logs.pl** remembers here

[2] http://www.nagiosexchange.org/Misc.54.0.html.

at what point in the log file it should carry on searching the next time it is run. This means that the plugin does not require a copy of the processed log file. Instead of the variable **$log_file**, it uses the name of the log file to be examined in each case and creates a separate position file for each log file.

What exactly **check_logs.pl** is to do is defined by the Perl array **@log_files**. The entry **file_name** points to the log file to be tested (with the absolute path), and **reg_exp** contains the regular expression[3], for which **check_logs.pl** should search the log file. In the example above this is just a simple text called **ntpd** in the case of the **/var/log/messages** log file, but there is an alternative in the case of **/var/log/warn**: the regular expression **(named|dhcpd)** matches lines that contain either the text **named** or the text **dhcpd**.

The only specification that the plugin itself requires when it is run is the configuration file (option **-c**) :

```
nagios@linux:local/libexec$ ./check_logs.pl -c /etc/nagios/check_logs.cfg
messages => OK; warn => OK;

nagios@linux:local/libexec$ ./check_logs.pl -c /etc/nagios/check_logs.cfg
messages => OK; warn => (4): Jul  2 14:33:25 swobspace dhcpd:
Configuration file errors encountered -- exiting;
```

The first command shows the basic principle: in the text output the plugin for each log file announces separately whether it has found a matching event or not. In the above example it didn't find anything, so it returns OK. In the second command the plugin comes across four relevant entries in the **warn** log file, but it doesn't find any in **/var/log/messages**. Because of this, the plugin returns a WARNING; OK is given only if no relevant events were found in any of the log files checked. In its output line, after **(4):**, the plugin remembers the last of the four lines found.

7.6 Keeping Tabs on the Number of Logged-in Users

The plugin **check_users** is used to monitor the number of logged-in users:

```
user@linux:nagios/libexec$ ./check_users -w 5 -c 10
USERS CRITICAL - 20 users currently logged in |users=20;5;10;0
```

[3] In the form of Perl-compatible regular expressions (PCRE, see **man perlre**), since **check_logs.pl** is a Perl script.

It has just two options:

-w *number* | --warning=*number*
: This is the threshold for the number of logged-in users after which the plugin should give a warning.

-c *number* | --critical=*number*
: This is the threshold for a critical state, measured by the number of logged-in users.

The performance data after the | is as usual visible only on the command line; Nagios does not include it in the Web interface.

7.7 Checking the System Time

7.7.1 Checking the system time via NTP

The **check_ntp** plugin compares the clock time of the local computer with that of an available NTP server in the network. If the Nagios server keeps time via NTP accurately enough, so that it can serve as a reference itself, then it can also be used as a network plugin, provided that the host to be checked in the network has an NTP daemon installed.

The plugin requires the program **ntpdate**, which, if you compile Nagios yourself, must already be available before the **check_ntp** installation. You should also install the program **ntpq**, which determines the *jitter*. This is a measure of the runtime deviations of incoming NTP packages. If the fluctuations are too large, the time synchronization will be imprecise.

In the simplest case, **check_ntp** is called, specifying the computer (here: **ntpserver**) whose time should be compared with that of the local computer:

```
nagios@linux:nagios/libexec$ ./check_ntp -H ntpserver
NTP OK: Offset -8.875159 secs, jitter 0.819 msec, peer is stratum 0
```

The deviation found here is over eight seconds. Whether this is tolerated or not depends on the intended use. If you want to compare log file entries for many computers, then they should all be NTP-synchronized. Then there is no problem in using **-w 1 -c 2**, which would already categorize a deviation of two seconds as critical.

check_ntp has the following options:

-H *address* | --host=*address*
: This is the NTP server with which the plugin should compare the local system time.

-w *floating_point_decimal* | --warning=*floating_point_decimal*
: This is the warning limit in seconds. The warning is given if the fluctuation of the local system time is larger than the threshold specified. The default is 60 seconds.

-c *floating_point_decimal* | --critical=*floating_point_decimal*
: If the local system time deviates more than *floating_point_decimal* seconds (in the default setting 120 seconds) from that of the NTP server, the status becomes CRITICAL.

-j *milliseconds* | --jwarn=*milliseconds*
: This is the warning limit for the jitter in milliseconds. The default here is 5000.

-k *milliseconds* | --jcrit=*milliseconds*
: The critical threshold for the jitter. The default is 10000 milliseconds.

7.7.2 Checking system time with the time protocol

Apart from the *Network Time Protocol* NTP there is another protocol, older and more simple: the *Time Protocol* described in RFC 868, in which communication takes place via TCP port 37. On many Unix systems the corresponding server is integrated into the inet daemon, so you do not have to start a separate daemon. With check_time, Nagios provides an appropriate test plugin.

check_time can also be used as a network plugin, in a similar way to check_ntp, but this again assumes that the time service is available for every client. In most cases it will therefore be used as a local plugin that compares its own clock time with that of a central time server (here: timesrv):

```
nagios@linux:nagios/libexec$ ./check_time -H timesrv -w 10 -c 60
TIME CRITICAL - 1160 second time difference| time=0s;;;0 offset=1160s;10
;60;0
```

The performance data after the | sign, not shown in the Web interface, contains the response time in seconds, with time (here: zero seconds); offset describes by how much the clock time differs from that of the time server (here: 1160 seconds). The other values, each separated by a semicolon, provide the warning limit, the critical threshold, and the minimum (see also Section 17.1 from page 314). Since we have not set any threshold values with the options -W or -C, the corresponding entries for time are empty.

check_time has the following options:

-H *address* | --hostname=*address*
: This is the host name or IP address of the time server.

-p *port* | --port=*port*
: This is the TCP port specification, if different from the default **37**.

-u | --udp
: Normally the time server is queried via TCP. With **-u** you can use UDP if the server supports this.

-w *integer* | --warning-variance=*integer*
: If the local time deviates more than ***integer*** seconds from that of the time server, the plugin returns a WARNING. ***integer*** is always positive, and this covers clocks that are running both slow and fast.

-c *integer* | --critical-variance=*integer*
: If there is more than ***integer*** seconds difference between the local and the time server time, the return value of the plugin is CRITICAL.

-W *integer* | --warning-connect=*integer*
: If the time server needs more than ***integer*** seconds for the response, a WARNING is returned.

-C *integer* | --critical-connect=*integer*
: If the time server does not respond within ***integer*** seconds, the plugin reacts with the return value CRITICAL.

7.8 Regularly Checking the Status of the Mail Queue

The **check_mailq** plugin can be used to monitor the mail queue of a mail server for e-mails that have not yet been delivered. **check_mailq** runs the program **mailq** of the mail service installed. Unfortunately each MTA interprets the mail queue differently, so the plugin can evaluate only mail queues from mail services that the programmer has taken into account. These are, specifically: **sendmail**, **qmail**, **postfix**, and **exim**. **check_mailq** has the following options:

-w *number* | --warning=*number*
: If there are at least ***number*** mails in the mail queue, the plugin gives a warning.

-c *number* | --critical=*number*
: As soon as there are at least ***number*** of mails in the queue waiting to be delivered, then the critical status has been reached.

-W *number_of_domains* | --Warning=*number_of_domains*
: This is the warning limit with respect to the number of recipient domains of a message waiting in the mail queue. Thus **-W 3** generates a warning if there are any mails in the queue that are addressed to three or more different recipient domains.

-C *number_of_domains* | --Critical=*number_of_domains*
: This is the critical threshold with respect to the number of recipient domains (like **-W**).

-M *daemon* | --mailserver=*daemon* (from version 1.4)
: This specifies the mail service used. Possible values for *daemon* are **sendmail** (the default), **qmail**, **postfix**, and **exim**.

-t *timeout* | --timeout=*timeout*
: After *timeout* seconds, the plugin stops the test and returns the CRITICAL status. The default here—as an exception—is **15** seconds (usually it is 10 seconds).

In the following example, Nagios should give a warning if there are at least five mails in the queue; if the number reaches ten, the status of the MTAs Postfix used here becomes CRITICAL:

```
user@linux:nagios/libexec$ ./check_mailq -w 5 -c 10 -M postfix
OK: mailq reports queue is empty|unsent=0;5;10;0
```

Since the queue is empty, **check_mailq** returns OK here.

7.9 Keeping an Eye on the Modification Date of a File

With the **check_file_age** plugin you can monitor not only the last modification date of a file, but also its size. From version 1.4 it is included in the default installation. In version 1.3.x the sources can be found in the subdirectory **contrib**; the plugin created from this must be copied manually to the plugin directory.

In the simplest case it is just run with the name and path of the file to be monitored:

```
user@linux:nagios/libexec$ ./check_file_age /var/log/messages
WARNING - /var/log/syslog/messages is 376 seconds old and 7186250 bytes
```

Here the plugin gives a warning, since the warning limit set is 240 seconds and the critical limit, 600 seconds. The last modification of the file was 376 seconds ago—that is, inside the warning range.

The file size is taken into account by **check_file_age** only if a warning limit for the file size (option **-W**) is explicitly specified. The plugin could then give a warning if the file is smaller than the given limit (in bytes). The defaults for the warning and critical limits here are both zero bytes.

check_file_age has the following options:

-w *integer* | --warning-age=*integer*
: If the file is older than *integer*[4] (the default is **240**) seconds, the plugin issues a warning.

-c *integer* | --critical-age=*integer*
: A critical status occurs if the file is older than *integer* (default: **600**) seconds.

-W *size* | --warning-size=*size*
: If the file is smaller than *size* bytes, the plugin gives a warning. If the option is omitted, **0** bytes is the limit. In this case **check_file_age** does not take the file size into account.

-C *size* | --critical-size=*size*
: A file size smaller than *size* bytes sets off a critical status. The default is **0** bytes, which means that the file size is ignored.

-f *file* | --file=*file*
: The name of the file to be tested. The option may be omitted if you instead—as in the above example—just give the file name itself as an argument.

7.10 Monitoring UPSs with apcupsd

To monitor uninterruptible power supplies (UPS) from the company APC there is the possibility, apart from the Network UPS Tools described in Section 6.11 from page 126 of using the **apcupsd** daemon, optimized specifically for use with these UPSs. The software can be obtained from **http://www.apcupsd.com/** and is licensed under the GPL, despite the fact that it is vendor-dependent.

The principal function here is the capacity to be able to shut down systems in the event of power failure, rather than a mere monitoring function with Nagios. For this latter purpose, it is easier to configure the Network UPS Tools.

Nearly all Linux distributions contain a working **apcupsd** package,[5] so you don't have to worry about installing it. Nagios does not include an **apcupsd** plugin, but

[4] Because **check_file_age** is a Perl script, it does not matter in this case whether an integer or a floating-point decimal is specified. Fractions of a second do not play a role in the file system.

[5] At least SuSE and Debian use this package name.

there is a very simple and effective script available for download at http://www.negative1.org/check_apc/: check_apc. It is also licensed under the GPL, but it has no network capabilities. The plugin cannot be given a host when it is run, and it also does not support any other types of options. Instead of this, internal commands control its functionality, which are given as the first argument.

Executing check_apc status tests whether the UPS is online. If this is the case, the plugin returns the OK status, in all other cases it returns CRITICAL:

```
user@linux:nagios/libexec$ ./check_apc status
UPS OK - ONLINE
```

check_apc load *warn crit* checks the load currently on the UPS and displays it as a percentage of the maximum capacity. A warning is given if the load is greater than the warning limits specified in *warn* (in the following example, 60 percent), CRITICAL if the load is greater than *crit* (here 80 percent):

```
user@linux:nagios/libexec$ ./check_apc load 60 80
UPS OK - LOAD: 39%
```

The load status of the UPS is checked by the command check_apc bcharge *warn crit*. Here the warning limit *warn* and the critical limit *crit* are also given in percent. The value 100 means "fully loaded." The plugin accordingly gives a warning if the load is smaller than the warning limit, and a CRITICAL if the load is smaller than the critical limit:

```
user@linux:nagios/libexec$ ./check_apc bcharge 50 30
UPS OK - Battery Charge: 100%
```

You can find out how long the saved energy will last with check_apc time *warn crit*. Here check_apc gives a warning if the remaining time is less than *warn* minutes, and a CRITICAL if the remaining time is less than *crit* minutes:

```
user@linux:nagios/libexec$ ./check_apc time 20 10
UPS OK - Time Left: 30 mins
```

7.11 Nagios Monitors Itself

If necessary, Nagios can even monitor itself: the included plugin, check_nagios, tests, on the one hand, whether Nagios processes are running and, on the other hand, the age of the log file nagios.log in the Nagios var directory, for example /var/nagios/nagios.log.

Despite this, the question needs to be asked: if Nagios itself is not running, then the system simply cannot perform the plugin, which in turn cannot deliver an error message. The solution to this problem consists in having two Nagios servers, each of which addresses the locally installed plugin on the opposite server, with the help of NRPE (see Chapter 10 from page 165).

If you have just one Nagios server you can also run **check_nagios** alone via cron and have the return value checked using a shell script. In this case, you take action yourself, as shown in Section 7.11.1, so that you are suitably informed of this.

The plugin has the following options:

-C */path/to/nagios* | --command=*/path/to/nagios*
: This is the complete **nagios** command, including the path (e.g., **-C /usr/local/nagios/bin/nagios**).

-F */path/to/logfile* | --filename=*/path/to/logfile*
: This is the path to where the Nagios log file **nagios.log** is saved. The file is located in the Nagios **var** directory.

-e *integer* | --expires=*integer*
: This is the maximum age of the log file. If there have been no changes to the file for longer than *integer* minutes, **check_nagios** issues a warning.

  You should make sure that this time specification is large enough: if no errors are currently occurring, Nagios will not log anything in the log file. The only reliable way to obtain a regular entry is with the parameter **retention_update_interval** in the configuration file **nagios.cfg** (see page 438). The default value is 60 minutes.

In the following example the log file should not be older than 60 minutes (this corresponds to the default *retention update interval* (see page 438):

```
user@linux:nagios/libexec$ ./check_nagios -e 60 \
    -F /var/nagios/nagios.log -C /usr/local/nagios/bin/nagios
Nagios ok: located 5 processes, status log updated 303 seconds ago
```

With currently five running Nagios processes and a log file last changed 303 seconds ago (a good five minutes), everything is in order here. If the **-e** parameter is omitted, the plugin always gives a warning.

7.11.1 Running the plugin manually with a script

The following example script demonstrates how the plugin is called outside the Nagios environment. It starts **check_nagios** initially as Nagios does and then evaluates the return value. If the status is not **0**, it sends an e-mail to the administrator **nagios-admin@example.com**, using the external **mailx** program:

```
#!/bin/bash

NAGCHK="/usr/local/nagios/libexec/check_nagios"
PARAMS="-e 60 -F /var/nagios/nagios.log -C /usr/local/nagios/bin/nagios"

INFO=`$NAGCHK $PARAMS`
STATUS=$?

case $STATUS in
   0) echo "OK : " $INFO
      ;;
   *) echo "ERROR : " $INFO | \
         /usr/bin/mailx -s "Nagios Error" nagios-admin@example.com
      ;;
esac
```

The script can be run at regular intervals via a cronjob—such as every 15 minutes. But then it will also "irritate" the administrator every quarter of an hour with an e-mail. There is certainly room for improvement in this respect—but that would go beyond the scope of this book.

7.11.2 check_nagios as a tool for CGI programs

Using the **nagios_check_command** parameter (see page 445) you can also use the plugin in the file **cgi.cfg**. If the parameter is set there, the CGI programs use the specified command to see if Nagios is operational. The test integrated into the CGI programs functions so well, however, that you do not need to go to the trouble of defining **nagios_check_command**.

7.12 Hardware Checks with LM Sensors

Modern mainboards are equipped with sensors that allow you to check the "health" of the system. In the **lm-sensors**[6] project it is also possible in Linux to query this data via I2C or SMBus (*System Management Bus*, a I2C special case).

To enable this, the kernel must have a suitable driver. Kernel 2.4.x normally requires additional modules, which are included in the software.[7] With a little luck, your distribution may include precompiled modules (e.g. SuSE). Kernel 2.6, however, already includes many drivers; here you just compile the entire branch below **I2C Hardware Sensors Chip support**.

It would take too much space here to detail the installation of the necessary modules. We will therefore only go into detail for the **check_sensors** plugin, and assume

[6] http://www.lm-sensors.nu/
[7] http://secure.netroedge.com/~lm78/download.html

that the corresponding kernel driver is already loaded as a module. Help is provided during operation with the **sensors-detect** program from the **lm-sensors** package, which does a number of tests and then tells you which modules need to be loaded. If all requirements are fulfilled, running the **sensors** program will produce an output similar to the following one, and shows that the onboard sensors are providing data:

```
user@linux:~$  sensors
fscher-i2c-0-73
Adapter: SMBus I801 adapter at 2400
Temp1/CPU:      +41.00 C
Temp2/MB:       +45.00 C
Temp3/AUX:      failed
Fan1/PS:          1440 RPM
Fan2/CPU:            0 RPM
Fan3/AUX:            0 RPM
+12V:           +11.86 V
+5V:             +5.10 V
Battery:         +3.07 V
```

The output depends on the hardware, so it will be slightly different for each computer. Here you can see, for example, the CPU and motherboard temperatures (41 and 45 degrees Celsius), the rotation speed of the fans, and the voltages on the 12- and 5-volt circuits and on the battery. Depending on the board design and the manufacturer, some details may be missing; in this example, only the fan for the power supply **FAN1/PS**[8] provides information; **Fan3/AUX** refers to an additional fan inside the computer box that, although it is running, is not recorded by the chipset.

Apart from the standard options **-h** (help function), **-v** (*verbose*), which displays the response of the sensors, and **-V**, which shows the plugin version, the plugin itself has no special options. Warning and critical limits must be set via the **lm-sensors** configuration. **check_sensors** only returns the status given by the onboard sensors:

```
user@linux:nagios/libexec$ ./check_sensors
sensor ok
```

If this is called with the **-v** option, you can see more clearly whether the test works:

```
user@linux:nagios/libexec$ ./check_sensors -v
fscher-i2c-0-73 Adapter: SMBus I801 adapter at 2400 Temp1/CPU: +40.00 C
Temp2/MB: +45.00 C Temp3/AUX: failed Fan1/PS: 1440 RPM Fan2/CPU: 0 RPM
Fan3/AUX: 0 RPM +12V: +11.86 V +5V: +5.10 V Battery: +3.07 V
sensor ok
```

[8] PS stands for *power supply*; but the names displayed can be edited in **/etc/sensors.conf**.

The output line is only wrapped for printing purposes; the plugin displays verbose information on a single line.

Alternatively you can use SNMP to access the sensor data: the NET-SNMP package (see Chapter 11.2 from page 184) provides the data delivered by **lm-sensors**, and with the SNMP plugin **check_snmp**, warning limits can also be set from Nagios. This solution is described in Section 11.3.1 from page 196.

7.13 The Dummy Plugin for Tests

For tests expected to end with a defined response, the **check_dummy** plugin can be used. it is given a return value and the desired response text as parameters, and it provides exactly these two responses as a result:

```
nagios@linux:nagios/libexec$ ./check_dummy 1 "Debugging"
WARNING: Debugging
nagios@linux:nagios/libexec$ echo $?
1
```

The output line contains the defined response, preceded by the status in text form. the return value can again be checked with **echo $?**: **1** stands for WARNING.

Alternatively you can give **check_dummy** a **0** (OK), an **2** (CRITICAL) or a **3** (UNKNOWN) as the first argument. The second argument, the response text, is optional.