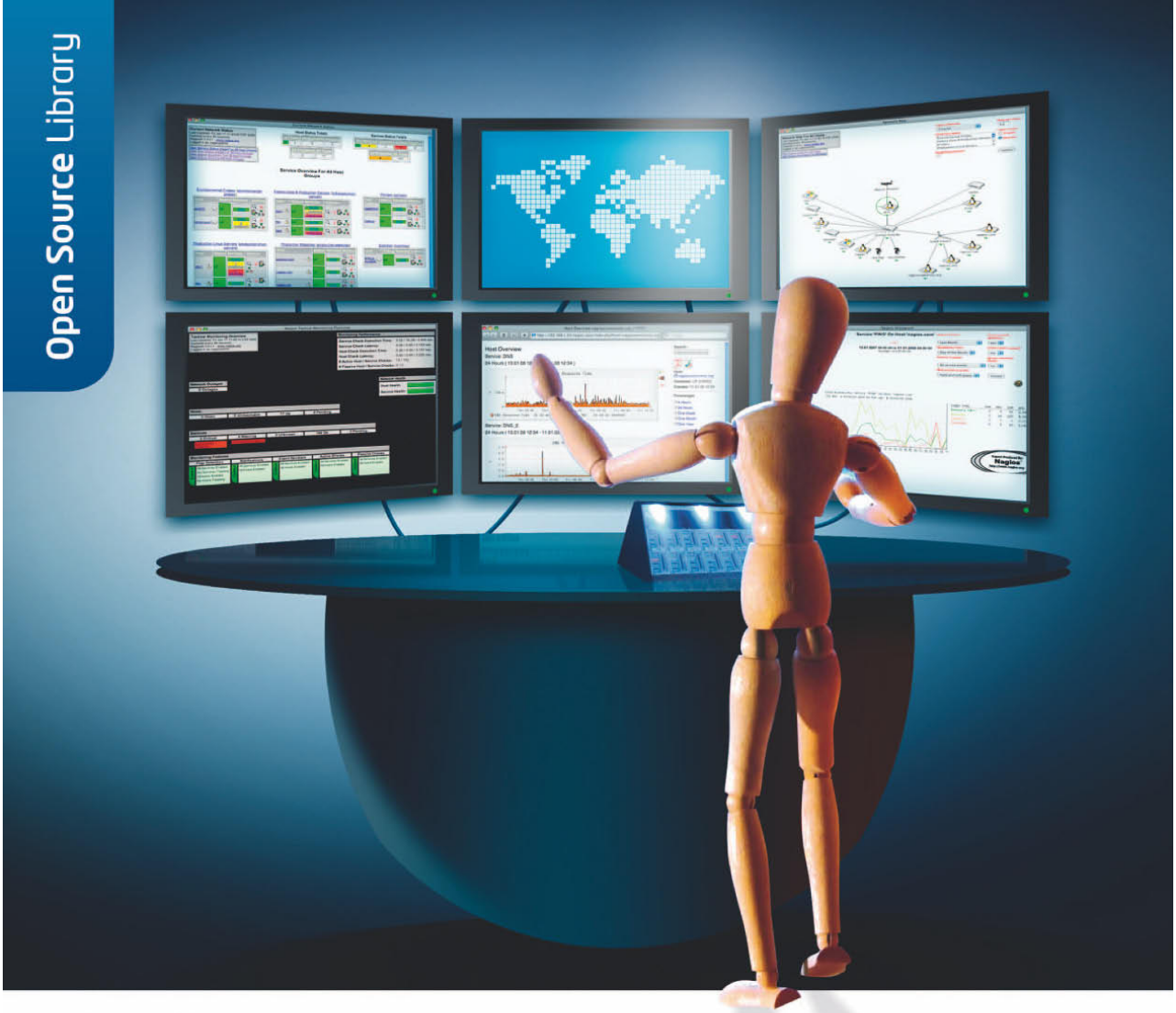
Open Source Library



Gerhard Laußer

Nagios

Das Praxisbuch

Open Source-Monitoring im Unternehmen

ADDISON-WESLEY

3. Erstellen eigener Plugins

Seinen Siegeszug hat Nagios zu einem großen Teil seinem modularen Konzept zu verdanken. Nagios allein ist ja nur das Framework, welches das Zusammenspiel von für sich autarken Komponenten steuert. Die eigentliche Aufgabe beim Monitoring, nachzuschauen, ob ein beliebiges Objekt im Netzwerk fehlerfrei funktioniert, wird von den Plugins wahrgenommen. Da es Nagios im Grunde egal ist, welcher Natur so ein Objekt ist, sei es ein Dienst, ein Prozess, ein Stück Hardware oder eine Datenbank, reicht die Definition einer Schnittstelle zwischen Nagios und Plugin. Die mehr oder weniger komplizierte Suche nach Fehlern lässt sich so ganz einfach delegieren. Diese Aufteilung spiegelt sich auch in den an der Nagios-Entwicklung beteiligten Personen wieder. Ethan Galstad programmiert das Nagios-Framework, während ein zweites Projekt unter der Leitung von Ton Voon sich um die Nagios-Plugins kümmert. Damit ist übrigens nicht die Summe aller existierenden Plugins gemeint, sondern eine Teilmenge davon, die auf Sourceforge unter der Bezeichnung *Nagios Plugins Project* zu finden ist. Üblicherweise ist diese Kollektion in jeder Nagios-Installation zu finden. Wenn im Laufe dieses Kapitels von Nagios-Plugins (mit Bindestrich) oder von „offiziellen" Plugins die Rede ist, dann sind damit die Plugins des *Nagios Plugin Project* gemeint.

Daneben ist im Laufe der Zeit ist eine Unzahl von weiteren Plugins entstanden, die freundlicherweise von den Autoren der Allgemeinheit zur Verfügung gestellt wurden. Die meisten davon findet man auf den Webseiten *Nagios Exchange*[1] und *Monitoring Exchange*[2]. Auch einige Firmen, die sich mit Nagios beschäftigen, behalten ihr Know-how nicht für sich und bieten die von ihnen geschriebenen Plugins zum Download an. Auf der Suche nach einer Monitoringlösung für spezielle Geräte und Applikationen wird man in den meisten Fällen bei Google fündig. Und nicht zuletzt kann man im *Nagios-Portal*[3] nachfragen, ob schon jemand bei sich in der Firma vor einem ähnlichen Problem stand und bereit ist, sein Plugin oder zumindest das dafür nötige Know-how weiterzugeben.

Leider kann diese Stärke von Nagios auch als Schwäche gesehen werden. Unzählige Plugins bedeutet auch unzählige schlechte Plugins. Sucht man bei *Nagios Exchange* nach einer Lösung für sein Problem, dann wird man zwar viele Treffer erhalten. Bei genauem Hinsehen stellt man aber fest, dass die Hälfte der gefundenen Plugins mit der heißen Nadel gestrickt

1 http://exchange.nagios.org
2 http://monitoring-exchange.org
3 http://www.nagios-portal.de

und schnell hochgeladen wurden, es nahezu identische Plugins unterschiedlichster Programmierqualität gibt, der Autor unbekannt verzogen ist, oder man zur Abdeckung seiner Anforderung fünf Plugins braucht, weil diese jeweils nur einen einzelnen Aspekt berücksichtigen. Es ist schade, dass es bisher nicht gelungen ist, Ordnung in dieses Chaos zu bringen. Ein Anwender hat dann die Qual der Wahl. In manchen Fällen wird man sich dann entscheiden, selbst ein maßgeschneidertes Plugin zu schreiben. Wie man dabei vorgeht und was man beachten sollte, damit auch andere von der Arbeit profitieren können, wird in diesem Kapitel beschrieben.

3.1 Grundlagen

Plugins werden vom Nagios-Prozess in seiner Rolle als Scheduler üblicherweise in festen Zeitabständen aufgerufen. In den meisten Fällen dürften diese Intervalle von 5 Minuten pro Service sein, die auf eine Minute verkürzt werden, sobald eine Fehlersituation festgestellt wurde. Da Nagios im Normalfall als Daemon unter der Benutzerkennung *nagios* läuft, müssen beim Schreiben von Plugins ein paar Regeln eingehalten werden, damit es in dieser Laufzeitumgebung nicht zu Fehlern kommt:

» Das Plugin darf keine Eingaben von STDIN erwarten. Dies würde zu einem Timeout führen, da Nagios sämtlichen Input ausschließlich über Kommandozeilenparameter oder Environmentvariablen an Plugins liefert.

» Das Plugin muss ohne die Environmentvariable *$DISPLAY* lauffähig sein. Das bedeutet, dass keine Kommandos benutzt werden dürfen, die ein graphisches Terminal voraussetzen. Wenn der Nagios- oder NRPE-Prozess, der das Plugin ausführt, nicht in einer XWindows-Umgebung läuft (was der Normalfall ist), dann schlagen solche Kommandos fehl.

» Plugins dürfen kein kontrollierendes Terminal voraussetzen. Sie werden als Backgroundprozesse ausgeführt, d. h. selbst wenn man Nagios in einem Terminalfenster startet und im Vordergrund laufen lässt, die Plugins werden vom TTY entkoppelt.

» Der Nagios- bzw. NRPE-Prozess läuft üblicherweise unter der Kennung des Nagios-Benutzers, somit auch die von ihnen gestarteten Plugins. Daher ist darauf zu achten, dass alle vom Plugin verwendeten Dateien und externen Kommandos lesbar bzw. ausführbar sind.

» Plugins sollen ihre Aufgabe so schnell wie möglich erfüllen. Als Faustregel gilt, dass eine Laufzeit von unter einer Sekunde anzustreben ist. In Umgebungen mit mehreren tausend Services und langsamen Plugins summieren sich selbst Zehntelsekunden und können für erhöhte Latenzzeiten verantwortlich sein.

3.1.1 Ein einfaches Beispiel

Zur Einführung soll ein Plugin vorgestellt werden, welches feststellt, ob der root-Benutzer an der Console eines Computers angemeldet ist und die Sitzung seit über einem Tag inaktiv war. So etwas kann ein Sicherheitsrisiko darstellen. Möglicherweise hat sich ein Systemadministrator im Rechenzentrum eingeloggt, wurde abgelenkt und hat dann vergessen, sich wieder abzumelden. Ein Unbefugter könnte dann mit der offenen root-Shell Schaden anrichten. Zunächst ruft man das **w**-Kommando auf, mit dem man sich die existierenden Sitzungen anzeigen lassen kann.

```
# w
10:15am  up 5 day(s), 19:14, 4 users, load average: 0.89, 0.95, 1.36
User     tty           login@  idle   JCPU   PCPU  what
root     console      Mon 3pm 6days     14         -sh
root     pts/2        Sat 1pm       772:19         -sh
root     pts/4        Mon 3pm 6days   2:14   2:14  sh
root     pts/5        Tue 8am  1:10     23         bash
nagios   pts/5        10:17am                     -sh
```

Wie man sieht, gibt es an der Konsole eine Session, die seit 6 Tagen inaktiv ist. Man kann davon ausgehen, dass jemand vergessen hat, sich auszuloggen. Das Plugin wird also die Ausgabe von **w** einlesen und dann zunächst nach root-Benutzern suchen. Im zweiten Schritt wird das verwendete Terminal untersucht. Lautet dessen Name *„console"* oder wie es bei Linux der Fall ist, *tty[0-9]*, dann liegt hier kein Login mit ssh oder ähnlichen Clients vor, sondern an der Hardwarekonsole des Servers selbst. Nun wird noch die Idle-Time bewertet. Dabei interessieren nur diejenigen Zeiten, die bereits in Tagen gezählt werden. Mit dem **expr**-Befehl ermittelt man per Mustervergleich die Anzahl der Tage. Ist diese grösser als 1, dann wird die Variable *sessions* hochgezählt. Diese gibt am Schluss auch den Ausschlag, ob das Resultat des Checks OK oder CRITICAL ist. Einen WARNING-Zustand liefert dieses Plugin nicht, da eine einzige offene root-Shell genauso problematisch ist wie hundert.

Listing 3.1: **check_idle_console**

```
#! /bin/sh

exec 3>&1 4>&2 >/dev/null 2>&1
TMPFILE=/tmp/check_idle_console.$$
w > $TMPFILE
sessions=0
while read user terminal rest
do
echo user $user term $terminal
  if [ "$user" = "root" ]; then
    if expr "$terminal" : "console" || \
       expr "$terminal" : "tty[0-9]"; then
      days=`expr "$rest" : '.*[ ]\([0-9]*\)days'`
      if [ -n "$days" ]; then
        if [ $days -gt 1 ]; then
          sessions=`expr $sessions + 1`
```

```
        fi
      fi
    fi
  fi
done < $TMPFILE
rm -r $TMPFILE
exec 1>&3 2>&4 3>&- 4>&-

if [ $sessions -ge 1 ]; then
  echo "$sessions idle root shells found"
  exit 1
else
  echo "no idle root shells found"
  exit 0
fi
```

Was einem bei diesem Beispiel vielleicht auffällt, sind die **exec**-Zeilen. Diese dienen dazu, eventuelle Ausgaben der einzelnen Kommandos nach */dev/null* umzuleiten. Beispielsweise gibt der **expr**-Befehl beim Mustervergleich die Anzahl der gematchten Zeichen aus. Ohne den Trick mit **exec** müsste man jedes Kommando einzeln mit *>/dev/null* zum Schweigen bringen. Im Einzelnen passiert dann folgendes:

» 3>&1 4>&2 bewirkt, dass die Filehandles 1 und 2 (stdout und stderr) dupliziert werden, so dass Handle 3 und 4 die ursprünglichen Werte zwischenspeichern.

» >/dev/null 2>&1 leiten stdout und stderr nach /dev/null um. Sollten die nun folgenden Kommandos irgendwelche Zeichen ausgeben, so werden diese im Output des Plugins nicht sichtbar sein. Erst am Schluss soll ein Resultat ausgegeben werden. Dazu müssen stdout und stderr wieder hergestellt werden.

» 1>&3 2>&4 3>&- 4>&- setzt die Filehandles 1 und 2 wieder auf ihren ursprünglichen Wert und schließt die temporären Kanäle 3 und 4. Damit werden stdout und stderr wieder auf das Terminal umgeleitet.

Weiterhin fragt man sich vielleicht, warum die Ausgabe des **w**-Kommandos in eine Datei umgeleitet wurde, anstatt sie dem **read**-Befehl direkt mit einer Pipe zuzuführen.

```
w | while read user terminal rest
do
…
done
```

Der Inhalt der while-Schleife würde in diesem Fall in einer eigenen Subshell ausgeführt, die nur eine Kopie der *session*-Variablen erhalten und diese erhöhen würde. Das Original würde weiterhin den Wert 0 behalten. Es gibt zwar Konstrukte, die dafür sorgen, dass diese Schleife im Kontext des aufrufenden Scripts ausgeführt wird, allerdings unterscheiden sich diese je nach verwendeter Shell. Die Lösung mit der temporären Datei ist daher aus Gründen der Portablilität vorzuziehen. Man darf aber nicht vergessen, sie wieder zu löschen.

Man sieht also, dass das Monitoring mit Nagios durch einfachste Mittel in seiner Funktionalität erweitert werden kann. Wie man Plugins für kompliziertere Einsatzfälle erstellt, welche formalen Regeln dabei eingehalten werden müssen und welche Hilfsmittel einem dabei zur Verfügung stehen, wird nun im Detail beschrieben.

3.1.2 Developer Guidelines

Auf der Nagios-Homepage[4] gibt es die Beschreibung Nagios Plugin API. Dort findet man die zwei wichtigsten Regeln, die ein Nagios-Plugin erfüllen muss:

» Ausgabe von mindestens einer Zeile Text

» Beendigung mit einem der Exitcodes 0, 1, 2, 3.

Mehr ist im Grunde nicht nötig, um ein beliebiges Script oder Binary als Nagios-Plugin einzusetzen. Dabei ist insbesondere unerheblich in welcher Script- oder Programmiersprache es erstellt wurde. Weiterhin ist auf dieser Seite sehr anschaulich beschrieben, wie die Performancedaten an die Ausgabe angehängt werden. Darauf wird im nächsten Abschnitt noch genauer eingegangen.

Die umfassendste Anleitung von Regeln, die bei der Erstellung von Plugins zu beachten sind, findet man in den *Nagios plug-in development guidelines* Dort wird beschrieben, welche Kommandozeilenparameter von jedem Plugin unterstützt werden sollten, welche Systemaufrufe zu vermeiden sind und wie man Timeouts handhabt. Daneben gibt es auch eine ausführliche Beschreibung des Formats, in dem Performancedaten dargestellt werden müssen.

3.1.3 Beispiele für erlaubte Ausgabeformate

Mit der Version 3 von Nagios wurde eine wichtige Neuerung eingeführt. Früher musste die gesamte Ausgabe eines Plugins in eine einzige Zeile gepackt und die Performancedaten durch ein Pipe-Symbol getrennt an diese angehängt werden. Das aktuelle Release akzeptiert und verarbeitet jetzt aber auch Ausgaben, die sich über mehrere Zeilen erstrecken. Begrenzt wird dies nur durch die maximale Puffergröße von 8k. Alles, was darüber hinausgeht, wird von Nagios einfach abgeschnitten. (Wie man den Puffer vergrößern kann, wurde im Kapitel *Installation* beschrieben). Verwendet man zusätzlich Performancedaten, dann sind einige Regeln zu beachten.

» Zeile 1 darf Ausgabetext und, durch das Pipe-Symbol abgetrennt, Performancedaten enthalten. Das entspricht dem Verhalten von Nagios 2.x.

» Die (optionalen) weiteren Zeilen 2...n dürfen ausschließlich Ausgabetext enthalten.

4 http://nagios.sourceforge.net/docs/3_0/pluginapi.html

» Die Zeile n+1 darf wieder Ausgabetext und, durch das Pipe-Symbol abgetrennt, Performancedaten enthalten.

» Die Zeilen ab n+2 dürfen nur noch Performancedaten enthalten.

TIPP

Die Ausgabe eines Plugins kann auch HTML-Code sein. Damit lassen sich Informationen in der Nagios-Weboberfläche schöner, z.B. in Tabellenform darstellen. Das Plugin check_multi kennt beispielsweise eine Option, mit der die Ergebnisse der einzelnen Checks untereinander angezeigt werden, wobei sie je nach Exitcode farblich in rot, gelb oder grün unterlegt werden. Die gesamte Ausgabe wird nur im Fenster *Service State Information* angezeigt. Bei den anderen, wie z.B. *Service Status Details For Host* erscheint nur die erste Zeile. Diese wird üblicherweise auch in Notifications versandt. Man sollte also deshalb die erste Zeile in normalem Text ausgeben und, idealerweise durch eine Option zuschaltbar, den Rest als HTML. So bleiben Mails und SMS lesbar und die Übersichtsseiten der Weboberfläche werden nicht überfrachtet. Erst wenn man direkt auf einen Service klickt, erscheint die vollständige und grafisch aufgewertete Ausgabe des Plugins. Die Voraussetzung für die Verwendung von HTML-Code ist das Setzen des Parameters *escape_html_tags=1* in der Konfigurationsdatei *cgi.cfg*, da man ansonsten den Quelltext sehen würde.

Die Kombinationsmöglichkeiten sollen nun anhand von Beispielen aufgezählt werden. Jeweils gefolgt werden sie von einer Tabelle, in der die Makros *$SERVICEOUTPUT$*, *$SERVICEPERFDATA$* und *$LONGSERVICEOUTPUT$* angeführt werden. Die rechte Spalte beinhaltet die Werte, wie sie von Nagios nach dem Aufruf des beispielhaften Plugins **check_things** gesetzt werden. Die Makros sowie die aus ihnen entstehenden Environmentvariablen sind für die weitere Verarbeitung der Checkergebnisse durch Eventhandler oder Notificationscripts wichtig.

Eine Zeile Text ohne Performancedaten

```
01 nagsrv$ check_things --warning 5 --critical 10
02 OK - Things are ok, Thing A is 2, Thing B is 4, Thing C is 0, Thing D is
4
```

Dieses und das nächste Beispiel zeigen die "klassische" Variante einer Plugin-Ausgabe, wie sie bis einschließlich Nagios 2.10 aussehen musste. Das Resultat inklusive der optionalen Performancedaten besteht aus genau einer Zeile Text.

$SERVICEOUTPUT$	Things are ok, Thing A is 2, Thing B is 4, Thing C is 0
$SERVICEPERFDATA$	
$LONGSERVICEOUTPUT$	

Eine Zeile Text mit Performancedaten

```
01 nagsrv$ check_things --warning 5 --critical 10
02 OK - Things are ok, Thing A is 2, Thing B is 4, Thing C is 0 | things=3
thing_a=2;5;10 thing_b=4;5;10 thing_c=0;5;10
```

$SERVICEOUTPUT$	Things are ok, Thing A is 2, Thing B is 4, Thing C is 0
$SERVICEPERFDATA$	things=3 thing_a=2;5;10 thing_b=4;5;10 thing_c=0;5;10
$LONGSERVICEOUTPUT$	

Mehrzeiliger Text ohne Performancedaten

```
01 nagsrv$ check_things --warning 5 --critical 10
02 OK - Things are ok
03 Thing A is 2
04 Thing B is 4
05 Thing C is 0
```

Dieses und das nächste Beispiel zeigen, wie ein Plugin die relevante Information in der ersten Zeile unterbringen und damit auch zur Version 2 von Nagios kompatibel sein kann. Die Zeilen 2–5 dienen der weiteren Verdeutlichung des Checkergebnisses und werden auf der Seite *Service Details* der Nagios Weboberfläche angezeigt. Sie stehen auch z.B. einem Notificationscript zur Verfügung und können somit versandt werden. Dabei gibt es zwei Möglichkeiten der Übergabe. Entweder per Kommandozeilenparameter, indem man in der entsprechenden Command-Definition die Makros in der hier gezeigten Notation angibt, oder indem das Script die Environmentvariablen *$NAGIOS_SERVICEOUTPUT* usw. auswertet. Zu beachten ist hierbei, im Macro *$LONGSERVICEOUTPUT$* nicht die gesamte Ausgabe des Plugins steht, sondern nur der mit Zeile 2 beginnende Abschnitt. Diese Aufteilung lässt sich bei der Konfiguration der Notifications so nutzen, dass man z.B. bei SMS-Alarme nur den kurzen *$SERVICEOUTPUT$* verschickt. Bei anderen Transportmedien wie Email oder Tickets, deren Inhalt beliebig lang sein darf, kann man hingegen *$SERVICEOUTPUT$* und *$LONGSERVICEOUTPUT$* aneinanderhängen und so die Checkergebnisse in aller Ausführlichkeit mitschicken.

$SERVICEOUTPUT$	Things are ok
$SERVICEPERFDATA$	
$LONGSERVICEOUTPUT$	Thing A is 2\nThing B is 4\nThing C is 0

Mehrzeiliger Text mit Performancedaten in der ersten Zeile

```
01 nagsrv$ check_things --warning 5 --critical 10
02 OK - Things are ok | things=3 thing_a=2;5;10 thing_b=4;5;10 thing_
c=0;5;10
03 Thing A is 2
04 Thing B is 4
05 Thing C is 0
```

$SERVICEOUTPUT$	Things are ok
$SERVICEPERFDATA$	things=3 thing_a=2;5;10 thing_b=4;5;10 thing_c=0;5;10
$LONGSERVICEOUTPUT$	Thing A is 2\nThing B is 4\nThing C is 0

Mehrzeiliger Text mit Performancedaten in der ersten und der letzten Zeile

```
01 nagsrv$ check_things --warning 5 --critical 10
02 OK - Things are ok | things=3
03 Thing A is 2
04 Thing B is 4
05 Thing C is 0 | thing_a=2;5;10 thing_b=4;5;10 thing_c=0;5;10
```

Dieses und die zwei folgenden Beispiele demonstrieren, wie Performancedaten aufgeteilt werden müssen, wenn sie mehrere Zeilen umfassen. Wichtig ist dabei nur, dass nur die erste und die letzte Zeile mit dem Ergebnistext Performancedaten enthalten dürfen.

$SERVICEOUTPUT$	Things are ok
$SERVICEPERFDATA$	things=3 thing_a=2;5;10 thing_b=4;5;10 thing_c=0;5;10
$LONGSERVICEOUTPUT$	Thing A is 2\nThing B is 4\nThing C is 0

Mehrzeiliger Text mit Performancedaten in der letzten Zeile

```
01 nagsrv$ check_things --warning 5 --critical 10
02 OK - Things are ok
03 Thing A is 2
04 Thing B is 4
05 Thing C is 0| things=3 thing_a=2;5;10 thing_b=4;5;10 thing_c=0;5;10
```

Wenn während der Ausführung eines Plugins Informationen und Performancedaten gesammelt werden und man die Ergebnisse erst am Schluss ausgibt, dann ist diese Form der Ausgabe für die Programmierung am einfachsten.

$SERVICEOUTPUT$	Things are ok
$SERVICEPERFDATA$	things=3 thing_a=2;5;10 thing_b=4;5;10 thing_c=0;5;10
$LONGSERVICEOUTPUT$	Thing A is 2\nThing B is 4\nThing C is 0

Mehrzeiliger Text mit mehrzeiligen Performancedaten

```
01 nagsrv$ check_things --warning 5 --critical 10
02 OK - Things are ok | things=3
03 Thing A is 2
04 Thing B is 4
05 Thing C is 0 | thing_a=2;5;10
07 thing_b=4;5;10
08 thing_c=0;5;10
```

Auch Performancedaten dürfen mehrzeilig ausgegeben werden, sofern sie am Ende der Ausgabe stehen. Nach dem Pipe-Symbol in Zeile 5 ist kein erläuternder Text mehr erlaubt.

$SERVICEOUTPUT$	Things are ok
$SERVICEPERFDATA$	things=3 thing_a=2;5;10 thing_b=4;5;10 thing_c=0;5;10
$LONGSERVICEOUTPUT$	Thing A is 2\nThing B is 4\nThing C is 0

3.1.4 Programmiersprachen

Wie bereits erwähnt, ist es unerheblich, in welcher Sprache ein Nagios-Plugin erstellt wurde. Wichtig ist allein die Einhaltung der Konventionen bezüglich der Ausgabe und des Exitcodes. Dennoch sollten Sie als Plugin-Autor nicht drauflos programmieren, sondern sich überlegen, wie kompliziert es für den Anwender ist, das Resultat zu installieren. Mindestens genau so wichtig ist es, dass jemand, der im Plugin einen Fehler findet, diesen idealerweise selbst beheben kann. Daher sollen nun die Vor- und Nachteile der gebräuchlichsten Script- und Programmiersprachen gegenübergestellt werden.

» Shell

Vorteile

» Einfache Probleme lassen sich mit Shellscripts sehr schnell lösen. Im einfachsten Fall ist lediglich die Ausführung eines Systemkommandos nötig, um einen Fehlerzustand im Betriebssystem oder einer Applikation festzustellen. Es muss dann nur noch mit wenigen Zeilen Code eine aussagekräftige Meldung und der Exitcode erzeugt werden, um ein gültiges Plugin zu bekommen.

» In vielen Firmen existieren bereits Prüfscripts, die zu Vor-Nagios-Zeiten erstellt wurden. Oft ist es möglich, diese durch geringfügige Änderungen in Nagios-Plugins zu verwandeln und ihre Funktionalität für das neue Monitoring-System weiterzuverwenden.

- Da eine Shell integraler Bestandteil eines jeden Unix-Systems ist, lassen sich Shell-Plugins mit, wenn überhaupt nötig, nur kleinen Anpassungen auf beliebigen Plattformen verwenden.

  Nachteile

- Bei komplexeren Scripts muss man sehr diszipliniert programmieren, damit die Übersichtlichkeit erhalten bleibt.
- Aufwändige Mustervergleiche, sowie Such- und Ersetz-Operationen sind mit Bordmitteln oft nicht möglich und müssen durch Aufrufe von sed oder awk abgearbeitet werden.
- Fehlerbehandlung ist durch den eingeschränkten Sprachumfang kompliziert und macht den Code unleserlich.

- Perl

  Vorteile

- Routinen, die spezifisch für Nagios-Plugins sind, müssen nicht selbst programmiert werden. Man kann auf das Modul *Nagios::Plugin* zurückgreifen, wodurch Funktionen wie *add_message()* oder *exit_plugin()* automatisch zur Verfügung stehen.
- Den meisten Administratoren ist die Programmierung in Perl bestens vertraut. Die Hemmschwelle, bei Unklarheiten in den Quellcode zu schauen, sinkt dadurch. Das verbreitete Know-how kann sogar dem Plugin-Entwickler insofern helfen, dass Anwender ihm fertige Bugfixes schicken, anstatt einfach nur auf Fehlermeldungen hinzuweisen.
- Ein Perl-Interpreter gehört heute zum Standardumfang der gebräuchlichen Unix-Derivate. Die Unterschiede zwischen den einzelnen Releases sind für die meisten in Perl geschriebenen Plugins unerheblich. Auch für das Windows-Betriebssystem gibt es mehrere (auch kostenlose) Interpreter, so dass die Portabilität auch auf diese Plattform ausgedehnt werden kann.
- Perl-Scripts lassen sich zur Not auch kompilieren und als ausführbare Binärdateien verteilen. Damit hat man die Möglichkeit, Perl-Plugins auf Windows-Systemen einzusetzen, ohne auf allen Rechnern einen Interpreter installieren zu müssen.
- Nagios verfügt über einen *embedded Perl Interpreter*. Verwendet man diesen, dann wird der Perl-Code des Plugins direkt im Nagios-Prozess ausgeführt, ohne dass ein eigener Prozess für einen externen Interpreter gestartet werden muss.
- Es gibt zahlreiche Module, die mächtige Funktionalität bereitstellen, so dass man sich bei der Erstellung eines Plugins auf die Programmlogik konzentrieren kann und sich nicht um Details auf einer niedrigeren technischen Ebene kümmern muss. Beispiele für diese Module sind *Net::SNMP*, *Net::LDAP*, *DBI* uvm.

Nachteile

» Keine bekannt.

» C, C++

Vorteile

» Bei kompilierten Plugins fallen die Startupzeiten für einen Kommandointerpreter weg. Sie gelten in Puncto Laufzeit als unschlagbar.

Nachteile

» Die Programmierung von Plugins in C (oder einer ähnlichen Sprache) erfordert eine entsprechende Entwicklungsumgebung. Diese ist zwar bei den meisten Linux-Systemen bereits bei einer Defaultinstallation vorhanden, jedoch gilt dies nicht für andere Unix-Plattformen oder Windows. In manchen Umgebungen ist es dem Nagios-Administrator auch nicht erlaubt, sich Compiler zu besorgen und zu installieren.

» Tauchen Bugs in einem Plugin auf, so bleibt die Fehlersuche mit Sicherheit am Entwickler hängen, da die Anwender entweder nur ein kompiliertes Binary erhalten haben oder davor zurückschrecken, C-Code zu debuggen.

» Python, PHP

Vorteile

» Die Verwendung von zwar durchaus gebräuchlichen, im Bereich der Nagios-Plugins aber eher exotischen Sprachen, hat keinen erkennbaren Vorteil gegenüber z.B. Perl. Die Motivation, Python oder PHP zu verwenden, basiert meistens auf persönlichen Präferenzen. Um auch hier wenigstens einen Pluspunkt zu vergeben, kann man sagen, dass manches Plugin seine Existenz dem Spaß am Programmieren verdankt und der Autor es ansonsten nicht geschrieben hätte.

Nachteile

» Unter Linux ist zumindest PHP weit verbreitet. Auch Python lässt sich leicht nachinstallieren. Anders sieht es aus, wenn man Nagios auf proprietären Plattformen betreibt und PHP/Python womöglich aus den Sourcen kompilieren muss. Dabei kann erheblicher Aufwand entstehen, der in keinem Verhältnis zum Gewinn steht. In der gleichen Zeit könnte man das Plugin auch in Perl umschreiben.

» Manche Firmen haben Restriktionen, was die eingesetzten Programmiersprachen für unternehmenskritische Anwendungen betrifft. Erlaubt sind nur solche, die im Standardumfang von Betriebssystemen enthalten sind, z.B. Shell und eventuell noch Perl. 3rd Party-Produkte dürfen nicht verwendet werden. Der Grund dafür ist nicht unbedingt Misstrauen, sondern das Fehlen eines einklagbaren Supports.

» Java

Vorteile

» Die Prüfung von Java-Applikationen durch ein Plugin ist u. U. nicht anders möglich als durch die clientseitige Verwendung dieser Programmiersprache. Der bekannteste Vertreter dieser Klasse von Plugins ist **check_jmx**. Java-Applicationserver geben ihr Innenleben meist nur über das JMX-Protokoll preis, das in keiner anderen Sprache implementiert wurde. Man kann aber nicht von einem echten Vorteil sprechen, wenn die Verwendung von Java zwingend erforderlich ist.

Nachteile

» Man muss eine aufgeblähte Java-Laufzeitumgebung installieren. Vorgefertigte RPM-Pakete sind i. d. R. nicht verfügbar.

» Im Administratoren-Umfeld ist Java-Know-how nur spärlich vorhanden.

» Bei jedem Plugin-Aufruf muss die Java Virtual Machine gestartet werden. Das Problem der erheblichen Startzeiten hat sich zwar mit neuen Java-Versionen gebessert, jedoch ist der Speicherverbrauch mit ca. 300MB (bei Perl ca. 80KB) sehr hoch.

Natürlich ist es jedem Plugin-Autor überlassen, die Programmiersprache seiner Wahl zu verwenden. Eine weite Verbreitung wird man aber nur erreichen, wenn man entweder etwas Einzigartiges geschrieben hat oder es den potentiellen Anwendern leicht macht. Letzteres erreicht man, indem auf die Software-Infrastruktur Rücksicht nimmt, die üblicherweise in den Firmen bereits vorhanden ist, sowie auf das in Admin-Kreisen verbreitete Programmier-Know-how. Daher lautet hier die Empfehlung: Perl.

3.1.5 Performancetuning

Wenn man Shell-Scripts verwendet, sollte man darauf achten, dass wenige externe Programme aufgerufen werden, insbesondere nicht in Schleifen. Die Korn-Shell und die BASH bieten die Möglichkeit, Pattern Matching vorzunehmen, ohne dabei **grep** aufrufen zu müssen. Ein kleines Beispiel soll demonstrieren, welchen Geschwindigkeitsgewinn man erzielen kann, wenn man die Features moderner Shells benutzt. Das Script soll die Datei */etc/mtab* auslesen und die Anzahl der NFS-Mounts zählen. Der Einfachheit halber werden diese durch einen Doppelpunkt identifiziert

```
nagsrv$ cat /etc/mtab
/dev/mapper/VolGroup00-LogVol00 / ext3 rw 0 0
proc /proc proc rw 0 0
sysfs /sys sysfs rw 0 0
devpts /dev/pts devpts rw,gid=5,mode=620 0 0
/dev/sda1 /boot ext3 rw 0 0
tmpfs /dev/shm tmpfs rw 0 0
none /proc/sys/fs/binfmt_misc binfmt_misc rw 0 0
sunrpc /var/lib/nfs/rpc_pipefs rpc_pipefs rw 0 0
dbsrv12:/opt/ibm/expc /root/expc nfs rw,addr=10.0.12.145 0 0
```

```
/dev/sdc1 /opt/ibm ext3 rw 0 0
nas.naprax.de:/mnt/md1/db2 /mnt/account-p nfs rw,addr=10.0.4.216 0 0
…
```

Mit den üblichen Unix-Hilfsmitteln im Hinterkopf würde man in den meisten Fällen so vorgehen, dass man den Dateiinhalt mit **cat** ausliest, in jeder Zeile den Doppelpunkt mit **grep** sucht und die Zählervariable mit dem „Taschenrechner" **bc** erhöht.

Listing 3.2: **slowscript**

```
#! /bin/ksh
nfsmounts=0
for line in `cat /etc/mtab`
do
  echo $line | grep ':' >/dev/null 2>&1
  if [ $? -eq 0 ]; then
    nfsmounts=`echo $nfsmounts + 1 | bc`
  fi
done
echo $nfsmounts nfsmounts
```

Ausgeführt auf einem Computer, der 130 Mounts hat, wovon 120 NFS-Mounts sind, würde dies bedeuten, dass 130 mal **grep** aufgerufen wird und 120 mal **bc**. (Es wird davon ausgegangen, dass **echo** ein Builtin-Kommando ist, ansonsten müsste man weitere 120 Aufrufe dazuzählen). Damit kommt man auf 250 Fork-Exec-Operationen, die in der Schleife ausgeführt werden. Es dürfte klar sein, dass sich das auf die Gesamtlaufzeit des Plugins negativ auswirkt.

```
$ time slowscript
120 nfsmounts

real    0m5.50s
user    0m1.78s
sys     0m5.32s
```

5 Sekunden für so eine einfache Aufgabenstellung sind viel zu hoch. Richtig verheerend können in so einem Szenario mehrfach verschachtelte Schleifen sein, da der Zeitaufwand pro Zeile exponentiell ansteigen würde. Bei gewöhnlichen Scripts, die ein Administrator für seine täglichen Aufgaben benutzt, kommt es auf ein paar Sekunden mehr oder weniger sicher nicht an. Im Falle von Nagios-Plugins jedoch sollte man sich die Mühe machen und, falls die fehlende Portabilität nicht dagegen spricht, modernere Shell-Konstrukte verwenden, um die Laufzeit so gering wie möglich zu bekommen. Das beginnt damit, dass man das **cat**-Kommando durch eine einfache Dateiumleitung mit dem Kleiner-Zeichen ersetzt. Reguläre Ausrücke kann man mit Variableninhalten vergleichen, indem man *Conditional Expressions* verwendet, die in doppelten eckigen Klammern stehen. Genauso verfährt man mit *Arithmetic Expressions* und doppelten runden Klammern. Sowohl Bash als auch KSH beherrschen diese Syntax. Letztere Shell dürfte auf den meisten Unix-Betriebsystemen vorhanden sein, so dass auch die Portabilität gewährleistet ist. Das getunte Script sieht dann so aus:

Listing 3.3: **fastscript**

```
#! /bin/ksh
nfsmounts=0
for line in $(< /etc/mtab)
do
  if [[ "$line" == *:* ]]; then
    nfsmounts=$(($nfsmounts + 1))
  fi
done
echo $nfsmounts nfsmounts
```

Tatsächlich hat sich der Aufwand gelohnt, denn die Laufzeit konnte drastisch (um das Hundertfache) verkürzt werden.

```
$ time  fastscript
120 nfsmounts

real    0m0.05s
user    0m0.05s
sys     0m0.00s
```

Verwendet man andere Scriptsprachen, wobei in den meisten Fällen Perl damit gemeint sein wird, steht einem ein größerer Sprachumfang als bei einer Shell zur Verfügung. Die Notwendigkeit, externe Programme aufrufen zu müssen, beschränkt sich dann nur noch auf spezielle Command Line Tools, die den Zustand von Geräten oder Applikationen ermitteln. Trotzdem sollte man auch hier prüfen, ob sich der Aufruf eines solchen Kommandos nicht dadurch ersetzen lässt, indem man z.B. Dateien aus dem */proc*- bzw. */sys*-Filesystem liest.

Für in C geschriebene Plugins kann man noch erwähnen, dass die Binaries nach dem Kompilieren mit dem **strip**-Kommando von Debugging-Ballast befreit werden sollten. Davon darf man sich zwar keine Wunder erhoffen, aber bei einer großen Anzahl von Services lässt sich so der durch die Plugins verbrauchte Speicher ein wenig reduzieren. Der Aufwand dafür ist minimal.

3.1.6 Superuser-Privilegien

Sind für die Ausführung eines Plugins Superuser-Privilegien nötig, dann muss dies über *sudo*-Berechtigungen geregelt werden. Das Plugin unter der *root*-Kennung laufen zu lassen wäre mehr als fahrlässig. Beispielsweise ruft das Plugin *check_mailq* das Kommando */usr/bin/mailq* auf. Dieses benötigt aber bei manchen Linux-Distributionen root-Rechte und schlägt zunächst fehl.

```
nagsrv$ check_mailq -w 10 -c 50
Program mode requires special privileges, e.g., root or TrustedUser.
CRITICAL: Error code 78 returned from /usr/bin/mailq
```

Der falsche Weg wäre, jetzt gleich das gesamte Plugin mit root-Rechten aufzurufen, also die entsprechende Nagios-Konfiguration so zu ändern, dass das Check-Command **sudo check_mailq -w 10 -c 50** lautet. Da aber das Script **check_mailq** möglicherweise mehreren Personen zugänglich ist, könnte es in böser Absicht manipuliert werden und somit Schaden anrichten. Besser ist es, nur so viele Teile des Plugins mit erhöhten Privilegien laufen zu lassen, wie minimal nötig sind. Man wird also **check_mailq** editieren und den Aufruf von **/usr/bin/mailq** in **sudo /usr/bin/mailq** ändern. Damit wird korrekte Ausführung bei minimaler Rechtevergabe erreicht. Die Regel lautet also: Plugins laufen immer unter der Kennung des Nagios-Benutzers und erlangen, falls nötig, höhere Privilegien durch wohldosierte Vergabe von sudo-Rechten. Es ist noch anzumerken, dass es ein grober Fehler ist, Plugins als root-Benutzer zu testen, da hierbei ein Berechtigungsniveau vorliegt, das im späteren Produktiveinsatz nicht mehr gegeben ist.

Neben den Programmen, die ein Plugin aufruft und die Superuser-Rechte verlangen, müssen eventuell auch Dateien gelesen werden, auf die der Nagios-Benutzer von Haus aus keine Leseberechtigung besitzt. Bevor man nun mit **chmod 644** o. ä. an eine Datei herangeht und somit den Zugriff auch für beliebige andere User erlaubt, sollte man sich die Möglichkeiten ansehen, die ACLs bieten. Damit lassen sich die Berechtigungen auf Objekte im Filesystem wesentlich feiner steuern als mit den traditionellen Mode-Bits. Insbesondere kann man damit gezielt einzelnen Benutzern Leserechte erteilen. Angenommen, ein Plugin muss auf die Datei */etc/my_application.conf* zugreifen.

```
nagsrv$ cat /etc/my_application.conf
cat: /etc/my_application.conf: Permission denied
nagsrv$ ls -l /etc/my_application.conf
-rw------- 1 root root 0 Apr 27 13:23 /etc/my_application.conf
```

Dies wird fehlschlagen, da ausschließlich der root-Benutzer Leserechte für die Datei besitzt. Unter Linux gibt es den Befehl **setfacl**, mit dem man die *Access Control List* einer Datei modifizieren kann. Man kann ihn einsetzen, um dem Benutzer nagios Leserechte zu erteilen.

```
nagsrv1# setfacl -m u:nagios:r-- /etc/my_application.conf
```

Man erkennt das Vorhandensein von ACLs daran, dass die Mode-Bits durch ein Plus-Zeichen ergänzt wurden. Nun darf der User nagios die Datei lesen.

```
[nagios@nagsrv1 tmp]$ ls -l /etc/my_application.conf
-rw-r-----+ 1 root root 61 Apr 27 13:27 /etc/my_application.conf
[nagios@nagsrv1 tmp]$ cat /etc/my_application.conf
[settings]
home_dir = /opt/my_application
start_server = yes
…
```

3.1.7 Timeouts

Gelegentlich kann es vorkommen, dass Scripts wesentlich länger laufen, als sie eigentlich sollten. Je nach Belastung des Computers, auf dem ein Plugin läuft, muss man mit Schwankungen von ein paar Sekunden rechnen, besonders dann, wenn aufwändigere Prüfungen durchgeführt werden. Eine Laufzeit von unter einer Sekunde wird als ideal angesehen, kann aber nicht immer eingehalten werden. Daher geht man in den meisten Fällen davon aus, dass ein Problem vorliegt, wenn das Script länger als 10 Sekunden läuft. Nagios selbst hat die Möglichkeit, sowohl systemweit als auch auf Service-Ebene, Timeouts zu setzen und die Ausführung des Plugins abzubrechen. Dies resultiert dann in einem *UNKNOWN*-Status. In manchen Fällen, in denen bereits eine unüblich lange Laufzeit auf Probleme hinweist, ist es aber sinnvoll, wenn das Plugin das Timeout-Handling selbst in die Hand nimmt. So kann man auch *WARNING*- oder *CRITICAL*-Zustände erreichen und die Fehlermeldung entsprechend aussagekräftig formulieren.

Bei einem Shell-Script kann man einen Timeout-Mechanismus so verwirklichen:

```
01 timeouthandler() {
02   echo "Timeout!!"
03   exit 2
04 }
05
06 timer() {
07   sleep $2
08   /usr/bin/kill -s ALRM $1  # nicht built-in kill verwenden!
09 }
10
11 trap 'timeouthandler' ALRM
12 timer $$ 10 &
13 TIMERPID=$!
14
15 for i in 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0
16 do
17   echo nummer $i
18   sleep 1
19 done
20
21 kill $TIMERPID
22
23 # hier geht es weiter zur normalen Ausgabe des Checkresultats
24 …
```

Hier wurde ersatzweise eine Schleife von 20 sleep-Befehlen verwendet, um eine lange Laufzeit zu simulieren. In der Zeile 11 wird festgelegt, dass zur Funktion *timeouthandler()* gesprungen wird, sobald das laufende Script ein *ALRM*-Signal empfängt. Dieses wird üblicherweise für Timer-Events verwendet. In Zeile 12 wird dann die Funktion *timer()* aufgerufen. Sie bekommt als Parameter die Prozess-Id des laufenden Scripts und die Anzahl der Sekunden, die dieses zu seiner Ausführung höchstens brauchen darf, bevor es gewaltsam beendet wird. Da die Zeile mit einem &-Zeichen endet, wird die *timer*-Funktion im Hintergrund, also in einem eigenen Prozess ausgeführt. Dessen *PID* wird in der Variablen *TIMER*-

PID gespeichert. In timer() selbst wird dann zuerst ein **sleep**-Kommando ausgeführt. Es bewirkt, dass erst die vereinbarte Zeit von 10 Sekunden verstreicht, bevor dann mit dem **kill**-Befehl das *ALRM*-Signal an den Hauptprozess geschickt wird. (Falls dieser tatsächlich so lange gebraucht hat und noch existiert). Parallel dazu läuft die for-Schleife. Nach zehn Sekunden trifft das *ALRM*-Signal ein und es wird sofort die Handler-Funktion *timeouthandler()* ausgeführt. Diese beendet dann das Programm mit einer entsprechenden Fehlermeldung *„Timeout!!"*. Wäre das Script hingegen vor diesen zehn Sekunden fertig geworden, wäre damit auch der *timer*-Prozess überflüssig. In Zeile 21 wird er deshalb beendet.

Wenn man ein Plugin in Perl schreibt, wird der Umgang mit Timeouts etwas einfacher. Man kann dann einfach auf den *alarm*-Systemcall von Unix zurückgreifen.

```
01 eval {
02    local $SIG{ALRM} = sub {
03        die "timeout";
04    };
05    alarm(10);
06
07    for my $i (1..20) {
08      printf "nummer %d\n", $i;
09      sleep 1;
10    }
11
12    alarm(0);
13 };
14 if ($@) {
15    if ($@ =~ /timeout/) {
16      print "Timeout!!\n";
17      exit 2;
18    }
19    else {
20      print "sonstiger Programmfehler: $@\n";
21      exit 3;
22    }
23 } else {
24    # keine Timer-Probleme
25    # weiter zur normalen Ausgabe
26 }
```

Bei Perl gibt es die nützliche Funktion *eval*, mit der sich Programmteile in einem robusten Block ausführen lassen, der alle Arten von Fehlern auffängt. Nach dem *eval*-Block kann man die Variable *$@* auswerten, die normalerweise leer bleibt, bei Problemen jedoch die Fehlerursache enthält. Zunächst wird wieder eine Handlerfunktion eingerichtet, die beim Eintreffen eines *ALRM*-Signals ausgeführt wird. Dies geschieht in Zeile 2. Als Reaktion auf das Signal wird das normalerweise tödliche *die*-Statement ausgeführt. Anstatt aber das komplette Script abzubrechen, wird hier nur die lang laufende for-Schleife unterbrochen und der *eval*-Block verlassen. Mit der alarm-Funktion in Zeile 5 wird ein Timer auf 10 Sekunden gesetzt. Dieser kümmert sich dann darum, dass nach Ablauf dieser Zeit analog zum **kill**-Kommando in der Shell-Variante das *ALRM*-Signal geschickt wird. Gibt man dieser einen String mit, dann findet man diesen in der *$@*-Variablen. Somit kann man ab Zeile 14

ermitteln, ob ein Timeout vorlag, ein anderer schwerwiegender Fehler innerhalb des *eval* passiert ist oder ob alles normal verlief. In diesem Fall wäre *$@* dann undefiniert. Auch hier muss man in Zeile 12 den Timer wieder abschalten, wenn er nicht mehr gebraucht wird, um ein dann unerwünschtes Signal zu verhindern.

In anderen Programmiersprachen gibt es ähnliche Konstrukte, um den Programmfluss bei unerwünscht langen Laufzeiten vorzeitig abzubrechen. Es ist nicht in jedem Fall nötig, sein Plugin damit auszustatten. Man könnte auch jeweils zu Beginn und am Ende des Scripts die Zeit stoppen und aus der Differenz Schlüsse über mögliche Probleme ziehen. Manchmal kann man aber davon ausgehen, dass eine Verzögerung von 10 Sekunden durch z.B. Netzwerkprobleme verursacht wird, die mit Sicherheit auch noch nach einer Minute bestehen. In so einem Fall wird man die Ausführung des Plugins abbrechen. Ein hängender **nslookup**- oder **ping**-Befehl wäre ein Beispiel für so eine Situation.

3.1.8 Compilierung von Perl-Plugins

Bei den Vorteilen von Perl für die Entwicklung von Nagios-Plugins wurde erwähnt, dass es möglich sei, diese zu kompilieren und dann als ausführbare Binärdateien zu verteilen. Dieses Vorgehen ist für folgende zwei Situationen von großem Wert:

» Wenn eine Landschaft von Windows-Servern überwacht werden soll, dann braucht man nicht auf jeder Maschine einen Perl-Interpreter installieren und warten. Es ist durchaus üblich, dass Firmenrichtlinien dies sowieso verbieten. In dem Fall reicht es, wenn man auf einem Testrechner ein EXE-File des Plugins erzeugt und dieses dann im Netzwerk verteilt.

» Es gibt Unix-Altlasten, z.B. Server, die unter Solaris 6 laufen und auf denen ein entsprechend veraltetes Perl vorhanden ist. Nach dem Motto „never touch a running system" kann es unerwünscht sein, ein Update auf z.B. Perl 5.8.x vorzunehmen. Auch eine parallele Installation eines neueren Perl-Releases kommt aus verschiedenen Gründen nicht in Frage, etwa weil der vorhandene Speicherplatz nicht ausreicht oder kein C-Compiler verfügbar ist. Auf Plugins, die ein Perl neueren Datums voraussetzen, muss man aber trotzdem nicht verzichten. Man kompiliert sie auf einem binärkompatiblen Computer zu ausführbaren Binaries und setzt diese dann auf dem alten Gerät ein.

Wie das funktioniert, soll in diesem Abschnitt gezeigt werden. Es gibt mehrere, auch kommerzielle Tools, um aus einem Perl-Script ein Binary zu machen. Davon hat sich das Perl-Modul *PAR* als am zuverlässigsten und portabelsten herausgestellt. Obendrein ist es kostenlos. Die Installation nimmt wieder der root-User mit der CPAN-Shell vor:

```
nagsrv# perl -MCPAN -e 'install PAR::Packer'
```

Mit dem Modul *PAR::Packer* wird auch das Kommando **pp** installiert. Es steht für *Par Packager* und führt die Umwandlung eines Perl-Scripts in ein standalone-Executable durch. Die dafür benötigten Module *PAR* und *PAR::Dist* werden von der CPAN-Shell automatisch

mitinstalliert. Sollte es Probleme dabei geben, dann kann man auch manuell eingreifen. Dazu wechselt man in das Verzeichnis *$HOME/.cpan/build* und führt **make install** nacheinander in den drei Modul-Unterverzeichnissen aus.

```
nagsrv# cd $HOME/.cpan/build
nagsrv# cd PAR
nagsrv# make install
nagsrv# cd ../PAR-Dist
nagsrv# make install
nagsrv# cd ../PAR-Packer
nagsrv# make install
```

Danach kann man bereits aus einem einfachen Testscript das erste Binary erzeugen.

```
nagsrv$ cat test.pl
#! /usr/bin/perl
use strict;
foreach my $i (1..3) {
   print "hello$i ";
}
print "\n";

nagsrv$ file test.pl
test.pl: perl script text executable

nagsrv$ pp -o test test.pl
nagsrv$ ./test
hello1 hello2 hello3

nagsrv$ file test
test: ELF 32-Bit LSB executable, Intel 80386, version 1 (SYSV), for GNU/
Linux 2.6.9, dynamically linked (uses shared libs), for GNU/Linux 2.6.9,
stripped
```

Unter Windows ist es wichtig, einen geeigneten Perl-Interpreter zu verwenden. Mit dem verbreiteten *ActiveState-Perl* ist es nicht möglich (Stand April 2009), *PAR* erfolgreich zu installieren. Fehlerfrei funktioniert es hingegen mit *Strawberry-Perl*[5], einem OpenSource-Perl für Windows, das auch ansonsten der freien Version von *ActiveState-Perl* vorzuziehen ist. Danach geht man wie unter Unix vor, indem man das *PAR::Packer*-Modul mit der CPAN-Shell installiert. Man startet die Kompilierung und gibt dem Binary diesmal die Dateiendung *.exe*.

```
C:\Users\nagios> pp -o test.exe test.pl
C:\Users\nagios> dir test.
C:\Users\nagios>dir test*
 Volume in Laufwerk C: hat keine Bezeichnung.
 Volumeseriennummer: D4CD-31F1
```

5 http://www.strawberryperl.com

```
 Verzeichnis von C:\Users\nagios

27.04.2009  20:54         2.702.696 test.exe
27.04.2009  20:53                92 test.pl
               2 Datei(en)      2.702.788 Bytes
               0 Verzeichnis(se),  7.825.334.272 Bytes frei*

C:\Users\nagios> .\test
hello1 hello2 hello3
```

Beim ersten Aufruf des Executables wird man sich fragen, warum die Ausführung so lange dauert. Anders als beim Compilieren eines C-Programms wird hier kein schneller Maschinencode erzeugt. Stattdessen wird des Testscript mitsamt des Perl-Interpreters sowie den zur Laufzeit benötigten Modulen in eine einzige Datei gepackt. Bei deren Ausführung wird dann der Inhalt in ein temporäres Verzeichnis extrahiert und das Script mit der so entstandenen lokalen Perl-Umgebung aufgerufen. Bei allen weiteren Läufen fällt dann zwar der Auspack-Vorgang weg, aber hohe Ausführgeschwindigkeiten darf man trotzdem nicht erwarten. Der Vorteil der PAR-Methode ist einzig, ohne eine lokale Perl-Installation im Zielsystem auszukommen.

Was noch zu beachten ist, sind Module, die nicht mit der *use*-Anweisung in das Script eingebunden werden, sondern erst zur Laufzeit mit *require* nachgeladen werden. Diese sind zunächst nicht im Executable enthalten, wodurch dieses dann während der Ausführung mit einer Fehlermeldung abbricht. Solche Module muss man bei der Kompilierung explizit angeben, damit sie in das Archiv aufgenommen werden. Häufig ist davon *PerlIO* betroffen. Der Aufruf von **pp** lautet dann in diesem Fall

```
nagsrv$ pp -M PerlIO -o test test.pl
```

3.1.9 Kommandozeilenparameter

Nahezu jedes Script wird mit Kommandozeilenparametern aufgerufen. Es ist die Aufgabe des Programmierers, diese auf Gültigkeit zu prüfen und falls dieses geklappt hat, die Parameter den entsprechenden Variablen im Code zuzuweisen. Es ist üblich, dass ein Plugin einen Mindestsatz von allgemeingültigen Parametern unterstützt. Diese lauten:

» --version (-V) gibt die Versionsnummer des Plugins aus.

» --help (-h) liefert wiederum die Versionsnummer und zusätzlich eine Übersicht über die möglichen Kommandozeilenparameter. Sollten diese nur in bestimmten Kombinationen möglich sein, dann ist es hilfreich, an dieser Stelle auch Beispiele auszugeben, wie man das Plugin mit ihnen aufrufen kann.

» --timeout (-t) legt eine maximale Zeitspanne fest, die das Plugin zu seiner Ausführung benötigen darf. Nach deren Ablauf beendet es sich mit einer entsprechenden Timeout-Meldung. Damit kann man dem systemweiten Timeout-Mechanismus von Nagios zuvorkommen, der nur eine nichtssagende Meldung liefert.

» --verbose (-v) sorgt dafür, dass während der Ausführung des Plugins zusätzliche Informationen ausgegeben werden, die z.B. beim Debuggen hilfreich sein können.

» --warming (-w) legt einen Wertebereich fest, der, falls der gemessene Wert in ihm liegt, einen Exitcode 1 zur Folge hat. Dieser Parameter ist nur dann sinnvoll, wenn das Plugin einen numerischen Messwert ermittelt.

» --critical (-c) hat dieselbe Aufgabe, nur dass in diesem Fall ein Exitcode 2 resultiert.

» --hostname (-H) kommt nur bei Plugins zum Einsatz, die einen entfernten Host überprüfen. Man übergibt dem Plugin mit diesem Parameter dessen Hostnamen oder die entsprechende IP-Adresse.

Weitere häufig benutzte Parameter sind *--port*, *--url*, *--username*, *--password* und *--community*. Wie man bemerkt, ist hier bei Verwendung der Kurzform *–p* oder *–u* keine eindeutige Zuordnung mehr möglich. Es wird daher empfohlen, defaultmäßig die Langform zu benutzen und einzelne Buchstaben nur noch aus Kompatibilitätsgründen zur Einhaltung der o. g. Konventionen zu verwenden.

Wie man in einem selbstgeschriebenen Plugin die Parameterliste auswertet, wird nunmehr anhand eines Beispiels beschrieben. Die unterstützten Optionen sollen lauten:

» --version

» --help

» --timeout mit einem optionalen Integer-Wert (Default: 10)
--timeout [5]

» --hostname mit einem String
--hostname srv.naprax.de

» --port mit einem Integer-Wert
--port 8001

» --environment mit einem Key-Value-String, wobei Mehrfachangaben möglich sind
--environment INSTDIR=/opt/plugin --environment INSTUSER=nagios

» --index mit einem Integer-Wert, wobei Mehrfachangaben möglich sind
--index 3 --index 8 --index 29 --index 87

» --coords mit drei Zahlen
--coords 129.2 10.83 19.12

In einem Perl-basierten Plugin greift man dazu auf das Modul *Getopt::Long* zurück. Es erleichtert Ihnen die Auswertung selbst komplexer Kommandozeilenparameter ungemein. Die o. g. Liste würde man dann mit folgendem Code am Programmanfang verarbeiten.

```
use Getopt::Long qw(:config no_ignore_case);
%commandline = ();
my @params = (
```

```
    'version|V',
    'help|h',
    'timeout|t:10i',
    'hostname=s',
    'port=i',
    'environment|e=s%',
    'index|i=i@',
    'coords|c=f@{3}',
);
```

Mit der Funktion *GetOptions* werden dann die Parameter, mit denen das Script aufgerufen wurde, ausgewertet. Entsprechen diese nicht den Vorgaben, die durch die Variable *@params* beschrieben wurden, dann wird eine Funktion aufgerufen, die eine Übersicht der erlaubten Parameter ausgibt. Danach bricht das Script mit dem Exitstatus *UNKNOWN* ab.

```
if (! GetOptions(\%commandline, @params)) {
  print_usage();
  exit $ERRORS{UNKNOWN};
}
```

Wenn in der Liste der Parameter *--version* gefunden wurde, dann existiert ein entsprechender Key im Hash *%commandline* und die Versionsnummer kann mit Hilfe der Version *print_revision* ausgegeben werden. Da in diesem Fall das Plugin nichts weiter zu tun hat, wird es mit dem Status *OK* beendet.

```
if (exists $commandline{version}) {
  print_revision($PROGNAME, $REVISION);
  exit $ERRORS{OK};
}
```

Das gleiche passiert, wenn das Script mit *--help* aufgerufen wurde. Diesmal wird ein Text ausgegeben, der Zweck und Benutzung des Plugins beschreibt.

```
if (exists $commandline{help}) {
  print_help();
  exit $ERRORS{OK};
}
```

Der Parameter *--timeout* kann mit und ohne eine folgende Zahl angegeben werden. Fehlt diese, dann wird stattdessen der Defaultwert 10 angenommen. Das wurde in *@params* durch die Angabe *:10i* ausgedrückt. Da somit der Key *timeout* im *%commandline*-Hash in jedem Fall definiert ist, braucht man nicht erst dessen Existenz abfragen.

```
$timeout = $commandline{timeout};
```

Mit der Angabe *=s* wurde festgelegt, dass der Hostname ein String sein muss. Dieser darf auch Leerzeichen enthalten (was im Fall eines Hostnamens allerdings keinen Sinn ergibt), in dem Fall muss man den String beim Aufruf des Scripts in Anführungszeichen einschließen, z.B. *--hostname „das ist kein gültiger Hostname“*.

```
if (exists $commandline{hostname}) {
    $hostname = $commandline{hostname};
}
```

Für die Portnummer gibt es keinen Defaultwert. Man muss also mit *exists* prüfen, ob der Parameter überhaupt angegeben wurde. Wenn ja, dann kann man den Integerwert (*=i*) einer Variablen zuweisen.

```
if (exists $commandline{port}) {
    $port = $commandline{port};
}
```

Der Parameter *--environment* kann mehrmals angegeben werden, wobei die einzelnen Werte die Form *key=value* annehmen müssen. Dies wurde mit der Vorschrift *=s%* ausgedrückt, die bewirkt, dass *$commandline{environment}* auf einen weiteren Hash verweist, der aus den Key-Value-Paaren von der Kommandozeile aufgebaut wurde.

```
if (exists $commandline{environment}) {
    foreach (keys %{$commandline{environment}}) {
      …
    }
}
```

Ähnlich geht man vor, wenn ein Parameter mehrfach angegeben werden kann, und dabei einfache Werte, z.B. Integer, annimmt. In diesem Beispiel bewirkt die Angabe von *=i@*, dass *$commandline{index}* auf ein Array zeigt, in dem die mit den einzelnen *--index* angegebenen Zahlen gespeichert wurden.

```
if (exists $commandline{index}) {
    foreach (@{$commandline{index}}) {
      …
    }
}
```

Im Falle von *--coords* wurde durch die Funktion *GetOptions* bereits überprüft, dass auch tatsächlich die durch *=f{3}* verlangten drei Zahlen angegeben wurden. Danach passiert in diesem Fall dasselbe wie im vorhergehenden. Diesmal zeigt *$commandline{coords}* auf ein Array der Länge 3, in dem die verlangten 3 Werte zu finden sind.

```
if (exists $commandline{coords}) {
    foreach (@{$commandline{coords}}) {
      …
    }
}
```

Das Modul *Getopt::Long* sorgt also dafür, dass die Kommandozeilenparameter in strukturierter Form in der Variablen *%commandline* auftauchen. Dies soll an einem Beispiel verdeutlicht werden. Ruft man das Plugin mit folgenden Optionen auf

```
--timeout 11 --hostname nagsrv.naprax.de
--environment INSTDIR=/opt/plugin  --environment INSTUSER=nagios
--index 0 --index 5 --coords 12.3 16.01 2.45
```

dann sieht der Hash *%commandline* so aus:

```
{
    'environment' => {
        'INSTUSER' => 'nagios',
        'INSTDIR' => '/opt/plugin'
    },
    'index' => [
        0,
        5
    ],
    'coords' => [
        '12.3',
        '16.01',
        '2.45'
    ],
    'timeout' => 11,
    'hostname' => 'nagsrv.naprax.de'
};
```

Bei einem Shell-basierten Plugin ist das leider nicht ganz so einfach. Die moderneren Shells kennen die Funktion *getopts*, die beim Zerlegen der Kommandozeilenparameter in handhabbare Portionen hilft, aber leider ist die Unterstützung von Langnamen nicht einheitlich implementiert bzw. gar nicht vorhanden. Etwas mehr Glück hat man mit dem externen Kommando *getopt*, jedoch gibt es auch damit keine 100%ige Portabilität. Komplexe Datentypen wie Arrays können mit diesen beiden Hilfsfunktionen gar nicht gebildet werden. Daher wird im folgenden Script eine Methode zur Interpretation von Kommandozeilenparametern vorgeschlagen, die zwar auf den ersten Blick umständlich erscheinen mag, jedoch portabel ist und mit den gängigsten Shells getestet wurde. Die Aufbereitung der Parameter *--environment*, *--index* und *--coords* findet hier in anderer Form statt, aber die jeweils angegebenen Werte stehen genauso wie bei der Perl-Variante zur weiteren Verarbeitung zur Verfügung.

Am Anfang des Shell-Scripts werden einige Variablen initialisiert, die entweder direkt die Werte der Kommandozeilenparameter aufnehmen werden oder für den Aufbau von Arrays nötig sind. Da es für den Timeout einen Defaultwert geben soll, wird dieser bereits hier vergeben. Die Variable *OPTERR* dient dazu, Fehlermeldungen zu speichern, falls die angegebenen Parameter nicht den Vorgaben entsprechen.

```
TIMEOUT=10
HOSTNAME=
INDEXARR=
INDEXINDEX=0
COORDS=
ENVINDEX=0

PROGNAME="check_beispiel"
REVISION="1.0"

OPTERR=
```

Zunächst werden wieder *stdout* und *stderr* nach */dev/null* umgeleitet, damit die Ausgaben der nachfolgenden **expr**-Kommandos unterdrückt werden. In einer while-Schleife wird dann der jeweils erste Kommandozeilenparameter *$1* der Variablen *OPTSTR* zugewiesen.

```
exec 3>&1 4>&2 >/dev/null 2>&1
while :; do
    case "$#" in 0) break; esac;
    OPTSTR="$1"
    shift
```

Danach wird geprüft, ob auf den ersten Parameter ein weiterer String folgt. Wenn ja, wird unterschieden, ob dieser mit einem Minuszeichen beginnt und somit eine eigene Option darstellt, oder nicht. In dem Fall ist er im Zusammenhang mit dem Parameter *OPTSTR* zu sehen und wird in der Variablen *OPTARG* gespeichert.

```
    if [ ! -z "$2" ]; then
        if { expr "$2" : "-[\-]*[a-z]"; }; then
            OPTARG=
        else
            OPTARG=$2
            shift
        fi
    fi
```

Die Anweisung **shift** bewirkt, dass der erste Parameter von der Liste entfernt und die folgenden um eine Position nach links rutschen. Nun beginnt eine Verzweigung, denn die unterschiedlichen Parameter müssen unterschiedlich behandelt werden. Die Zeichenfolge -- schließt die Verarbeitung der vordefinierten Parameter ab. Man verwendet sie aber nur, wenn z.B. eine Liste von Dateinamen angegeben wurde, die nach dieser Schleife abgearbeitet werden soll.

```
case "$OPTSTR" in
    --)
        break
        ;;
```

Wurde das Script mit –help aufgerufen, dann wird die Ausgabeumleitung aufgehoben und eine Funktion *print_help* aufgerufen, die den Pluginnamen, die Versionsnummer und einen Hilfetext ausgibt. Sie besteht im Grunde nur auch **echo**-Anweisungen und wird deshalb hier nicht explizit aufgeführt. Danach beendet man das Plugin mit einem *OK*-Status.

```
    --help)
        exec 1>&3 2>&4 3>&- 4>&-
        print_help $PROGNAME $REVISION
        exit 0
```

Beim Parameter *--timeout* ist nichts weiter zu tun, als nachzusehen, ob ein Wert mitgegeben wurde. Wenn ja, wird dieser der Variablen *TIMEOUT* zugewiesen, andernfalls bleibt es beim Initialisierungswert 10.

```
    --timeout)
        test ! -z "$OPTARG" && TIMEOUT=$OPTARG
        ;;
```

Da ein Hostname zwingend angegeben werden muss, wird geprüft, ob ein solcher in der Variablen *OPTARG* vorliegt. Ist diese leer, dann wird eine Fehlermeldung in *OPTERR* geschrieben. Dies würde dann am Ende der Schleife zum Abbruch und der Ausgabe dieser Meldung führen.

```
    --hostname)
        if [ ! -z "$OPTARG" ]; then
            HOSTNAME=$OPTARG
        else
            OPTERR="--hostname needs an argument"
        fi
        ;;
```

Im Gegensatz zu Perl gibt es bei einer Shell keine Hash-Strukturen. Man behilft sich deshalb mit einem Workaround, bei dem Key-Value-Pärchen in zwei unterschiedlichen Variablen gespeichert werden. Diese wiederum sind durchnummeriert, wodurch auch mehrfache Angaben von *--environment* möglich sind. Der Basisname der Variablen lautet in diesem Beispiel *ENV*. Demzufolge würden die Paare *key1=value1* und *key2=value2* in *ENV0KEY* und *ENV0VAL* sowie *ENV1KEY* und *ENV1VAL* landen. Die laufende Nummer wird in der Variablen *ENVINDEX* gespeichert. Man braucht sie später bei der Auswertung der der *ENV${LFD_NR}VAL*-Variablen.

```
--environment)
    if [ ! -z "$OPTARG" ]; then
        if { expr "$OPTARG" : '.*=.*'; }; then
            TMP="`expr "$OPTARG" : '\(.*\)='`"
            eval ENV${ENVINDEX}KEY="\$TMP"
            TMP="`expr "$OPTARG" : '.*=\(.*\)'`"
            eval ENV${ENVINDEX}VAL="\$TMP"
            ENVINDEX=`expr $ENVINDEX + 1`
        else
            OPTERR="$OPTARG is not of format key=value"
        fi
    else
        OPTERR="--environment needs an argument"
    fi
    ;;
```

In ähnlicher Form werden auch die Angaben gespeichert, die auf evtl. mehrere Parameter vom Typ *--index* folgen. Es wird wieder ein Array simuliert, indem man Variablennamen vergibt, die eine laufende Nummer enthalten. In diesem Fall lautet der Name *INDEXARR${LFD_NR}*.

```
--index)
    if [ ! -z "$OPTARG" ]; then
        eval INDEXARR$INDEXINDEX="$OPTARG"
        INDEXINDEX=`expr $INDEXINDEX + 1`
    else
        OPTERR="--index needs an argument"
    fi
    ;;
```

3D-Koordinaten, die in diesem Beispiel mit dem Parameter *--coords* angegeben werden, werden durch Leerzeichen getrennt als String in der Variablen *COORDS* gespeichert. Das wird hier so gemacht, um eine weitere Alternative vorzustellen. Natürlich könnte man auch hier mit nummerierten Variablennamen und einem simulierten Array arbeiten. Als Besonderheit gibt es die for-Schleife, mit der die nächsten drei Parameter nach *--coords* eingelesen werden

```
--coords)
    if [ ! -z "$OPTARG" ]; then
        COORDS="$COORDS $OPTARG"
        for i in 1 2
        do
            if [ -z "$1" ]; then
                OPTERR="--coords needs 3 arguments"
            else
                if { expr "$1" : "-[\-]*[a-z]"; }; then
                    OPTERR="--coords needs 3 arguments"
                else
                    COORDS="$COORDS $1"
                    shift
                fi
```

```
                fi
            done
        else
            OPTERR="--coords needs 3 arguments"
        fi
        ;;
```

Zuletzt werden Kommandozeilenparameter abgefangen, die nicht in der vorhergehenden Liste namentlich genannt wurden. Diese gelten dann als unbekannt, was mit einer entsprechenden Fehlermeldung quittiert wird.

```
        *)
            OPTERR="unknown: $OPTSTR"
            break
            ;;
    esac
```

Als letzte Anweisung in der Schleife wird geprüft, ob im vorhergehenden Schritt ein Formatfehler festgestellt wurde. Ist die Variable *OPTERR* infolge dessen nicht leer, dann wird die Schleife an dieser Stelle verlassen.

```
    if [ ! -z "$OPTERR" ]; then
        break
    fi
done
```

Jetzt kann die Ausgabeumleitung wieder aufgehoben werden, da zumindest in diesem Beispiel nur noch **echo**-Befehle folgen. Deren Ausgabe ist natürlich ausdrücklich erwünscht.

```
exec 1>&3 2>&4 3>&- 4>&-
```

Falls es zu einem Fehler gekommen ist, dann wird er an dieser Stelle ausgegeben und das Plugin mit einem *UNKNOWN*-Code beendet.

```
if [ ! -z "$OPTERR" ]; then
  echo "$OPTERR"
  exit 3
fi
```

Nachdem nun die Kommandozeilenparameter analysiert und verarbeitet wurden, stehen die entsprechenden Variablen für den eigentlichen Plugin-Code zur Verfügung.

```
echo timeout $TIMEOUT
echo hostname $HOSTNAME
CNT=0
while [ $CNT -lt $ENVINDEX ]; do
  eval ENVKEY="\$ENV${CNT}KEY"
  eval ENVVAL="\$ENV${CNT}VAL"
```

```
  echo $ENVKEY=$ENVVAL
  CNT=`expr $CNT + 1`
done
CNT=0
while [ $CNT -lt $INDEXINDEX ]; do
  eval INDEXVAL="\$INDEXARR$CNT"
  echo INDEX: $INDEXVAL
  CNT=`expr $CNT + 1`
done
echo coords $COORDS
```

Wenn man vorhat, viele Plugins auf Shell-Basis zu schreiben, sollte man einen Blick auf die Bibliothek *Shflags*[6] werfen. Sie stellt eine API bereit, mit der sich das Handling von Kommandozeilenparametern schnell und portabel programmieren lässt.

3.1.10 Das Perl-Modul Nagios::Plugin

Von Ton Voon, dem Maintainer des *Nagios Plugin Project* stammt ein Perl-Modul, welches das Programmieren von Plugins zu einem Kinderspiel macht. Zwar muss man nach wie vor den eigentlichen Code schreiben, der Applikationen und Geräte abfragt, um einen Fehler festzustellen. Wie das zu bewerkstelligen ist, weiß nur der Plugin-Autor, der ein konkretes Problem zu lösen hat. Aber viele Funktionen wie das Handling der Kommandozeilenparameter, Ausgabe von Usage- und Help-Texten, Formatierung von Performancedaten und das Berechnen des Exit-Status bekommt man vorgefertigt geliefert, wenn man das Modul *Nagios::Plugin* benutzt. Es gibt mehrere Möglichkeiten, dieses Modul zu installieren.

» Wenn man die Nagios-Plugins aus den Sourcen compiliert (wie im Kapitel Installation beschrieben), dann muss man configure noch eine weitere Option angeben.

  ```
  ./configure … --enable-perl-modules
  ```

  Dadurch wird das Perl-Modul ins Verzeichnis /usr/local/nagios/perl/lib installiert.

» Die neueste Version von *Nagios::Plugin* erhält man auf der CPAN-Seite[7]. Entweder man führt Download und Installation selbst durch, oder man geht den bequemeren Weg und benutzt die CPAN-Shell.

  ```
  Perl -MCPAN -eshell ‚install Nagios::Plugin‘
  ```

  Dabei ist jedoch zu beachten, dass das Modul in diesem Fall nicht unter *~/perl* landen wird, sondern in einem Pfad, den die lokale Perl-Installation vorgibt, üblicherweise */usr/lib/perl5*. Deshalb benötigt man für diese Methode root-Rechte.

6 http://code.google.com/p/shflags
7 http://search.cpan.org/~tonvoon/Nagios-Plugin-0.32/

» Wenn man die Nagios-Plugins nicht selber kompiliert, sondern auf ein fertiges Paket zurückgreift, hängt es von der Distribution ab, ob das Perl-Modul mitgeliefert wird. Wenn ja, dann wird es i. d. R. ebenfalls unter */usr/lib/perl5* zu finden sein.

Beginnt man nun mit dem Schreiben eines Plugins, dann muss man zuerst dafür sorgen, dass *Nagios::Plugin* zur Laufzeit geladen wird. Das erreicht man mit der folgenden Anweisung zu Beginn des Scripts.

```
use Nagios::Plugin;
```

Danach stehen einem die Funktionen aus dem Modul zur Verfügung, genauer gesagt die Definition der Klasse *Nagios::Plugin* und zahlreiche Methoden. Ebenfalls vordefiniert sind die Konstanten *OK*, *WARNING*, *CRITICAL*, *UNKNOWN* und *DEPENDENT*. Sie stehen für die numerischen Entsprechungen 0, 1, 2, 3, 4 und sollten diesen aus Gründen der leichteren Lesbarkeit des Codes vorgezogen werden. Man beginnt, indem man ein Objekt dieser Klasse erzeugt. Dazu verwendet man den Konstruktor *new* und erhält eine Variable, die einen Zeiger auf das Objekt darstellt.

```
my $plugin = Nagios::Plugin->new();
```

Dem Aufruf von *new* kann man verschiedene Parameter mitgeben, die dann insbesondere beim Aufruf des Plugins mit *--help* oder *--version* zum Tragen kommen.

» *shortname* ist ein Bezeichner, der der Ausgabe des Plugins vorangestellt wird. Fehlt diese Angabe, dann wird ersatzweise der Dateiname (in Großbuchstaben und ohne führendes „check_“) des Plugins verwendet.

» *usage* ist ein Text, der mehrere Zeilen umfassen darf und der in kurzer Form die möglichen Kommandozeilenparameter auflistet. Er wird ausgegeben, wenn ein ungültiger Parameter die falsche Anzahl von Argumenten angegeben wurde. Der Text erscheint auch, wenn das Plugin mit *--usage* aufgerufen wurde. Bei Angabe von *--help* taucht er ebenfalls im Vorspann auf.

» *version* ist ein String, der die Versionsnummer beinhaltet. Er wird für den Aufruf des Plugins mit *--version* benötigt.

» *url* ist ein Link, der auf die Homepage des Plugins verweist. Die Angabe ist optional und erscheint, wenn man *--version* oder *--help* angibt.

» *blurb* ist ein kurzer Text (1-2 Zeilen), der grob den Zweck des Plugins beschreibt. Er wird bei Angabe des Parameters *--help* als Titel ausgegeben.

» *license* beschreibt die Lizenz, unter der das Plugin veröffentlicht wurde. Fehlt dieser Parameter, dann wird stattdessen ein Hinweis auf die GPL eingesetzt. Die Lizenz erscheint bei Angabe von *--help* am Kopf der Ausgabe.

» *extra* ist ein langer Hilfetext, der ausgegeben wird, wenn das Plugin mit *--help* aufgerufen wurde. Üblicherweise beinhaltet er nähere Informationen über die Funktionsweise des Plugins und die Handhabung von Thresholds.

» *plugin* ist der Name des Plugins, der bei *--version*, *--usage* und *--help* ausgegeben wird. Lässt man diesen Parameter weg, so wird der Dateiname des Plugins verwendet.

» *timeout* gibt die maximale Laufzeit des Plugins in Sekunden an. Wird diese Zeitspanne überschritten, dann wird das Plugin vorzeitig mit einer entsprechenden Timeout-Meldung abgebrochen. Allerdings geschieht das nicht automatisch. Den dazu nötigen Aufruf der *alarm*-Funktion müssen Sie selbst schreiben. Wie das gemacht wird, sehen Sie in den folgenden Beispielen. Fehlt dieser Parameter und wurde auch auf der Kommandozeile mit *--timeout* kein Wert angegeben, dann kann das Plugin theoretisch unendlich lange Zeit laufen und wird auch nicht vorzeitig beendet.

Der Konstruktor könnte beispielsweise mit folgenden Argumenten aufgerufen werden:

```
my $plugin = Nagios::Plugin->new(
    shortname => 'fileproc',
    usage => "Usage: %s [ -v|--verbose ]  [-H <host>] [-t <timeout>]
    [ -F|--file = <the file to monitor> ]
    [ -P|--proc=<the process to monitor> ],
    [ -c|--critical=<critical threshold> ]
    [ -w|--warning=<warning threshold> ]',
    version => '1.2',
    blurb => 'This plugin monitors a file and a process
It is written inPerl using the Nagios::Plugin modules.  It accepts a pathna-
me and and a process id',
    extra => 'Please note that the -proc parameter is optional and will be
set to the default value 1 if missing.',
    plugin => 'check_beispiel',
    url => 'http://www.naprax.de/plugins',
);
```

Beim Ausführen des Plugins mit den Parametern *--version* und *--usage* würden dann (vorausgesetzt man ergänzt obigen Code um den weiter unten vorgestellten Methodenaufruf *$plugin->getopts();*) folgende Ausgaben produziert:

```
$ check_beispiel --version
check_beispiel 1.2 [http://www.naprax.de/plugins]

$ check_beispiel --usage
Usage: check_beispiel [ -v|--verbose ]  [-H <host>] [-t <timeout>]
    [ -F|--file = <the file to monitor> ]
    [ -P|--proc=<the process to monitor> ],
    [ -c|--critical=<critical threshold> ]
    [ -w|--warning=<warning threshold> ]
```

Die Kurzbezeichnung *shortname* kommt zum Vorschein, wenn man das Plugin wie vorgesehen aufruft:

```
$ check_beispiel --file /dev/null --proc 1
fileproc OK - file and proc found
```

Diese Form der Ausgabe mit einem Präfix vor dem Statuscode ist eher unüblich. Leider lässt sich *shortname* auch dann nicht unterdrücken, wenn man beim Aufruf von *new* diesem Parameter einen Leerstring oder *undef* zuweist. Es gibt aber einen Trick, wie man diesen Bezeichner trotzdem los wird. Mit einem Typeglob definiert man einfach die verantwortliche Funktion um, dann beginnt die Ausgabe wieder mit OK.

```
*Nagios::Plugin::Functions::get_shortname = sub {
    return undef; # suppress output of shortname
};
```

Nachdem nun das *Nagios::Plugin*-Objekt *$plugin* erzeugt wurde, lassen sich darauf allerhand Methoden anwenden, die dem Programmierer Arbeit abnehmen. Die erste Gruppe dient dazu, den Umgang mit Kommandozeilenparametern zu vereinfachen. (Sie verwenden intern das Perl-Modul *Getopt*, das im vorhergehenden Abschnitt behandelt wurde).

Optionsmethoden

Nach dem Aufruf des Konstruktors *new* kennt das Plugin bereits die Basisparameter *--help*, *--usage* sowie *--version*, ohne dass dafür extra Code programmiert werden müsste. Natürlich reichen diese in vielen Fällen nicht aus, um die Besonderheiten eines Plugins zu berücksichtigen. Im soeben gezeigten Beispiel kommen die Parameter *--file* und *--proc* vor. Man muss also dafür sorgen, dass diese als gültig erkannt werden. Das geht ganz einfach mit der Methode *add_arg*. Man übergibt dieser den Namen und, falls auch Argumente erwartet werden, den entsprechenden Datentyp. Das ist aber noch nicht alles, was *add_arg* kann. Wie hilfreich die Methode ist, soll an den folgenden zwei Aufrufen gezeigt werden:

```
$plugin->add_arg(
    spec => 'file=s',
    help => "--file
   The name of the file to be monitored",
    required => 1,
);
$plugin->add_arg(
    spec => 'proc=i',
    help => "--proc
   The pid of the process",
    required => 0,
    default => 1,
);
```

Diese Zeilen sorgen dafür, dass der Kommandozeilenparameter *--file* ein Argument in Form eines Strings erwartet, dass beim Aufruf des Plugins mit *--help* ein Hilfetext in die Ausgabe einfließt, der die Aufgabe des *--file*-Parameters beschreibt und dass *--file* zwingend angegeben werden muss. Der zweite Aufruf definiert einen optionalen Parameter *--proc*, dessen Argument eine Zahl sein muss. Ersatzweise wird ein Defaultwert von 1 angenommen. Man kann also mit dieser Methode sehr einfach Parameter hinzufügen oder auch wegnehmen, wobei die „Gebrauchsanleitung" beim Aufruf des Plugins mit *--help* dynamisch erzeugt wird. Wenn die Definition der erlaubten Kommandozeilenparameter vollständig ist, ruft man die Methode *getopts* auf.

```
$plugin->getopts();
```

Diese sorgt dafür, dass die tatsächlich angegebenen Parameter zur Laufzeit des Plugins evaluiert und mit den Vorgaben verglichen werden. Stimmen Anzahl und Format nicht, dann wird die Kurzanleitung ausgegeben, die beim Aufruf des Konstruktors *new* mit *usage* definiert wurde. Im Normalfall aber werden interne Flags und Variablen gesetzt, indem ihnen die Werte der Argumente von der Kommandozeile zugewiesen werden. Im weiteren Verlauf des Plugins wird dann mit Hilfe der Methode *opts* auf ihren Inhalt zugegriffen. Für jeden bekannten Kommandozeilenparameter wurde zu diesem Zweck automatisch eine gleichnamige Funktion erzeugt.

```
my $filename = $plugin->opts()->file();
my $pid = $plugins->opts()->proc();
```

Die leeren Klammern können auch weggelassen werden. Sie wurden hier nur angegeben, um zu verdeutlichen, dass es sich um Methodenaufrufe handelt. Im Beispiel sind die Variablen *$filename* und *$pid* Skalare. Wie bei der Erläuterung des Perl-Moduls Getopt gezeigt wurde, sind bei Mehrfachangabe des gleichen Parameters auch Hashes und Arrays möglich. In dem Fall würde man dann einen Zeiger auf eine Variable des entsprechenden Datentyps erhalten. Nach der Definition und dem Einlesen von Kommandozeilenparametern wird man im Plugin mit der Programmierung individuellen Codes weitermachen, der die Prüfungen vornimmt und somit die Kernaufgabe wahrnimmt. Dabei fallen in den meisten Fällen Messwerte an, deren Größe ausschlaggebend für das Endergebnis des Plugins ist.

Thresholdmethoden

Man schließt von den Messwerten auf den Zustand des überprüften Systems, indem man sie mit vordefinierten Schwellwerten vergleicht. Liegen die Messungen in einem bestimmten Intervall, so geht man davon aus, dass alles in Ordnung ist. Liegen sie außerhalb des gewünschten Bereichs, dann kann das ein Zeichen von bereits eingetretenen oder zu erwartenden Problemen sein. In dem Fall muss man noch unterscheiden, ob eine Warnung ausreicht, oder ob der Messwert so weit vom Normalwert abweicht, dass ein Critical Error

gerechtfertigt ist. Üblicherweise gibt man in so einem Fall beim Aufruf des Plugins die Parameter *--warning* und *--critical* an, aus deren Über- oder Unterschreitung ein entsprechender WARNING- oder CRITICAL-Status resultieren kann. Die dazu nötigen Berechnungen führt die Methode *check_threshold* durch, deren Rückgabewert eine der Konstanten *OK*, *WARNING* oder *CRITICAL* ist. Es gibt drei Varianten der Anwendung von *check_threshold*.

» Im einfachsten Fall wurden bereits durch die Kommandozeilenparameter *--warning* und *--critical* zwei Schwellwerte vergeben. Man muss dann nur noch den Messwert angeben, der mit ihnen verglichen werden soll.

```
my $result = $plugin->check_threshold(
    check => $messwert
);
```

Entgegen der Dokumentation des Plugins werden die Thresholds nicht automatisch gesetzt, wenn man *--warning* oder *--critical* angibt. Daran ändern auch Defaultwerte in den entsprechenden *add_arg*-Aufrufen nichts. Dies mag sich in Zukunft ändern. Bis dahin sollte man aber nach dem Aufruf von *getopts()* mit folgenden Zeilen die Erzeugung eines *Nagios::Plugin::Threshold*-Objekts erzwingen:

```
$plugin->set_thresholds(
    warning => $p->opts->warning,
    critical => $p->opts->critical,
);
```

Ohne diesen Schritt würde ein *check_threshold* nicht richtig funktionieren.

» Man kann die Schwellwerte auch direkt beim Aufruf von check_thresholds angeben. Diese hätten Vorrang vor etwaigen (von *--warning* und *--critical* stammenden) internen Thresholds.

```
my $result = $plugin->check_threshold(
    check => $messwert
    warning => ‚10:‘,
    critical => ‚5:‘,
);
```

» Mit der Methode *set_threshold* lassen sich an jeder Stelle im Programm neue Schwellwerte setzen. Dadurch lässt sich auf bestimmte Rahmenbedingungen reagieren. Beispielsweise könnte die Uhrzeit Einfluss darauf haben, wie „streng“ eine gemessene Zahlengröße bewertet wird.

```
$plugin->set_threshold(
    warning => $warning,
    critical => $critical,
);
…
```

```
my $result = $plugin->check_threshold(
    check => $messwert
);
```

Als Schwellwerte kann man Zahlen sowie Strings angeben. Letzteres ist nötig, um z.B. sogenannte *falling thresholds* auszudrücken, also Schwellwerte, bei deren **Unter**schreitung ein Alarm ausgelöst werden soll (Bsp. *„10:“*). Auch der Vergleich von Zahlenwerten mit sog. *Ranges* ist dadurch möglich. Intern werden die Schwellwerte in einem *Nagios::Plugin::Threshold*-Objekt gespeichert. Man wird *check_thresholds* selten für sich allein verwenden. Meistens benutzt man sie im Zusammenhang mit der *add_message* Methode.

Messagemethoden

Einfache Plugins sind so aufgebaut, dass eine Prüfung stattfindet, die ein Ergebnis liefert und dieses anschließend am Programmende in Textform ausgegeben wird. Manchmal werden aber auch mehrere, z. T. voneinander unabhängige Untersuchungen angestellt, Schleifen durchlaufen, wobei jedes Mal ein Einzelprüfergebnis anfällt. Diese müssten dann zusammengeführt werden, damit am Schluss ein Gesamtergebnis ausgegeben werden kann. Dafür gibt es die Methode *add_message*, mit der sich das *$plugin*-Objekt ein Zwischenergebnis zum Zwecke der späteren Verarbeitung erst einmal merken kann. Man übergibt ihr einen der Codes *OK*, *WARNING* oder *CRITICAL* und einen String, der damit auch gleich in die richtige Kategorie eingeordnet wird.

```
$plugin->add_message(WARNING, 'Process not running');
```

Damit wird erreicht, dass die Meldung sozusagen mit der Markierung WARNING versehen und für die spätere Verwendung zwischengespeichert wird. Wie bereits erwähnt, benutzt man häufig *check_thresholds* zusammen mit *add_message*. Da letztere Funktion als ersten Parameter einen Errorlevel erwartet und *check_thresholds* einen Returncode genau dieser Art liefert, kann man die beiden folgendermaßen kombinieren:

```
my $messwert = …..;
$plugin->add_message(
    $plugin->check_threshold(
        check => $messwert
    ),
    sprintf("Messwert: %d", $messwert),
);
```

Intern wird das so gehandhabt, dass im $plugin-Objekt für jeden Errorlevel ein Array existiert. Mit *add_message* wird dann der Message-String als neues Element in das entsprechende Array aufgenommen. Auf diese Weise könnte man auch vorgehen, wenn man ein Plugin schreibt, das nicht auf die Funktionen in *Nagios::Plugin* zurückgreift.

Die sich ansammelnden Meldungen müssen am Ende des Plugins zusammenhängend ausgegeben werden. Damit die wichtigsten Informationen zuerst kommen, erscheinen die Teilstücke sinnvollerweise in der Reihenfolge *CRITICAL*, *WARNING*, *OK*. Dabei hilft die Methode *check_messages*. Ruft man diese im skalaren Kontext auf, dann liefert sie einfach den Exitcode, den sie aus dem Vorhandensein von Meldungen der unterschiedlichen Schweregrade ermittelt.

```
my $exitcode = $plugin->check_messages();
```

Ruft man die Methode hingegen im Listenkontext auf, dann erhält man einen Exitcode sowie einen String, der sich aus den einzelnen Meldungen zusammensetzt und für die endgültige Ausgabe des Plugins verwendet werden kann.

```
my ($exitcode, $exitmessage) = $plugin->check_messages();
```

Ruft man *check_messages* in dieser Form auf, dann wird *$exitmessage* nach folgendem Algorithmus aufgebaut: Wenn es Messages der Kategorie *X* gibt, dann werden diese zusammengehängt, wobei jeweils ein Leerzeichen als Trennzeichen dient. Dabei wird nach der Reihenfolge *CRITICAL*, *WARNING*, *OK* vorgegangen. Es ist zu beachten, dass ausschließlich Meldungen einer Kategorie für die Ausgabe verwendet werden. Wurden mit *add_message()* sowohl eine *CRITICAL*- als auch eine *WARNING*-Message gespeichert, dann wird letztere nicht auftauchen. Anhand einiger Beispielmeldungen soll dies demonstriert werden:

```
$plugin->add_message(CRITICAL, "crit1");
$plugin->add_message(WARNING, "warn1");
$plugin->add_message(CRITICAL, "crit2");
$plugin->add_message(CRITICAL, "crit3");
$plugin->add_message(WARNING, "warn2");
$plugin->add_message(OK, "ok1");
```

Der oben gezeigte Aufruf von *check_messages* ohne weitere Parameter würde dann folgende Ergebnisse liefern:

```
$exitcode == CRITICAL
$exitmessage == 'crit1 crit2 crit3'
```

Dieses Verhalten kann man aber auch beeinflussen. *check_messages* akzeptiert dazu einige Namensparameter.

» *join* legt einen String fest, der die einzelnen Meldungen einer Kategorie voneinander trennt. Defaultmäßig wird dafür ein Leerzeichen verwendet.

» *join_all* sorgt dafür, dass nicht nur die Meldungen der Kategorie mit der größten Kritikalität ausgegeben werden, sondern sämtliche zwischengespeicherten Strings. Die einzelnen Kategorien werden wiederum durch eine Zeichenfolge getrennt, die man dem Parameter join_all mitgibt.

» *critical* ist ein Zeiger auf ein Array, mit dem man *$exitmessage* neben denen durch add_message erzeugten *CRITICAL*-Strings noch weitere Meldungen beimischen kann.

» *warning* ist ebenfalls eine Arrayreferenz, mit der man zusätzliche *WARNING*-Meldungen einfügen kann.

» *ok* dient dazu, abschließende Meldungen unterzubringen, die informellen Zwecken dienen.

Der folgende Überblick soll zeigen, wie sich die Ausgabe verändert, wenn diese Parameter zum Einsatz kommen.

```
$plugin->check_messages(
    join => ', '
);
crit1, crit2, crit3

$plugin->check_messages(
    join => ', '
    join_all => '; ',
);
crit1, crit2, crit3; warn1, warn2; ok1

$plugin->check_messages(
    join => ', '
    join_all => '; ',
    warning => ['warna', 'warnb'],
    critical => ['crita'],
);
crita, crit1, crit2, crit3; warna, warnb, warn1, warn2; ok1
```

Wie diese Methode im Plugin eingesetzt wird, sieht man später im Abschnitt *Exitmethoden*.

Performancedatenmethoden

Ähnlich geht man vor, wenn Messwerte am Schluss als Performancedaten an die Ausgabe des Plugins angehängt werden sollen. Auch hier steht eine Methode zur Verfügung, die sowohl die Zwischenspeicherung übernimmt als auch bei der korrekten Formatierung der Performancedaten behilflich ist.

```
$plugin->add_perfdata(
    label => 'filesize',
    value => $messwert,
    uom => 'B',
    threshold => $plugin->threshold(),
);
```

Die Bedeutung der Parameter *label* und *value* dürfte klar sein. Laut *Plugin Development Guidelines* sind dies die Mindestangaben für ein gültiges Performancedatum. Optional ist dagegen die *Unit of Measurement*, die die Einheit ausdrückt, in der *$messwert* gemessen wurde. In diesem Fall ist dies *B* und steht für *Bytes*. Ebenfalls optional ist hier die Angabe

von Thresholds. Es wird ein Objekt des Typs *Nagios::Plugin::Threshold* erwartet. Dieses erhält man durch Aufruf der Methode *threshold()*, allerdings liefert diese nur dann ein Ergebnis, wenn zuvor mit *set_threshold()* dieses (globale) Objekt erzeugt wurde. Bei einem angenommenen Messwert von 10 und Critical- und Warningschwellwerten von 20 und 30 würde dann in diesem Beispiel folgende Zeichenkette Bestandteil der Plugin-Ausgabe sein:

```
filesize=10B;20;30
```

Am Schluss des Plugins müssen die gesammelten Zwischenergebnisse und Performancedaten noch zusammengefasst und der endgültige Exitcode ermittelt werden.

Exitmethoden

Die letzten beiden Befehle in einem Plugin sind üblicherweise die Ausgabe des Checkergebnisses in Form eines Textes und das Verlassen des Programms mit einem Exitcode. Die Methode *nagios_exit()* fasst sie zu einem einzigen Statement zusammen.

```
$plugin->nagios_exit(OK, 'no errors detected');
```

Wie man sieht, besteht die Parameterliste aus einem Exitcode und einem Text. Es liegt also nahe, hier einfach die Ausgabe von *check_messages* einzusetzen.

```
$plugin->nagios_exit(
    $plugin->check_messages()
);
```

Daneben gibt es noch die Alternative *nagios_die()*. Sie unterscheidet sich von *nagios_exit()* dadurch, dass die Liste der Parameter umgedreht ist, wobei der nunmehr zweite Parameter für den Exitcode optional ist. Fehlt dieser, dann wird *UNKNOWN* dafür verwendet. Diese Funktion kommt dann zum Einsatz, wenn ein Plugin nicht in der Lage ist, festzustellen, ob ein Fehlerfall vorliegt oder nicht. In dem Fall wird die Programmausführung mit *nagios_die()* kurzerhand vorzeitig abgebrochen.

Sonstiges

Auch wenn man das Plugin mit dem standardmäßig vorhandenen Kommandozeilenparameter *--timeout* aufgerufen hat, muss man sich selbst darum kümmern, dass ein Timer gestartet wird, der nach Ablauf der vorgesehenen Zeit den normalen Programmfluss unterbricht. Dies macht man mit folgender Anweisung, die man am besten unmittelbar nach dem Aufruf von *getopts()* platziert.

```
alarm $plugin->opts->timeout;
```

Wenn die Zeit abgelaufen ist, wird zu einem Signalhandler gesprungen, um dessen Initialisierung sich *getopts()* gekümmert hat. Der Handler gibt dann noch eine entsprechende Meldung aus und beendet das Plugin mit dem Status UNKNOWN.

```
UNKNOWN - plugin timed out (timeout 15s)
```

Wem diese Meldung nicht gefällt, der kann auch seinen eigenen Signalhandler definieren.

```
$SIG{ALRM} = sub {
  $plugin->nagios_die("mein eigener Text");
};
```

Damit lässt sich dann die Ausgabe im Timeout-Fall den eigenen Bedürfnissen anpassen.

Extra-opts

Ein sehr praktisches Feature, welches mit dem Release 1.4.12 der Nagios-Plugins eingeführt wurde, sind die Extra-Opts. Damit ist es möglich, Parameter von der Kommandozeile in eine Datei zu verlagern. Für den Aufruf eines Plugin bedeutet dies, dass nur noch ein einziger Parameter *--extra-opts* nötig ist, der dann bewirkt, dass alle weiteren Parameter aus einer Datei gelesen werden. Dadurch ist es möglich, sensible Daten, die man z.B. mit *--password* mitgibt und die dadurch in der Prozessliste zu sehen sind, in einer speziellen Datei zu verstecken. Da die Extra-Opts auch vom Modul *Nagios::Plugin* unterstützt werden, soll hier näher auf sie eingegangen werden. Anhand eines Beispiels wird demonstriert, wie man einem Plugin **check_beispiel** die Parameter *--file* und *--proc* mit Hilfe einer Datei übergibt. Der traditionelle Weg des Plugin-Aufrufs lautet:

```
check_beispiel --file /tmp/testfile --proc 1
```

Steckt man die Parameter nun in eine Datei namens *myparams.ini* mit folgendem Format

```
[check_beispiel]
file=/tmp/testfile
proc=1
```

dann verkürzt sich der Aufruf des Plugins auf:

```
check_beispiel --extra-opts @myparams.ini
```

So eine Optionsdatei ist in Sektionen (in diesem Fall nur eine einzige, *[check_beispiel]*) unterteilt, wie man es von INI-Dateien unter Windows kennt. Sektionen werden durch einen Bezeichner in eckigen Klammern eingeleitet, danach folgen Key-Value-Paare bis zur nächsten Sektion oder dem Dateiende. Im Beispiel wurde *--extra-opts* ein Dateiname mitgegeben. Man kann diesen auch weglassen, dann sucht sich das Plugin selbständig eine Parameterdatei. Dabei wird folgende Liste von Pfaden durchprobiert, bis ein Treffer erfolgt.

```
/etc/nagios/plugins.ini
/usr/local/nagios/etc/plugins.ini
/usr/local/etc/nagios/plugins.ini
/etc/opt/nagios/plugins.ini
/etc/nagios-plugins.ini
/usr/local/etc/nagios-plugins.ini
/etc/opt/nagios-plugins.ini
```

In den folgenden Beispielen wird davon ausgegangen, dass */usr/local/nagios/etc/plugins.ini* die Konfigurationsdatei der Wahl ist, in der die verschiedenen Parameter zentral gespeichert sind. Daher wird ab hier die Angabe von *@myparams.ini* wegfallen. Man sollte aber im Hinterkopf behalten, dass es jederzeit möglich ist, Parameter in beliebige Dateien auszulagern. Im INI-File wird dann nach einer Sektion gesucht, deren Name dem Plugin-Namen entspricht. Dieser ist durch den Parameter *plugin* im Aufruf von *new* oder ersatzweise durch den Dateinamen des Plugins vorgegeben.

In der Regel wird man mehrere Services in Nagios definiert haben, die auf demselben Plugin aufbauen. Sollten hier unterschiedliche Parameterlisten zum Einsatz kommen, dann ist es auch möglich, dies durch den direkten Aufruf einer bestimmten Sektion zu berücksichtigen. Man erweitert dazu die Datei */usr/local/nagios/etc/plugins.ini*

```
[testfile]
file=/tmp/testfile
proc=1

[prodfile]
file=/tmp/prodfile
proc=29
```

Jetzt kann man *check_beispiel* auf zwei Arten aufrufen und dabei die gewünschte Sektion angeben:

```
check_beispiel  --extra-opts  testfile

check_beispiel  --extra-opts  prodfile
```

In der herkömmlichen, ausführlichen Schreibweise hätten die Aufrufe so ausgesehen:

```
check_beispiel  --file  /tmp/testfile  --proc  1

check_beispiel  --file  /tmp/prodfile  --proc  29
```

Selbstverständlich steht diese Art des Aufrufs nach wie vor als dritte Variante zur Verfügung. Wenn die entsprechende Servicedefinition keine Leerzeichen enthält, dann bietet es sich an, diese als Sektionsbezeichnung zu verwenden. Man könnte dann die Konfiguration folgendermaßen aufbauen:

```
define service {
    service_description    app_beispiel_check_testfile
    …
    check_command   check_beispiel_extra
}
define service {
    service_description    app_beispiel_check_prodfile
    …
    check_command   check_beispiel_extra
}
define command {
    command_name    check_beispiel_extra
    command_line    $USER2$/check_beispiel \
                    --extra-opts $SERVICEDESC$
}
```

Die entsprechenden Abschnitte in der INI-Datei, die herangezogen wird, sehen dann folgendermaßen aus:

```
[app_beispiel_check_testfile]
file=/tmp/testfile
proc=1

[app_beispiel_check_prodfile]
file=/tmp/prodfile
proc=29
```

Wie bereits erwähnt, funktioniert das nicht, wenn die Service- und somit die Sektionsbezeichnung Leerzeichen enthält.

Der Parameter *--extra-opts* kann auch mehrfach angegeben werden. Damit ist es möglich, Daten aus mehreren Sektionen zu mischen. Angenommen, **check_beispiel** soll nun grundsätzlich mit dem Parameter *--timeout 20* aufgerufen werden. Dann könnte man diesen in einer von allen gemeinsam genutzten Sektion unterbringen, wohingegen die individuellen Parameter in eigenen Sektionen stehen.

```
[check_beispiel]
timeout=20

[app_beispiel_check_testfile]
file=/tmp/testfile
proc=1

[app_beispiel_check_prodfile]
file=/tmp/prodfile
proc=29
```

In diesem Fall reicht dann die Angabe von *--extra-opts* ohne Argument, um den Timeout-Eintrag zu finden. Für die restlichen Parameter ergänzt man *--extra-opts* je nach Einzelfall mit dem entsprechenden Sektionsnamen.

```
check_beispiel --extra-opts --extra-opts app_beispiel_check_prodfile
```

Der äquivalente ausführliche Aufruf würde lauten:

```
check_beispiel --timeout 20 --file /tmp/prodfile --proc 29
```

Es ist auch möglich, ein Plugin so aufzurufen, dass ein Teil der Parameter über den Weg einer INI-Datei und der andere Teil auf herkömmlichen Weg durch Angabe auf der Kommandozeile vorliegt. In dem Fall ist es nicht so, dass z.B. ein *--file* auf der Kommandozeile ein *file=* in der Datei überschreibt, wie man meinen könnte. Tatsächlich sieht es aus der Sicht des Plugin-Codes so aus, als stünden beide Parameter auf der Kommandozeile. Dies ist zu berücksichtigen, wenn beispielsweise Mehrfachangaben möglich sind.

Der wichtigste Grund, die Extra-Opts einzusetzen ist sicher der, dass man Plugins nicht mehr mit sensiblen Login-Daten (*--username, --password*) aufrufen möchte. Diese wären dann für jedermann sichtbar, der Zugang zum Nagios-Server hat und dort das **ps**-Kommando ausführt. Daneben ist es praktisch, sich häufig ändernde Anforderungen durch Anpassung der INI-Files zu erfüllen, ohne die Nagios-Konfiguration anfassen zu müssen. Dies ermöglicht es auch beliebigen Betriebsteams, eigenständig Änderungen vorzunehmen, ohne dass das Nagios-Team tätig werden muss. Man könnte z.B. mittels ACLs einem Serverbetreiber das Editieren eines für ihn reservierten INI-Files gestatten, in dem er die Liste der z.B. von **check_disk** überprüften Filesysteme ändern kann. Nicht zuletzt können die INI-Dateien auch dazu beitragen, die Nagios-Konfiguration zu verschlanken.

3.1.11 Der Nagios Embedded Perl Interpreter

Wenn man bei der Installation von Nagios die Option *--enable-embedded-perl* gewählt hat, dann erhält man eine besondere Variante von Nagios, genannt ePN (*embedded Perl Nagios*). Diese unterscheidet sich von der Standardausführung dadurch, dass ein Perl-Interpreter im Nagios-Core enthalten ist. Dadurch ist Nagios in der Lage, Perl-Code unmittelbar auszuführen, ohne erst einen externen Interpreter */usr/bin/perl* starten zu müssen. Zum Einsatz kommt diese Methode bei in Perl geschriebenen Plugins, Event Handlern und Notification Scripts. Speziell bei Ersteren kann ein deutlicher Performancegewinn erzielt werden, da sie wesentlich häufiger ausgeführt werden. ePN erkennt an der ersten Zeile, in welcher Sprache ein Plugin geschrieben wurde. Taucht dort das bekannte *#! /usr/bin/perl* auf, so ist das Plugin ein potentieller Kandidat für den internen Perl-Interpreter. (Auch andere Pfade sind möglich, etwa *#! /usr/local/bin/perl*). Ob dann der entsprechende Perl-Interpreter in einem eigenen Prozess gestartet wird oder ob der Plugin-Code von Nagios eingelesen und mit dem internen Interpreter ausgeführt wird, hängt von mehreren Konfigurationseinstellungen ab. Die wichtigste befindet sich in der Datei *nagios.cfg*.

```
enable_embedded_perl=1
```

Diese Option entscheidet systemweit, ob die Verwendung des embedded Perl Interpreter überhaupt gewünscht ist. Gibt man hier 0 an, dann wird er sozusagen auskonfiguriert und

alle Plugins werden auf die herkömmliche Art ausgeführt. Man kann die Option aber getrost auf aktiv setzen, denn es kommen noch weitere Regeln zum Zuge. Hat ePN also erkannt, dass es sich um ein Perl-Plugin handelt, dann werden dessen erste 10 Zeilen untersucht. Dort kann man mittels eines speziellen Kommentars angeben, ob es mit dem embedded Perl Interpreter ausgeführt werden soll oder nicht. Der Kommentar muss in folgendem Format vorliegen:

```
# nagios: +epn
```

Mit dieser Zeile teilt man Nagios mit, dass ausdrücklich gewünscht ist, das Plugin vom embedded Interpreter ausführen zu lassen. Das wird dann der Fall sein, wenn durch ausführliche Tests nachgewiesen wurde, dass dadurch keine Probleme zu erwarten sind. Im Zweifelsfall, oder wenn bekannt ist, dass ein Plugin die Voraussetzungen für einen Einsatz mit ePN nicht erfüllt, schaltet man die Sonderbehandlung besser ab:

```
# nagios: -epn
```

Dadurch sorgt man dafür, dass der in der ersten Zeile vorgefundene Perl-Interpreter gestartet wird, der dann in einem eigenen Prozess den Plugin-Code ausführt. Fehlt diese Angabe komplett, dann entscheidet ein weiterer Parameter in *nagios.cfg*, wie weiter verfahren wird.

```
use_embedded_perl_implicitly=1
```

Setzt man den Parameter auf 1, dann wird ein Plugin im Zweifelsfall vom embedded Perl Interpreter ausgeführt. Auf welchem Wege Nagios zu einer Entscheidung findet, verdeutlicht das folgende Ablaufdiagramm.

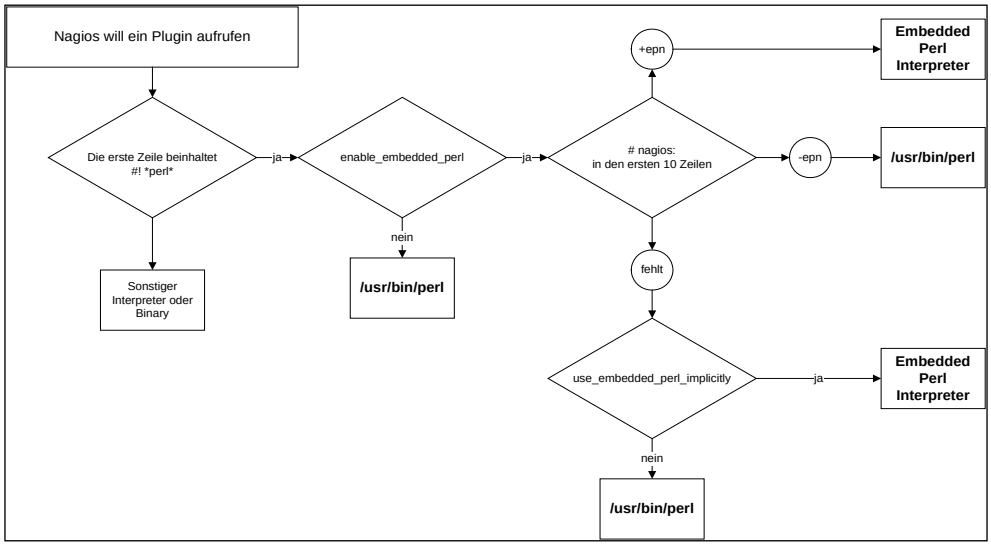


Abbildung 3.1: **Entscheidung für einen externen Perl-Interpreter oder embedded Perl**

Worauf ist nun zu achten, wenn nicht */usr/bin/perl* o.ä., sondern der Nagios-Prozess selbst ein Perl-Plugin ausführt. Als Programmierer muss man sich an einige Vorschriften[8] halten, wenn man ein ePN-fähiges Plugin programmiert. Diese sind auf der Webseite von Nagios genau beschrieben, deshalb sollen sie hier nicht noch einmal wiederholt werden. Deshalb wird davon ausgegangen, dass ein neues Plugin bereits erstellt wurde. Vor dem produktiven Einsatz sollte es dann in einer simulierten Umgebung auf seine ePN-Kompatibilität getestet werden. Dazu gibt es im *contrib*-Verzeichnis der Nagios-Sourcen das Programm **new_mini_epn**. Zumindest in der Version 3.10 von Nagios wird dieses nicht erstellt, wenn man **make contrib** aufruft. Man muss die Kompilierung und Installation von Hand ausführen.

```
nagsrv$ cd contrib
nagsrv$ make; make install
```

Danach ruft man **new_mini_epn** auf, das im letzten Schritt nach */usr/local/nagios/bin* kopiert wurde. Das Programm verlangt die Eingabe einer Kommandozeile, also dem Plugin mit dessen vollständiger Parameterliste.

```
nagsrv$ new_mini_epn
plugin command line:
```

Dabei muss man darauf achten, den vollständigen Pfad zum Plugin anzugeben, damit die Simulation dem realen Verhalten von Nagios entspricht. (Dort gibt man ja bei der Command-Definition i. d. R. *$USERx$/plugin* an). Danach liest **new_mini_epn** den Inhalt der Datei ein, führt ihn mit Hilfe des eingebauten Perl-Interpreters aus und zeigt dann den Exitcode und den Output des Plugins an.

```
nagsrv$ new_mini_epn
plugin command line: /usr/local/nagios/locallibexec/check_fs_ping --path /
mnt/missing
embedded perl plugin return code and output was: 2 & CRITICAL - /mnt/missing
does not exist | /mnt/missing=0.101345s;3;4
plugin command line:
```

Als Plugin-Entwickler sollte man nun alle möglichen Szenarien durchprobieren. Das Ausbleiben einer Fehlermeldung bedeutet aber noch nicht, dass sich das Plugin über einen längeren Zeitraum im realen Einsatz korrekt verhalten wird. Beim Test mit **new_mini_epn** treten nur die offensichtlichsten Fehler zutage. Man sollte also sein Plugin in einer Nagios-Umgebung eine Weile laufen lassen und beobachten, ob unerwartete Fehlermeldungen auftauchen oder ob Memory Leaks den Speicherbedarf des Nagios-Prozesses wachsen lassen.

Die Meinungen über ePN gehen auseinander. Aufgrund der Erfahrungen, von denen Anwender berichteten, kann der Einsatz von ePN weder empfohlen noch davon abgeraten werden. Manche sind begeistert vom Performancegewinn, andere gerieten in Teufels Küche. Man muss es

8 http://nagios.sourceforge.net/docs/3_0/epnplugins.html

einfach ausprobieren. Grundsätzlich sollte Nagios aber mit Embedded Perl kompiliert werden. Danach kann man mit *use_embedded_perl_implicitly=0* Schritt für Schritt einzelne Plugins mit *#epn+* vom internen Interpreter ausführen lassen und ihr Verhalten beobachten.

3.2 Ein nützliches Beispiel

Häufig werden große Datenbestände auf NAS-Filern abgelegt und in sogenannten Shares bereitgestellt. Server mounten sich dann diese Shares (im Unix-Bereich üblicherweise mit NFS) und greifen auf die Daten zu. Dadurch entsteht allerdings eine Abhängigkeit von externen Geräten. Nicht nur der Filer selbst kann ausfallen, auch das Netzwerk ist eine potentielle Fehlerquelle. Aber es muss noch nicht einmal zum Abbruch der Verbindung zwischen Server und Filer kommen. Bei parallelem Zugriff durch viele NFS-Clients kann es vorkommen, dass die Platten im NAS-Storage an die Grenzen ihrer Leistungsfähigkeit stoßen. Daten können dann nicht mehr in der gewünschten Geschwindigkeit an die Clients ausgeliefert werden. Auch Probleme im Netzwerk können den Datentransfer durch viele Retransmits und verlorene Pakete erheblich bremsen. In dieser Situation häufen sich i. d. R. die Beschwerden über langsam reagierende Applikationen. Bei der Fehlersuche wird man dann feststellen, dass ein Webserver Seiten nur mit Verzögerung ausliefert, danach stellt man fest, dass die Datenbank träge reagiert und bis man schließlich beim Filesystem-IO angelangt ist, kann eine Weile vergehen. Dabei hätte man womöglich lange vorher bemerken können, dass Dateizugriffszeiten ungewöhnlich lange dauern.

Ein anderes Szenario ist der Ausfall einer Netzwerkkomponente zwischen Server und Filer. Die gemounteten Filesysteme reagieren dann gar nicht mehr. Schlimmer noch, Prozesse, die die dort liegenden Dateien verarbeiten wollen, bleiben einfach stehen und lassen sich im Falle von Hard-Mounts noch nicht einmal mehr mit dem kill-Kommando beenden. Das ist natürlich ein beabsichtigtes Verhalten, damit nach der Behebung der Störung die Prozesse weiterlaufen, als sei nichts geschehen. Dennoch sollte man bei so einem Stillstand unbedingt alarmiert werden. Als Beispiel soll der folgende Mount dienen:

```
nagsrv# df /storage
Filesystem           1K-blocks      Used Available Use% Mounted on
nas.naprax.de:/mnt/md1/db2
                     961076704 842225600  70031168  93% /storage
```

Auf dem Server *nagsrv* gibt es also ein Verzeichnis */storage*, in welchem nicht-lokale Daten liegen. Tritt nun eine Störung auf, z.B. versehentliches Löschen der Berechtigung, Netzwerkausfall oder gar ein Absturz des Filers *nas.naprax.de*, dann sind Dateioperationen im Fileystem */storage* nicht mehr möglich. Ein Prozess, der einen Systemaufruf wie *open*, *read*, *stat* o. ä. ausführen möchte, wird vom Kernel blockiert und bleibt einfach hängen. Gleichzeitig erscheinen in der Messages-Datei des NFS-Clients *nagsrv* Meldungen der folgenden Form:

```
Apr 21 15:32:41 nagsrv kernel: nfs: server nas.naprax.de not responding,
still trying
```

Allerdings bemerkt der Kernel erst dann, dass der NFS-Server ein Problem hat, wenn ein Prozess auf die Daten im gemounteten Filesystem zugreifen will. Vorher fällt nicht auf, dass es zu einer Blockade kommen wird. Falls beispielsweise ein Cronjob zu jeder vollen Stunde eine Datei aus einem lokalen Verzeichnis nach */storage* kopiert und 5 Minuten später zieht jemand ein Netzwerkkabel aus dem Filer, dann verstreicht fast eine ganze Stunde zwischen dem Eintritt der Störung und deren Auswirkungen. Diese Zeit könnte man nutzen, um den Fehler zu beseitigen und die Ausführung des Cronjobs würde dann in keiner Weise beeinträchtigt sein. Das Ziel dieses Kapitels ist daher, ein Plugin zu schreiben, welches solche Fehlersituationen frühzeitig erkennt.

3.2.1 Die Spezifikation

Bevor man sich an die Erstellung eines Plugins macht, sollte der Funktionsumfang klar definiert werden.

» Das Plugin greift auf ein beliebiges Verzeichnis oder eine Datei zu, welche sinnvollerweise unterhalb eines nicht-lokalen Mountpoints liegen. Den Namen gibt man mit dem Kommandozeilenparameter *--path* an. Mehrfachnennung soll möglich sein. Besteht ein Wert hinter *--path* aus durch Kommas getrennten Verzeichnissen, so sollen sie behandelt werden, als seien sie einzeln angegeben worden. Beispielsweise ist *--path /mnt/data1,/mnt/data2* gleichbedeutend mit *--path /mnt/data1 --path /mnt/data2*.

» Die Zeit, die gebraucht wird, um Informationen über das Verzeichnis oder die Datei einzuholen, soll gestoppt werden.

» Existiert dieses Verzeichnis oder die Datei nicht, soll daraus ein CRITICAL-Status resultieren.

» Dauert das Einholen der Information länger als die durch *--warning* und *--critical* festgelegte Anzahl von Sekunden, soll daraus ein entsprechender Status resultieren.

» Bleibt ein Dateisystemzugriff hängen (z.B. wegen eines der soeben beschriebenen Fehlerszenarien), dann soll nach Ablauf eines Timeouts das Plugin mit CRITICAL-Status beendet werden. Die maximale Wartezeit gibt man mit dem Parameter *--timeout* vor.

» Es sollen Performancedaten ausgegeben werden, die als Reaktionszeit eines Filesystems interpretiert werden können.

» Das Plugin soll in Perl geschrieben werden.

Es wurde bereits angesprochen, dass ein Prozess beim Zugriff auf ein Verzeichnis, das mit NFS gemountet wurde, hängen bleibt, wenn der NFS-Server nicht mehr richtig funktioniert. Das gilt natürlich auch für Nagios-Plugins. Erschwerend kommt hinzu, dass der Prozess in diesem Zustand auch nicht beendet werden kann. Selbst ein **kill -9** bleibt wirkungslos. Das hat dann zur Folge, dass Nagios je nach Konfiguration alle 5 Minuten einen neuen Plugin-Prozess startet und diesen nach Ablauf der Timeout-Zeit zu beenden versucht, er aber

trotzdem in der Prozessliste verbleibt. Auch Cronjobs, die das fragliche Filesystem benutzen tragen dazu bei, dass sich mehr und mehr hängende Prozesse auf dem Nagios-Server ansammeln. Eine Methode, mehrfach laufende Plugins zu verhindern, wäre die Benutzung eines pid-Files. Bei jedem Aufruf würde ermittelt, ob es bereits ein laufendes Plugin gibt. Ist das der Fall, dann wird davon ausgegangen, dass das wegen eines nicht reagierenden Filesystems so ist und es wird ein *CRITICAL*-Status zurückgeliefert. Es gibt aber noch eine elegantere Methode, die überhaupt nicht zu hängenden Prozessen führt. Threads werden zwar auch bei der Ausführung von IO-Operationen blockiert, können jedoch in dieser Situation im Gegensatz zu Prozessen beendet werden. Dies macht man sich im Plugin zunutze, indem man die kritischen Dateizugriffe in eigenen Threads laufen lässt. Sie kehren unter normalen Umständen dann freiwillig zum Hauptthread zurück, oder werden durch einen Timeout zwangsweise beendet.

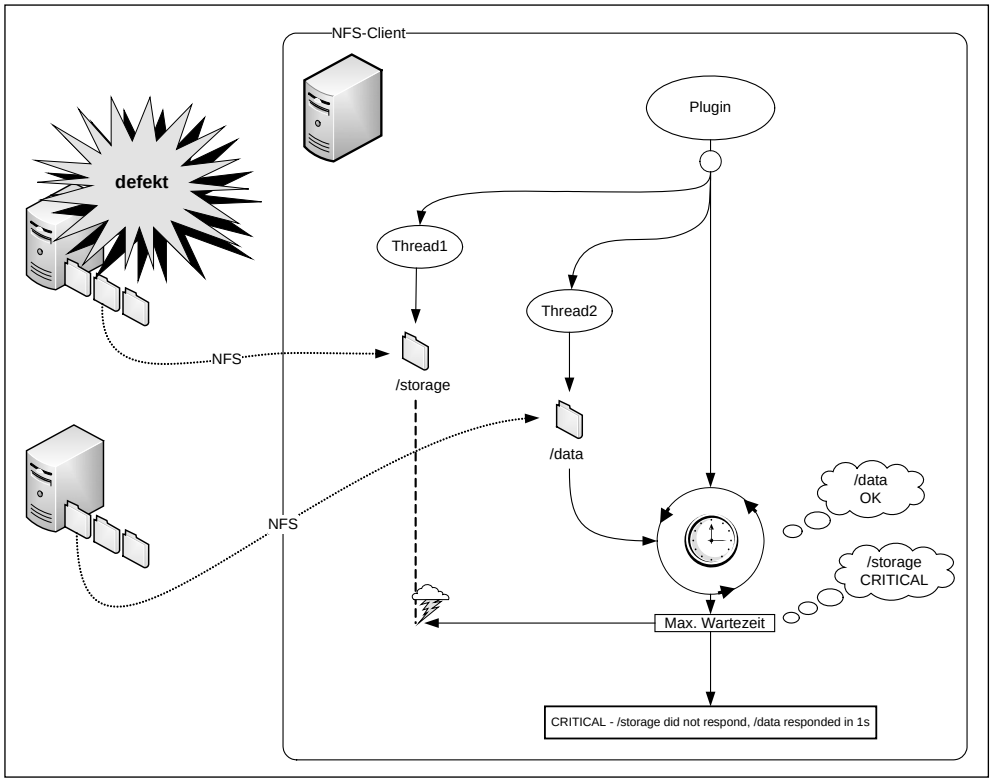


Abbildung 3.2: **Threads prüfen die Verfügbarkeit von Filesystemen**

3.2.2 check_fs_ping

Da sich das Plugin auf beliebige Arten von Filesystemen anwenden lassen soll und auch prüfen soll, ob ein Zugriff innerhalb einer bestimmten Zeitspanne erfolgreich ausgeführt

wird, liegt der Vergleich mit **check_ping** nahe, dass Knoten im Netzwerk auf ihre Erreichbarkeit prüft und die Antwortzeiten misst. Der Name für das hier beschriebene Plugin soll daher **check_fs_ping** lauten.

Zwar beinhaltet Perl bereits im Standardumfang das Modul *thread.pm*, jedoch fehlt diesem zumindest bei der Version 5.8.8 die Funktion *is_joinable()*. Diese ist jedoch für das Plugin unverzichtbar. Ob sie verfügbar ist, prüft man mit diesem Kommando:

```
perl -Mthreads -e '$t = threads->create(sub {}); $t->is_joinable();'
```

Erscheint daraufhin eine Fehlermeldung, die mit dem folgendem Text beginnt: *Can't locate auto/threads/is_joinable.al*, dann muss man die aktuelle Version des Moduls installieren, die auf CPAN zum Download bereitliegt. Am bequemsten geht das mit der CPAN-Shell.

```
nagsrv# perl -MCPAN -eshell
cpan> o conf http_proxy http://proxy.naprax.de:3128/
cpan> install thread
```

Das Plugin soll auch das Perl-Modul *Nagios::Plugin* verwenden. Es ist daher nötig, dass der Perl-Interpreter dessen Installationspfad kennt. Wenn es bei der Installation der Nagios-Plugins (siehe Kapitel Installation) im Unterverzeichnis *perl/lib* des Nagios-Homeverzeichnisses abgelegt wurde, dann muss man folgende Environmentvariable setzen:

```
export PERL5LIB=$PERL5LIB:$HOME/perl/lib
```

Alternativ kann auch direkt im Plugin dieser Pfad angegeben werden. Darauf wird an entsprechender Stelle hingewiesen.

Das fertige Plugin kann aus dem Netz heruntergeladen[9] werden. Schritt für Schritt soll nun der Code erläutert werden.

```
#! /usr/bin/perl
```

In der ersten Zeile findet sich nach den Zeichen *#!* der Pfad des Perl-Interpreters. Das ist ein Hinweis an den Unix-Kernel, wie die vorliegende Datei ausgeführt werden muss. Normalerweise müsste ein Perl-Script mit **perl <scriptname>** und z.B. ein Shell-Script mit **sh <scriptname>** aufgerufen werden. Da es aber üblich ist, den Interpreter wegzulassen und direkt **<scriptname>** aufzurufen, braucht man diese besondere Form eines Kommentars in der ersten Zeile. Aus *<scriptname>* macht dann im vorliegenden Fall das Betriebssystem selbständig **/usr/bin/perl –w <scriptname>**.

9 http://www.consol.de/opensource/nagios/check-fs-ping

```
use strict;
# use lib '/usr/local/nagios/perl/lib';
use Nagios::Plugin;
use threads;
```

Mit der use-Anweisung werden beim Programmstart externe Module geladen, die den Funktionsumfang von Perl erweitern. Der erste Eintrag dient dazu, den Code im Script besonders akribisch auf Fehler zu prüfen. Man sollte sich angewöhnen, immer *use strict* zu verwenden. Das zweite Modul *Nagios::Plugin* stellt die bereits besprochenen Plugin-Funktionen zur Verfügung. Zuletzt wird noch das *threads*-Modul benötigt. Es ermöglicht die Erzeugung und Verwaltung von Threads in einem Perl-Programm. Die aus kommentierte Zeile ist die angedeutete Alternative zur Environmentvariablen *$PERL5LIB*.

```
my $plugin = Nagios::Plugin->new(
    usage => 'Usage: %s '.
        [ -v|--verbose ] [-t <timeout>] '.
        '--warning <seconds> --critical <seconds> '.
        '--path <path to check> [--path <path to check> ...]'
);
```

Mit dieser Anweisung wird ein Perl-Objekt namens *$plugin* ins Leben gerufen. Der Konstruktor *new* bekommt hier mit *usage* einen String mitgeteilt, der als Hinweistext ausgegeben wird, falls das Plugin mit ungültigen Kommandozeilenparametern aufgerufen wird.

```
$plugin->add_arg(
    spec => 'warning|w=f',
    help => ['-w, --warning=INTEGER|FLOAT.',
             'Minimum "hang" time until warning. (default is 1s)'],
    required => 0,
);
```

Danach folgen die Definitionen für die zu erwartenden Kommandozeilenparameter durch die Methode *add_arg*. Im ersten Abschnitt wird mit *spec* angegeben, dass sowohl *--warning* als auch *-w* erlaubt sind. Der Parameter kann Ganz- oder auch Kommazahlen als Wert entgegennehmen. Das wird durch das *=f* ausgedrückt. Mit *help* teilt man dem *$plugin*-Objekt mit, dass der folgende Text zur Erläuterung der Option ausgegeben werden soll, wenn das Plugin mit *--help* aufgerufen wird. Schließlich folgt mit *require* noch der Hinweis, dass dieser Parameter nicht zwingend erforderlich ist. (Falls er fehlt, wird ein Defaultwert angenommen).

```
$plugin->add_arg(
    spec => 'critical|c=f',
    help => ['-c, --critical=INTEGER|FLOAT',
             'Minimum "hang" time until critical. (default is 5s)'],
    required => 0,
);
```

Dieser Aufruf ist mit dem vorherigen identisch. Auf dieselbe Art und Weise wird der Parameter *--critical* bekannt gemacht.

```
$plugin->add_arg(
    spec => 'path|p=s@',
    help => '--path=STRING . The path leading to the filesystem/file.',
    required => 1,
);
```

Das Besondere am Parameter *--path* ist, dass er mehrfach angegeben werden kann. Die einzelnen Werte, also die unterschiedlichen Verzeichnisnamen, werden dem Plugin dann als Array zur Verfügung stehen. Dies wird durch *=s@* ausgedrückt. Das *s* bedeutet dabei, dass dieses Mal statt einer Zahl ein beliebiger String erwartet wird. Natürlich handelt es sich hier um einen Pflichtparameter, daher bekommt *required* den Wert 1. Das Plugin muss ja mindestens ein Verzeichnis überprüfen.

```
$plugin->getopts();
```

Die Methode *getopts* parst nach dem Aufruf des Plugins die Kommandozeilenparameter und überprüft sie mit den soeben gemachten Definitionen auf ihre Korrektheit. Taucht ein unbekannter Parameter auf oder wird z.B. ein Wort vorgefunden, wo eine Zahl erwartet wurde, führt dies zur Ausgabe eines Hilfetextes und zum Abbruch des Scripts. Der Hilfetext setzt sich aus den Angaben zusammen, die bei den *add_arg*-Aufrufen mit *help* vorgegeben wurden.

```
$plugin->set_thresholds(
    warning => ($plugin->opts->warning() || 1),
    critical => ($plugin->opts->critical() || 5),
);
```

Wenn die Analyse der Plugin-Parameter erfolgreich war, werden die Schwellwerte gesetzt, mit denen später die ermittelten Messwerte verglichen werden. Hier muss darauf geachtet werden, dass *--warning* und *--critical* freiwillige Angaben sind. Werden sie auf der Kommandozeile weggelassen, dann muss dafür ein Defaultwert vergeben werden. Die Schwellwerte stehen nach dem Aufruf von *getopts* als Rückgabewerte von Funktionen zur Verfügung. Mit *$plugin->opts->warning()* bekommt man beispielsweise die Zahl, die man nach dem Parameter *--warning* eingegeben hat. Hat man hier keine Angaben gemacht, so ist der Rückgabewert der Funktion *undef*. Obige OR-Ausdrücke in Klammern liefern daher vorzugsweise den Wert von der Kommandozeile und falls es diesen nicht gibt, einen vorgegebenen Defaultwert.

```
my $threads = {};
foreach (map { split ',' } @{$plugin->opts->path}) {
  $threads->{$_}->{thread} = threads->create(
      sub {
        if (-e $_) {
```

```
            return 1;
          } else {
            return 0;
          }
        }
    );
}
```

In diesem Abschnitt wurde für jeden der Pfade, die mit *--path* angegeben wurden, ein eigener Thread erzeugt. Dazu dient der Konstruktor *threads->create()*. Er bekommt als Parameter eine Referenz auf eine Perl-Subroutine. Diese wird dann in einem eigenen Thread parallel zum Hauptprogramm ausgeführt. Die Subroutine macht nichts weiter als festzustellen, ob der angegebene Pfad existiert. Der Rückgabewert ist dann entsprechend 1 oder 0. Allerdings wird das *return*-Statement nur ausgeführt, wenn das unter dem Pfad liegende Filesystem verfügbar ist. Bei Problemen bleibt nämlich der Thread in der Abfrage *-e* hängen. Das kann im Falle eines defekten NFS-Servers so lange dauern, bis dieser repariert wurde. Ein Thread muss also dahingehend geprüft werden, ob er am Ende seiner Laufzeit mit *return* eine Aussage über die Existenz des abgefragten Pfades liefert und oder ob er sich überhaupt nicht beendet hat.

```
my $sleep = sub { sleep(shift) };
my $granularity = 1;
eval {
  require Time::HiRes;
  import Time::HiRes "sleep";
  $sleep = sub { Time::HiRes::sleep(shift) };
  $granularity = 0.1;
};
```

Diese Zeilen haben mit dem eigentlichen Ziel des Plugins nichts zu tun. Sie dienen nur dazu, die Aussagekraft des Plugin-Outputs zu verbessern. Die *sleep()*-Funktion von Perl dient dazu, den Programmfluss für eine gewisse Zeit zu stoppen und eine Pause einzulegen. Dazu ruft man das *sleep()*-Kommando auf und übergibt diesem die gewünschte Anzahl von Sekunden. Defaultmäßig sind hier nur ganze Zahlen möglich. Lädt man allerdings das Modul *Time::HiRes*, dann kann sleep auch mit Gleitkommazahlen umgehen. Damit kann man ein Perl-Script auch z.B. für eine Zehntelsekunde anhalten. Der obige Code definiert mit der Variablen *$sleep* eine Referenz auf eine Funktion, die entweder dem Default-*sleep()* oder, falls das Modul *Time::HiRes* verfügbar ist, dem hochauflösenden *sleep()* entspricht. Daneben wird noch eine Variable *$granularity* definiert, die angibt, in welchen Zeitintervallen geprüft werden soll, ob ein Thread beendet wurde.

Der nun folgende Abschnitt wird aus Gründen der Übersichtlichkeit verkürzt dargestellt. Die ausführliche Version findet man nach diesen Erläuterungen.

```
my $elapsed = 0;
my $timeout = $plugin->opts->timeout || 15;
```

In der Variablen *$elapsed* wird die verstrichene Zeit gespeichert. Sie wird in der folgenden Schleife ständig wachsen. Die Variable *$timeout* gibt die maximale Zeit an, während der auf zurückkehrende Threads gewartet wird. Spätestens wenn diese um ist, beendet sich das Plugin. Entweder man gibt beim Aufruf des Plugins mit dem Parameter *--timeout* die gewünschte Anzahl von Sekunden an, oder es wird ein Defaultwert von 15 Sekunden verwendet.

```
while ($elapsed < $timeout) {
   last if ! scalar(keys %{$threads});
   foreach (keys %{$threads}) {
```

In der nun beginnenden Schleife sollen alle laufenden Threads (von denen einer für jedes mit *--path* angegebene Verzeichnis gestartet wurde) überprüft werden, ob sie ihre Aufgabe erledigt haben. Wenn das bei einem Thread der Fall ist, wird der entsprechende Eintrag aus dem *$threads*-Hash entfernt. Deshalb wird die Schleife mit dem *last*-Kommando verlassen, wenn es keine Threads mehr zu überprüfen gibt. Innerhalb der *foreach*-Schleife enthält die Schleifenvariable *$_* den Verzeichnisnamen.

```
      # Feststellen, ob der Thread beendet wurde.
      if ($threads->{$_}->{thread}->is_joinable()) {
```

Mit der Methode *is_joinable()* stellt man fest, ob ein Thread fertig ist und vom Hauptprogramm „eingesammelt" werden kann. Dies macht man mit der Methode *join()*, die einen Thread endgültig beendet. Sie gibt dabei den Wert weiter, der im Thread mit *return* geliefert wurde.

```
         if ($threads->{$_}->{thread}->join()) {
```

Wenn die Subroutine mit return 1 bestätigt hat, dass das gewünschte Verzeichnis existiert und der Zugriff auf das zugrundeliegende Filesystem nicht wegen eines NFS-Problems blockiert wurde, kann in diesem Abschnitt ein Ergebnis für die spätere Ausgabe formuliert werden. Dazu muss man erst noch den Wert von *$elapsed* überprüfen. Er gibt an, wie lange der Zugriff gedauert hat. Man vergleicht ihn mit den Warning- und Critical-Schwellwerten und kann dann *add_nagios()* mit den entsprechenden Parametern aufrufen.

```
         } else {
```

Im Alternativzweig wird der Fall behandelt, bei dem zwar das zugrundeliegende Filesystem innerhalb des Timeouts auf den Zugriff reagiert hat, aber das gewünschte Verzeichnis oder die gewünschte Datei nicht gefunden wurde. In diesem Fall wird hier ein critical Status für diesen Pfad vermerkt. Man fragt sich vielleicht:»Wozu?«. Das Filesystem war ja immerhin ansprechbar. Es könnte ja sein, dass der NFS-Mount versehentlich weggefallen ist und der

Zugriff im lokalen Mountpoint stattgefunden hat. Deshalb gibt man mit *--path* auch idealerweise ein Verzeichnis oder eine Datei an, die nur auf dem gemouteten und nicht im lokalen Filesystem existiert.

```
        }
        delete $threads->{$_};
```

Nachdem ein Thread behandelt wurde, der sich freiwillig beendet und ein Resultat geliefert hat, kann er aus dem Hash *$threads* entfernt werden.

```
    } elsif ($threads->{$_}->{thread}->is_running()) {
```

In diesem Zweig werden Threads untersucht, die auch nach *$elapsed* Sekunden noch nicht fertig sind. Ein einfaches *else* hätte hier auch genügt, aber zur Verdeutlichung wird die Methode *is_running()* aufgerufen. Zugriffe auf funktionierende Filesysteme werden üblicherweise bereits beim ersten Schleifendurchlauf im vorhergehenden Zweig abgearbeitet. An diese Stelle im Code gelangt man dann nur noch, wenn tatsächlich ein hängendes Filesystem vorliegt.

```
        if ($plugin->check_threshold($elapsed) == 2) {
```

Hat die Wartezeit, ausgedrückt durch *$elapsed* bereits die Critical-Schwelle überschritten, kann man auch mit der Untersuchung des aktuellen Verzeichnisses aufhören. Weitere Wartezyklen würden nur die Laufzeit des Plugins erhöhen und sind unnötig, da der Status CRITICAL bereits feststeht. Man wird also an dieser Stelle mit *add_nagios()* eine entsprechende Fehlermeldung erzeugen.

```
            $threads->{$_}->{thread}->detach();
            delete $threads->{$_};
```

Danach kann der Thread zwangsweise beendet und der entsprechende Eintrag im *$threads*-Hash gelöscht werden. An dieser Stelle ist übrigens deutlich zu sehen, wie sich Prozesse und Threads von einander unterscheiden. Ein durch eine hängende IO-Operation blockierter Prozess ließe sich durch keinerlei Maßnahmen beenden (zumindest nicht bei einem NFS-Mount mit der Option *hard*). Threads dagegen kann man mit *detach()* aufräumen.

```
        }
    }
  }
  $elapsed += &$sleep($granularity);
}
```

Nach jedem Zyklus, bei dem die Threads untersucht wurden, wird eine kleine Pause eingelegt. Die dafür vorgesehene Zeit richtet sich danach, ob das *Time::HiRes*-Modul zur Ver-

fügung steht oder nicht. Wenn ja, wird eine Zehntelsekunde gewartet, andernfalls eine ganze Sekunde. Die Verwendung des hochauflösenden Timers hat einzig einen kosmetischen Grund für die Aufzeichnung der Performancedaten (beispielsweise durch PNP). Ohne *Time::HiRes* würde man für die y-Achse ausschließlich ganzzahlige Werte erhalten, i. d. R. 1. Dadurch bekäme man einfach eine durchgehende horizontale Linie. Mit *Time::HiRes* hingegen würden die kleinen Schwankungen im Zehntelsekundenbereich für einen eher gezackten Kurvenverlauf sorgen. Eigentlich müsste man zu *$elapsed* noch die Zeit dazuzählen, die bei der Behandlung der Threads aufgewendet werden musste. Da sie sich aber im Bereich weniger Microsekunden bewegt, wird darauf verzichtet.

```
my ($code, $message) = $plugin->check_messages(join_all => ', ');
$plugin->nagios_exit($code, $message);
```

Abschließend werden aus den Zwischenergebnissen, die mit *add_nagios()* im $plugin-Objekt gespeichert wurden, der Exitcode und der Ausgabetext erstellt. Durch die *nagios_exit()*-Methode werden noch die Performancedaten an den Text angehängt, dann beendet sich das Script mit dem vorgegebenen Exitcode.

Das komplette Plugin mit den *add_nagios()* und *add_perfdata()*-Aufrufen sieht dann folgendermaßen aus:

```
#! /usr/bin/perl

use strict;
use Nagios::Plugin;
use threads;
#use threads::shared;

*Nagios::Plugin::Functions::get_shortname = sub {
    return undef; # suppress output of shortname
};
my $plugin = Nagios::Plugin->new(
    shortname => '',
    usage => 'Usage: %s [ -v|--verbose ] [ -t <timeout> ] '.
        '--warning <seconds> --critical <seconds> '.
        '--path <path to check> [--path <path to check> ...]'
);
$plugin->add_arg(
    spec => 'path|p=s@',
    help => '--path=STRING . The path leading to the filesystem in ques-
tion.',
    required => 1,
);
$plugin->add_arg(
    spec => 'warning|w=s',
    help => ['-w, --warning=INTEGER.',
             'Minimum "hang" time until warning. (default is 1s)'],
    required => 0,
);
$plugin->add_arg(
    spec => 'critical|c=s',
```

```
    help => ['-c, --critical=INTEGER',
            'Minimum "hang" time until critical. (default is 5s)'],
    required => 0,
);

$plugin->getopts();
$plugin->set_thresholds(
    warning => ($plugin->opts->warning() || 1),
    critical => ($plugin->opts->critical() || 5),
);

my $threads = {};
foreach (map { split ',' } @{$plugin->opts->path()}) {
   $threads->{$_}->{thread} = threads->create(
       sub {
          if (-e $_) {
             return 1;
          } else {
             return 0;
          }
       }
   );
}

my $sleep = sub { sleep shift };
my $granularity = 1;
eval {
   require Time::HiRes;
   import Time::HiRes "sleep";
   $sleep = sub { Time::HiRes::sleep(shift) };
   $granularity = 0.1;
};
my $elapsed = 0;
my $timeout = $plugin->opts->timeout || 15;
while ($elapsed < $timeout) {
   last if ! scalar(keys %{$threads});
   foreach (keys %{$threads}) {
      if ($threads->{$_}->{thread}->is_joinable()) {
         if ($threads->{$_}->{thread}->join()) {
            my $level = $plugin->check_threshold($elapsed);
            $plugin->add_message($level,
                sprintf "%s responded within %.2fs", $_, $elapsed);
         } else {
            $plugin->add_message(CRITICAL,
                sprintf "%s does not exist", $_);
         }
         $plugin->add_perfdata(
             label => $_,
             value => $elapsed,
             uom => 's',
             threshold => $plugin->threshold(),
         );
         delete $threads->{$_};
      } elsif ($threads->{$_}->{thread}->is_running()) {
         if ($plugin->check_threshold($elapsed) == 2) {
            $threads->{$_}->{thread}->detach();
            $plugin->add_message(CRITICAL,
```

```
                sprintf "%s did not respond within %.2fs",
                    $_, $elapsed);
            $plugin->add_perfdata(
                label => $_,
                value => $elapsed,
                uom => 's',
                threshold => $plugin->threshold(),
            );
            delete $threads->{$_};
        }
    }
  }
  $elapsed += &$sleep($granularity);
}
my ($code, $message) = $plugin->check_messages(join_all => ', ');
$plugin->nagios_exit($code, $message);
```

Bevor man das Plugin in die produktive Nagios-Installation durch Kopieren in das *remotelibexec*-Verzeichnis einbindet, führt man üblicherweise ein paar Tests auf der Kommandozeile aus. Dabei sind zwei Dinge zu beachten:

» Nagios ruft die Plugins mit ihrem vollen Pfadnamen auf. Üblicherweise wird in der Command-Definition beim *check_command*-Attribut ein *$USERx$*-Makros verwendet, hinter dem sich das Unterverzeichnis *remotelibexec* verbirgt. Steht dort z.B.
check_command $USER3$/check_plugin
dann führt Nagios **/usr/local/nagios/libexec/check_plugin** aus. Genauso sollte man Plugins auch beim Testen aufrufen. Der Grund dafür ist, dass manche Plugins den Pfad auswerten, mit dem sie gestartet wurden, um Perl-Module nachzuladen. Ein Beispiel dafür ist *utils.pm*, das gemeinsam mit den Plugins im Unterverzeichnis *libexec* liegt.

» Plugins testet man als der User, unter dessen ID auch der Nagios-Daemon läuft und **niemals als root**. Man spart sich dadurch unangenehme Überraschungen, wenn das Plugin besondere Privilegien oder Pfadeinstellungen benutzt, die in einer root-Shell bestehen, aber später unter der Nagios-Kennung nicht mehr.

Mit dem **id**-Kommando prüft man also, ob man wirklich als User *nagios* angemeldet ist. Das mag übertrieben erscheinen, aber kostet nicht viel Mühe. Man sollte sich wirklich angewöhnen, bewusst als *nagios* zu arbeiten.

```
nagsrv$ id
uid=500(nagios) gid=500(nagios) groups=500(nagios),501(nagcmd)
```

Danach ruft man das Plugin mit dem vollen Pfadnamen auf. So sieht das Resultat aus, wenn das NFS-gemountete Filesystem */storage* ordnungsgemäß funktioniert:

```
nagsrv$ /usr/local/nagios/libexec/check_fs_ping --path /storage
OK - /storage responded within 0.10s | /storage=0.10131s;1;5
```

Wichtiger ist es allerdings, dass ein Plugin Fehler erkennt, denn das ist ja seine eigentliche Aufgabe. Deshalb sollte man alle erdenklichen Szenarien simulieren. Wenn das aus betrieblichen Gründen nicht möglich ist, sollte man zumindest im Code an den entsprechenden Stellen Manipulationen vornehmen, die dem Plugin eine Fehlersituation vortäuschen. Bei **check_fs_ping** könnte man beispielsweise durch Einfügen einer *sleep*-Anweisung die Ausführung der Threads künstlich anhalten.

```
$threads->{$_}->{thread} = threads->create(
    sub {
      sleep(300);
      if (-e $_) {
        return 1;
      } else {
        return 0;
      }
    }
);
```

Damit simuliert man einen blockierten IO-Systemaufruf. Natürlich ist es noch „authentischer", wenn man die Möglichkeit hat, einen NAS-Filer zu rebooten oder zumindest ein Netzwerkkabel zu ziehen. Dadurch würde jeder Zugriff auf das */storage*-Verzeichnis geblockt und die Ausgabe von **check_fs_ping** würde so aussehen:

```
nagsrv$ /usr/local/nagios/libexec/check_fs_ping --path /storage
CRITICAL - /storage did not respond within 5.05s | /storage=5.045139s;3;4
```

Wenn der NFS-Server wieder verfügbar ist, sollte normalerweise der Zugriff auf ein von ihm gemountetes Filesystem so funktionieren, als wäre nie etwas geschehen. Leider gibt es Fälle, in denen ein Mount nach dem Wiederanlauf beschädigt ist.

```
nagsrv# df
Filesystem           1K-blocks      Used Available Use% Mounted on
…
nas.naprax.de:/mnt/md1/db2
                             -         -         -   -  /storage

nagsrv# ls /storage
ls: /storage: Permission denied
```

Auch dann liefert **check_fs_ping** einen Critical-Status und man kann den Fehler beheben, indem man das Filesystem unmountet und anschließend wieder neu mountet.

```
nagsrv$ /usr/local/nagios/libexec/check_fs_ping --path /storage
CRITICAL - /storage does not exist | /storage=0.100482s;1;5
```

Wenn ein Host mehrere Filesysteme von verschiedenen Filern mountet, empfiehlt es sich für die Nagios-Konfiguration, die Mountpoints in jeweils einem Service pro Filer zusammenzufassen. Angenommen, der **df**-Befehl zeigt auf dem Server *wwwsrv18* folgende Mounts an:

```
wwwsrv18# df -t nfs
    1K-blocks      Used Available Use% Mounted on
netapp176.naprax.de:/vol/vgm448/qgm09766/sw-linux
     26214400  18255632   7958768  70% /mnt/sw/sw_linux
netapp161.naprax.de:/vol/vgm374/qgm08657/roll/shared1
      2406400    366584   2039816  16% /mnt/www/roll-p-shared1
netapp176.naprax.de:/vol/vgm448/qgm09834/intern/shared1
     10485760    695248   9790512   7% /mnt/www/intern-p-shared1
netapp176.naprax.de:/vol/vgm448/qgm09834/intern/web-intern1
     10485760    695248   9790512   7% /mnt/www/intern-p-web-intern1
netapp166.naprax.de:/vol/vgm373/qgm08396/erp/shared1
       204800     49304    155496  25% /mnt/www/erp-p-shared1
netapp166.naprax.de:/vol/vgm373/qgm08400/bench2/shared1
       921600      5672    915928   1% /mnt/www/bench2-p-shared1
netapp166.naprax.de:/vol/vgm373/qgm08426/ada/shared1
      3276800    122184   3154616   4% /mnt/www/ada-p-shared1
netapp171.naprax.de:/vol/vgm449/qgm10201/bded/web1
      1048576    138968    909608  14% /mnt/www/bded-p-web1
netapp161.naprax.de:/vol/vgm374/qgm08736/quick/shared1
      3276800   2608872    667928  80% /mnt/www/quick-p-shared1
netapp161.naprax.de:/vol/vgm374/qgm08737/print/shared2
      3145728     62560   3083168   2% /mnt/www/print-p-shared2
netapp171.naprax.de:/vol/vgm449/qgm10013/content/shared
      1048576     67384    981192   7% /mnt/www/content_shared
netapp161.naprax.de:/vol/vgm374/qgm08156/admin/shared1
      2097152     39848   2057304   2% /mnt/www/admin-p-shared1
netapp176.naprax.de:/vol/vgm448/qgm09709/depot/shared1
      1572864    705392    867472  45% /mnt/www/depot-p-shared1
netapp014.naprax.de:/vol/vgm018/qgm00745/bded2/web1
       307200    192024    115176  63% /mnt/bded2-p-web1
netapp014.naprax.de:/vol/vgm018/qgm00761/data12/web
       276480     23704    252776   9% /mnt/data12-p
netapp031.naprax.de:/vol/vgm017/qgm00858/account/web1
       204800       184    204616   1% /mnt/account-p
netapp031.naprax.de:/vol/vgm017/qgm00894/PREPRO/web1
      2560000   2301392    258608  90% /mnt/prepro-p
netapp031.naprax.de:/vol/vgm061/qgm00940/webapp-ssl
       102400       256    102144   1% /mnt/webapp-ssl
netapp031.naprax.de:/vol/vgm017/qgm00942/rollout/web1
       204800    123336     81464  61% /mnt/rollout-p
```

Wie man sieht, werden Filesysteme von den Storage-Systemen *netapp176*, *netapp161*, *netapp166*, *netapp014* und *netapp031* gemountet. Es gibt drei mögliche Varianten, wie man dann die Services des Hosts *wwwsrv18* in Nagios aufteilen kann:

» Ein einziger Service, wobei dem Plugin check_fs_ping sämtliche Mountpoints auf einmal übergeben werden. Dadurch würde die Ausgabe sehr lang (und womöglich wegen der Begrenzung auf 8192 Zeichen durch Nagios abgeschnitten) und unübersichtlich. Eine Alarmierungs-SMS wäre nur schwer lesbar.

» Ein Service pro Mountpoint. Diese Lösung wäre am übersichtlichsten, hätte aber zur Folge, dass beim Ausfall eines Filers gleich mehrere Alarme versandt würden.

» Ein Service pro Filer. In dieser Variante übergibt man dem Plugin **check_fs_ping** jeweils die zu einem bestimmten Filer gehörenden Mountpoints. Das hat gegenüber Methode 1 den Vorteil, dass nicht so viel Text in einer Notification auftaucht und gegenüber Methode 2, dass weniger Notifications verschickt werden. Man kann auch noch weiter gehen und das Ganze mit **check_multi** realisieren.
Das hätte den Vorteil, dass auch in der Ansicht *Service Detail* auf der Nagios-Webseite die Mounts übersichtlich untereinander anzeigen werden.

Listing 3.4: **check_fs_ping_multi_netapp031.cfg**

```
command [accountp] = \
    $USER2$/check_fs_ping --path /mnt/account-p
command [preprop] = \
    $USER2$/check_fs_ping --path /mnt/prepro-p
command [webappssl] = \
    $USER2$/check_fs_ping --path /mnt/webapp-ssl
command [rolloutp] = \
    $USER2$/check_fs_ping --path /mnt/rollout-p
```

Service State Information

Current Status:	OK (for 0d 0h 10m 48s)
Status Information:	OK - 4 plugins checked, 0 critical, 0 warning, 0 unknown, 4 ok [1] accountp OK - /mnt/account-p responded within 0.10s [2] preprop OK - /mnt/prepro-p responded within 0.10s [3] webappssl OK - /mnt/webapp-ssl responded within 0.10s [4] rolloutp OK - /mnt/rollout-p responded within 0.10s
Performance Data:	/mnt/account-p=0.100075s;3;4 /mnt/prepro-p=0.100852s;3;4 /mnt/webapp-ssl=0.100702s;3;4 /mnt/rollout-p=0.101135s;3;4

Abbildung 3.3: **Die Shares des Filers netapp031 auf einen Blick**

```
From: Armin Admin
To: Bernd Berserker
Subject: Re: NFS-Probleme

Hallo Bernd,

das Plugin ist fertig und leistet nützliche Dienste. Da es bisher kein ver-
gleichbares Plugin gab, würde ich es gern veröffentlichen. Ich möchte ande-
ren ersparen, das Rad nochmal erfinden zu müssen. Wie stehst du dazu? Da ich
die üblichen Einwände und Bedenken erwarte, ein paar Gedanken von mir dazu:
-  wir verschenken zwar Know How, aber es bringt uns keinerlei
   Vorteil, dieses geheimzuhalten. Und bitte sag jetzt nicht,
   dass wir mehr Geschäft machen, wenn unsere Konkurrenz sich
   nach wie vor mit NFS-Ausfällen rumärgern muss.
-  sollten Bugs in unserem Plugin sein, dann werden die hoffentlich
   von anderen entdeckt und wir bekommen gratis eine
   Qualitätssicherung
```

```
- möglicherweise haben die Anwender Ideen, wie man das Plugin noch
  einsetzen könnte, auf die wir selber gar nicht gekommen wären.
- da ich das übliche Geheule von „Haftung“ und „rechtlich“ erwarte,
  wobei mir nie jemand erklären konnte, was er sich denn genau
  drunter vorstellt … uns kann keiner was. Wir schenken ein Stück
  Software her und zwingen niemand, es einzusetzen. Obendrein als
  Quellcode. Wenn‘s nicht funktioniert und sich jemand beschwert…
  Pech gehabt, er hätte sich den Code anschauen können.
- Wir haben eine Menge Geld gespart durch Nagios. Dass wir unsere
  Erfahrung an die Szene zurückgeben, ist doch wohl das Mindeste.

Gruss,
Armin

p.s. wenn seitens NAPRAX kein Interesse an einer Veröffentlichung besteht
oder die Rechtsabteilung dich in endlose Diskussionen verwickelt, darf ich
dann so tun, als hätte ich das Plugin in meiner Freizeit entwickelt und es
als Privatperson veröffentlichen?
```

3.3 Veröffentlichen des Plugins

OpenSource-Projekte wie Nagios leben von der Mitwirkung vieler Freiwilliger. Eine der Stärken von Nagios, die immense Zahl von Plugins für jedes erdenkliche Einsatzszenario, ist dieser Bereitschaft zu Teilen zu verdanken. Versiegt der Strom an Ideen und Lösungen aus der Community, dann kann ein OpenSource-Produkt schneller uninteressant werden, als man denkt. Es wird dann einfach von Alternativen verdrängt, bei denen die Anwender einen gewissen Schwung und somit auch mehr Zukunftssicherheit sehen. Insbesondere wenn man in der eigenen Firma viel investiert hat, um Nagios als Monitoring-Lösung zu etablieren, wird man daran interessiert sein, dass die weitere Entwicklung nicht zum Stillstand kommt. Deshalb sollte jeder dazu beitragen, den Wert von Nagios durch Hinzufügen neuer und Verbesserung vorhandener Funktionalität zu steigern. Die zentrale Webseite, wo die meisten der selbstgeschriebenen Plugins abgelegt werden, ist *Nagios Exchange*[10]. Das von Nagios Enterprises betriebene Repository bietet eine Kategorisierung nach Anwendungsbereichen, eine Art Wiki für die Dokumentation des Plugins und eine Bewertungsfunktion. Man muss sich nur registrieren und kann dann seine Kreationen hochladen. Danach sollte man im *Nagios-Portal*[11] eine Ankündigung posten. Dort tummeln sich viele Profis, die das Plugin auf Herz und Nieren testen und wertvolle Vorschläge liefern, was man noch verbessern könnte. Auch eine Ankündigung auf der Mailingliste *nagios-users* ist zu empfehlen, um einen internationalen Kreis von möglichen Interessenten zu erreichen.

10 http://exchange.nagios.org
11 http://www.nagios-portal.de

TIPP

Jeder, der Nagios im Einsatz hat, wird irgendwann vor der Aufgabe stehen, selbst ein Plugin für spezielle Belange schreiben zu müssen. Es sollte eine Selbstverständlichkeit sein, das Resultat der Allgemeinheit zur Verfügung zu stellen, sofern nicht firmenspezifische Richtlinien dieses verbieten. Um Erlaubnis bitten, kostet aber nichts und im Zweifelsfall wird die Firma einverstanden sein, wenn der Autor sein Plugin auf rein privater Basis veröffentlicht und jeden Bezug zur Firma weglässt. Programmieranfänger sind oft der Meinung, ihre Arbeit könne den kritischen Blicken von Profis nicht standhalten und genieren sich, etwas einem breiten Publikum zum Download anzubieten. Hierzu ist zu sagen, dass die Nagios-Community niemanden herablassend behandeln wird. Jeder hat mal klein angefangen. Wichtiger als ausgefeilter Programmierstil ist die äußere Form von Plugins und die Dokumentation. Wenigstens eine kurze README-Datei ist zu empfehlen, schon deshalb, weil man damit auch einen gewissen Respekt vor den künftigen Anwendern ausdrückt.

Je nach Umfang des Plugins kann es sinnvoll sein, nicht nur das Script zu verteilen, sondern es zusammen mit der *README*-Datei, Dokumentation und sonstigen Dreingaben in ein *tgz*-Archiv zu packen. Unter Umständen kann es sogar nötig sein, eine vollständige, auf *./configure;make* basierende Build-Umgebung zu liefern. Die Beschreibung so einer Distribution würde den Rahmen dieses Kapitels sprengen, da detaillierte Kenntnisse der GNU-*autotools* nötig sind. Als Vorlage können die Plugins der *check_*_health*-Familie dienen, die im Kapitel *Datenbanken* beschrieben sind.

Wenn man ein Plugin veröffentlicht, muss man natürlich auch damit rechnen, Fragen dazu beantworten zu müssen. Daraus können sich interessante Diskussionen entwickeln und Ideen, auf die man selber nicht gekommen wäre. Die unangenehme Seite soll natürlich auch nicht verschwiegen werden. Im ungünstigsten Fall bekommt man mehrmals am Tag Mails, die zwar zunächst einen gewissen Unterhaltungswert haben, einem aber irgendwann ziemlich auf die Nerven gehen können.

```
Hi Mr Armin
Good evening.

I write you for the nagios plugin check_mssql_healt.
I have read and search in your forum but, for me, i am a beginner, is avery
problem. Your plugin is very, very,very good. I cannot install. Do i need
NRPE? please give advice. URGENT!!!
```

3.3.1 Aufwertung des Plugins

Das Anbieten eines Plugins zum öffentlichen Download ist ein erster Schritt, um auch andere an der Lösung eines spezifischen Problems teilhaben zu lassen. Man kann aber noch mehr tun, um das Werk abzurunden.

Konfigurationsbeispiele

Hat der künftige Anwender ein Plugin heruntergeladen, kann er es erst richtig nutzen, wenn es in das Nagios-System eingebunden wurde. Er steht also zunächst vor der Aufgabe, entsprechende Service- und Command-Definitionen zu erstellen. Das ist in vielen Fällen trivial, kann aber je nach Funktionsumfang des Plugins auch zu einer kniffligen Aufgabe werden. Man sollte also zumindest den Anfängern zuliebe ein Beispiel mitliefern, das zeigt, wie eine Servicedefinition aussehen könnte. Ausgehend vom vorher beschriebenen Beispiel mit dem Server *wwwsrv18* und dessen Mounts vom Filer *netapp031* könnte man folgende Definition vornehmen:

```
define service {
   service_description       os_linux_fs_check_ping_netapp031
   host_name                 wwwsrv18
   use                       os_linux_default,pnp-preview-popup
   check_command             \
       check_nrpe_arg!30!\
       check_fs_ping!/mnt/account-p,/mnt/prepro-p,/mnt/webapp-ssl,/mnt/
rollout-p
}
```

Damit würde der Service *os_linux_fs_check_ping_netapp031* sämtliche vom Filer *netapp031* gemounteten Filesysteme überwachen. Eine Alternative stellt wie bereits erwähnt, die Verwendung von **check_multi** dar. Die einzelnen Filesysteme werden dann nicht in der Nagios-Konfiguration, sondern in einer eigenen **check_multi**-Konfigdatei hinterlegt. Das hat den Vorteil, dass Nagios bei einer Änderung nicht neu gestartet werden muss. Auch ist es dadurch möglich, dass Benutzer, die nicht zum Nagios-Team gehören, die Liste der Filesysteme selbständig auf dem aktuellen Stand halten können. Man muss dazu nur die Konfigurationsdatei *check_fs_ping_multi_netapp031.cfg* in einem geeigneten Verzeichnis hinterlegen, das für diese Fremdbenutzer zugänglich ist. Die Servicedefinition für diese Variante sieht dann so aus:

```
define service {
   service_description       os_linux_fs_check_ping_multi_netapp031
   host_name                 wwwsrv18
   use                       os_linux_default,pnp-preview-popup
   check_command             \
       check_nrpe_arg!30!\
       check_fs_ping_multi!check_fs_ping_multi_netapp031.cfg
}
```

Auf diese Art und Weise legt man dann für jeden benutzten Filer einen eigenen Service an. Aus Gründen der Übersichtlichkeit sollte man sie in einer eigenen Datei speichern. Ein möglicher Name, der ausdrücken soll dass hier die Filesystem-bezogenen Checks zu finden sind, wäre *os_linux_fs.cfg*. Da das Plugin check_fs_ping auf dem NFS-Client *wwwsrv18* ausgeführt werden soll, muss dort auch der NRPE-Daemon entsprechend vorbereitet sein. Die Definitionen in *nrpe.cfg* sehen für die beiden vorgeschlagenen Services so aus:

Listing 3.5: **nrpe.cfg auf dem NFS-Client wwwsrv18**

```
command[check_fs_ping]=/usr/local/nagios/locallibexec/check_fs_ping --path
$ARG1$

command[check_fs_ping_multi]=/usr/local/nagios/locallibexec/check_multi -f /
usr/local/nagios/etc/plugin-configs/$ARG1$ --set
USER2=/usr/local/nagios/locallibexec
```

PNP-Template

Nagios verändert sich mit der Zeit von einem reinen Incident Management System zu einer umfassenden Plattform, die auch Performance Management beinhaltet. Diese Schlagwörter sagen nichts anderes aus, als dass viele Firmen die anfallenden Performancedaten aufheben und auswerten. Als de-Facto-AddOn für die graphische Aufbereitung der Daten hat sich PNP durchgesetzt. Dank seiner sehr einfachen Installationsprozedur fügt es einem Nagios-System großen Mehrwert hinzu. Zwar kann PNP die Performancedaten beliebiger Plugins visualisieren, es benutzt dafür jedoch ein generisches Design. Mit wenig Aufwand lässt sich jedoch für ein Plugin ein sogenanntes PNP-Template erstellen, das durch Farbgebung und Beschriftung individuell gestaltet werden kann. Wie das geht, wird im Kapitel *PNP* genau beschrieben. Man kann deshalb sein selbstgeschriebenes Plugin sozusagen auch optisch aufwerten, indem man ein solches PNP-Template beilegt. Das ist nicht übermäßig kompliziert, zumal mit PNP viele Templates mitgeliefert werden, die man als Vorlage benutzen kann.

```
<?php
#
# Copyright (c) Armin Admin (armin.admin@naprax.de)
# Plugin: check_fs_ping
#
$green = "33FF00E0";
$yellow = "FFFF00E0";
$red = "F83838E0";
$defcnt = 1;
foreach ($DS as $i) {
    $warning = ($WARN[$i] != "") ? $WARN[$i] : "";
    $critical = ($CRIT[$i] != "") ? $CRIT[$i] : "";
    $ds_name[$defcnt] = "Response Time";
    $opt[$defcnt] = "--vertical-label \"Response Time [s]\" -l 0 -r --title
\"Response time for filesystem $NAME[$i]\" ";
    $def[$defcnt] .= "DEF:response=$rrdfile:$DS[$i]:AVERAGE:reduce=LAST " ;
    $def[$defcnt] .= "CDEF:ag=response,$WARN[$i],LE,response,0,GT,INF,UNKN,I
F,UNKN,IF,ISINF,response,0,IF ";
    $def[$defcnt] .= "CDEF:ay=response,$CRIT[$i],LE,response,$WARN[$i],GT,IN
F,UNKN,IF,UNKN,IF,ISINF,response,0,IF ";
    $def[$defcnt] .= "CDEF:ar=response,100,LE,response,$CRIT[$i],GT,INF,UNKN
,IF,UNKN,IF,ISINF,response,0,IF ";
    $def[$defcnt] .= "AREA:ag#$green: " ;
    $def[$defcnt] .= "AREA:ay#$yellow: " ;
    $def[$defcnt] .= "AREA:ar#$red: " ;
    $def[$defcnt] .= "LINE:response#111111:\" \" ";
```

```
    $def[$defcnt] .= "VDEF:lresponse=response,LAST " ;
    $def[$defcnt] .= "VDEF:mresponse=response,MAXIMUM " ;
    $def[$defcnt] .= "VDEF:aresponse=response,AVERAGE " ;
    $def[$defcnt] .= "GPRINT:lresponse:\"$NAME[$i] response time was %.2lfs
(LAST)\" " ;
    $def[$defcnt] .= "GPRINT:mresponse:\"%.2lfs (MAX)\" ";
    $def[$defcnt] .= "GPRINT:aresponse:\"%.2lfs (AVG)\" ";
    $defcnt++;
}
?>
```

So ein PNP-Template als Dreingabe rundet das Plugin ab. Anwender, die großen Wert auf das optische Erscheinungsbild ihrer Nagios-Installation legen, werden es Ihnen danken.

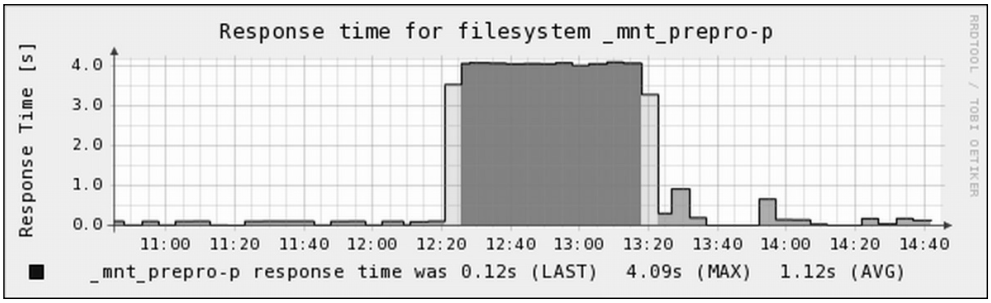


Abbildung 3.4: **Aufwertung des Plugins durch Erstellung eines Graphen**

3.4 Modifizieren existierender Plugins

Es ist nicht immer nötig, gleich ein neues Plugin zu schreiben. Wenn es bereits eines gibt, das eine ähnliche Thematik abdeckt, sollte man besser dazu beitragen, die neue Funktionalität in das vorhandene Plugin einfließen zu lassen. Damit hilft man den Nagios-Anwendern, den Überblick zu behalten. Es gibt bereits genug Mini-Plugins, die jeweils nur einen Einzelaspekt prüfen.

Die Problematik der hängenden NFS-Mounts tritt im Zusammenhang mit Nagios besonders bei der Verwendung von **check_disk** zu Tage. Mit diesem Plugin überwacht man üblicherweise das Vorhandensein und den freien Speicherplatz von Filesystemen. Fällt ein NFS-Server aus, gilt für **check_disk** das gleiche wie für jeden anderen Prozess auch: Er bleibt beim Aufruf einer Kernelroutine im IO-System stehen (im Fall von **check_disk** ist das der *stat*-Systemaufruf) und lässt sich nicht einmal mehr mit **kill -9** beenden. Nagios wird in diesem Fall einen Timeout feststellen und den Prozess, in dem das Plugin läuft, abschießen. Allerdings kümmert Nagios sich nicht darum, ob dieses den gewünschten Erfolg gezeigt hat. Nach Verstreichen des Retro-Intervalls startet es die Prozedur erneut, was wiederum mit Timeout und einem weiteren hängenden Prozess endet. Nach und nach füllt sich so die Prozessliste mit **check_disk**-Prozessen. Es liegt also nahe, dem Plugin **check_disk** den Trick mit den Threads beizubringen. Damit wäre es in der Lage, hängende NFS-Mounts zu erkennen und somit zu alarmieren und würde sich gleichzeitig auf saubere Art beenden. Die

Stelle, an der man ansetzt, ist die Funktion *stat_path()*. Jedes Mal bevor die Information über den freien Speicherplatz eines Filesystems eingeholt wird, ruft **check_disk** diese Funktion auf, um festzustellen, ob der entsprechende Pfad überhaupt existiert. Wenn das nicht der Fall ist, dann wird die Ausführung des Plugins sofort mit einer Fehlermeldung abgebrochen.

Listing 3.6: **check_disk.c(stat_path)**

```
void
stat_path (struct parameter_list *p)
{
  /* Stat entry to check that dir exists and is accessible */
  if (verbose >= 3)
    printf("calling stat on %s\n", p->name);
  if (stat (p->name, &stat_buf[0])) {
    if (verbose >= 3)
      printf("stat failed on %s\n", p->name);
    printf("DISK %s - ", _("CRITICAL"));
    die (STATE_CRITICAL, _("%s %s: %s\n"), p->name, _("is not accessible"),
stre
rror(errno));
  }
}
```

Der in dieser Funktion enthaltene *stat()*-Aufruf soll künftig in einem eigenen Thread laufen, dessen Ende vom Hauptprogramm abgefragt wird. Damit durch die Modifikationen möglichst wenig am Original-Code verändert wird, behält man diese Routine bei. Sie soll unverändert im Thread laufen, bekommt aber einen anderen Funktionsnamen. Den alten Namen bekommt eine neue Subroutine, die sich um Threaderzeugung und Fehlerbehandlung kümmert.

An diesem Beispiel soll im restlichen Kapitel demonstriert werden, wie man einen Patch für vorhandenen Code erstellt und wie man ihn beim Nagios-Plugin-Projekt einreicht. Gleichzeitig soll damit auch dazu aufgerufen werden, Plugins oder allgemein OpenSource-Software nicht als unantastbare Black Box zu sehen, sondern aktiv an der Verbesserung mitzuarbeiten.

Zur Erstellung eines Patches entpackt man am besten das Archiv *nagios-plugins-<release>.tar.gz*, so dass dessen Dateibaum im Originalzustand ohne Modifikationen durch ./configure oder make vorliegt.

```
nagsrv$ tar zxvf nagios-plugins-1.4.13.tar.gz
…
```

Danach erstellt man von dem entstandenen Verzeichnis eine Kopie. Diese dient dann als Arbeitsverzeichnis, in dem man die eigenen Modifikationen vornimmt.

```
nagsrv$ mkdir nagios-plugins-1.4.13.nfs
nagsrv$ cp -r nagios-plugins-1.4.13/ nagios-plugins-1.4.13.nfs
…
```

Aus Rücksicht auf Betriebssysteme, die die für die Verbesserung benötigten Thread-Routinen nicht kennen, schließt man den neuen Code in eine *#ifdef*-Anweisung für den Precompiler ein. Damit ist gewährleistet, dass nur Systeme, die POSIX-Thread-kompatibel sind, die zusätzliche Funktionalität benutzen. Für alle anderen Plattformen verhält sich das Plugin so wie bisher. Der Ersatz für die alte Funktion *stat_path()* sieht dann so aus:

```
void
stat_path (struct parameter_list *p)
{
#ifdef HAVE_PTHREAD_H
  pthread_t stat_thread;
  int status;
  int statdone = 0;
  int timer = timeout_interval;
  struct timespec req, rem;
  req.tv_sec = 0;
  pthread_create(&stat_thread, NULL, do_stat_path, p);
  while (timer-- > 0) {
    req.tv_nsec = 10000000; /* sleep 1/100s */
    nanosleep(&req, &rem);
    if (pthread_kill(stat_thread, 0)) {
      statdone = 1;
      break;
    } else {
      req.tv_nsec = 990000000; /* sleep 99/100s */
      nanosleep(&req, &rem);
    }
  }
  if (statdone == 1) { /* thread ended normally */
    pthread_join(stat_thread, (void *)&status);
  } else { /* thread is still running. kill it */
    pthread_detach(stat_thread);
    if (verbose >= 3)
      printf("stat did not return within %ds on %s\n", timeout_interval,
p->name
);
    printf("DISK %s - ", _("CRITICAL"));
    die (STATE_CRITICAL, _("%s %s: %s\n"), p->name, _("hangs"), _("Time-
out"));
  }
#else
  do_stat_path(p);
#endif
}

void
do_stat_path (struct parameter_list *p)
{
  /* Stat entry to check that dir exists and is accessible */
  if (verbose >= 3)
    printf("calling stat on %s\n", p->name);
  if (stat (p->name, &stat_buf[0])) {
    if (verbose >= 3)
      printf("stat failed on %s\n", p->name);
    printf("DISK %s - ", _("CRITICAL"));
```

```
    die (STATE_CRITICAL, _("%s %s: %s\n"), p->name, _("is not accessible"),
stre
rror(errno));
  }
}
```

Wie man sieht, wurde *stat_path()* in *do_stat_path()* umbenannt. Wenn *HAVE_PTHREAD_H* nicht definiert ist, also das ausführende Betriebssystem die Voraussetzungen für die neue Funktionalität nicht erfüllt, dann wird einfach der alte Code ausgeführt. Im Normalfall hingegen wird mit *pthread_create()* ein eigener Thread erzeugt, der die Routine *do_stat_path()* ausführt. Danach wird in sekündlichen Abständen mit *pthread_kill(stat_thread, 0)* geprüft, ob der Thread seine Aufgabe erfüllt hat und darauf wartet, vom Hauptprogramm „entlassen" zu werden. Die Verzögerung mittels *nanosleep* dauert eine Hundertstel Sekunde und lässt normal funktionierenden Filesystemen genügend Zeit, damit die entsprechenden Threads bereits beim ersten *pthread_kill*-Aufruf als beendet erkannt werden. Nur wenn dieses kurze Intervall nicht gereicht hat, um *do_stat_path* auszuführen, wird eine weitere Pause von 99 Hundertstel Sekunden eingelegt und danach ein erneuter Versuch gestartet. Wenn nach Ablauf des Timeouts von defaultmäßig 10 s ein Thread immer noch läuft, kann man davon ausgehen, dass der *stat*-Systemcall blockiert wurde und möglicherweise nie wieder zurückkehrt. In dem Fall wird mit *pthread_detach()* wenigstens der allokierte Speicher freigegeben und der Thread so weit wie möglich zerstört. Danach beendet sich das Plugin mit einer *CRITICAL*-Fehlermeldung. Am Dateianfang muss noch der Prototyp für die neue Funktion *do_stat_path()* eingefügt werden, sowie mittels einer *#include*-Direktive die Headerdatei *pthread.h* geladen werden. Diese beiden Änderungen wurden hier aus Gründen der Übersichtlichkeit nicht aufgelistet. Sie werden später beim Erzeugen des Patches sichtbar werden.

Neben der Datei *plugins/check_disk.c* muss man auch noch *configure.in* anpassen. Diese Datei ist die Grundlage des **configure**-Kommandos. Dieses soll später ermitteln, ob der ausführende Computer POSIX Threads kennt, bzw. ob die benötigten Headerfiles und Bibliotheken installiert sind. Dazu fügt man folgende Zeilen in *configure.in* ein:

```
dnl Check for POSIX thread libraries
AC_CHECK_HEADERS(pthread.h)
AC_CHECK_LIB(pthread,pthread_create,THRLIBS="-lpthread")
AC_SUBST(THRLIBS)
```

Durch diese zusätzlichen Anweisungen ergeben sich bei der späteren Ausführung von **./configure** Änderungen an zwei Stellen In der Datei *config.h* erscheint die Zeile

```
/* Define to 1 if you have the <pthread.h> header file. */
#define HAVE_PTHREAD_H 1
```

Dies ist das Flag, das in *check_disk.c* dafür verantwortlich ist, dass der aktualisierte Code übersetzt wird. Kennt ein Betriebssystem keine POSIX Threads, dann bleibt *HAVE_PTHREAD_H* undefiniert und das Plugin wird ohne die neue Funktionalität kompiliert. Da-

neben wird noch festgestellt, ob die Bibliothek *libpthread* existiert und falls ja, die Variable *THRLIBS* auf den Wert *–lpthread* gesetzt. Diese Einstellung wird benötigt, damit der Linker die Thread-Library zum fertigen Plugin **check_disk** dazubindet. Dazu muss man auch noch die Datei *plugins/Makefile.am* ändern, indem man *THRLIB* zur Liste der für **check_disk** benötigten Objectfiles und Libraries hinzufügt.

```
check_disk_LDADD = $(BASEOBJS) popen.o $(THRLIBS)
```

Da nicht nur C-Sourcen verändert wurden, sondern auch die für den Build-Prozess wichtige Datei *configure.in*, muss man mit dem Kommando **autoreconf** das Script **configure** neu erzeugen. Dieser Schritt setzt die Installation zweier RPMs voraus:

```
nagsrv# yum install autoconf
nagsrv# yum install libtool
```

Danach kann man die Übersetzung starten, an deren Ende ein Plugin **check_disk** entstehen wird, das die neue Funktionalität beinhaltet.

```
nagsrv# autoreconf
nagsrv# ./configure;make
```

Bevor man etwas veröffentlicht, sollte man es ausführlich testen. Zumindest auf den geläufigsten Plattformen sollte es funktionieren. Da die Nagios-Plugins auf den verschiedensten Betriebssystemen eingesetzt werden, ist es selbst der Entwicklermannschaft um Ton Voon nicht möglich, auf jedem von ihnen Testläufe durchzuführen. (**check_fs_ping** wurde unter Linux 2.6 und Solaris10 erfolgreich compiliert und eingesetzt). Zunächst mountet man sich auf den Testmaschinen ein NFS-Filesystem:

```
nagsrv# uname -a
Linux nagsrv1 2.6.18-92.1.22.el5 #1 SMP Tue Dec 16 11:57:43 EST 2008 x86_64
x86_64 x86_64 GNU/Linux
nagsrv# mount nas.naprax.de:/mnt/md1/db2 /mnt

solsrv1# uname -a
SunOS solsrv1 5.10 Generic_137138-09 i86pc i386 i86pc
solsrv1# mount nas.naprax.de:/mnt/md1/db2 /mnt
```

Danach prüft man, ob sich das Plugin so verhält, wie es das ohne die Modifikationen getan hätte.

```
nagsrv$ check_disk -w 2G -c 1G /mnt
DISK OK - free space: /mnt 67815 MB (7% inode=99%);| /mnt=823060
MB;938549;938550;0;938551

solsrv1$ check_disk -w 2G -c 1G -p /mnt
DISK OK - free space: /mnt 67815 MB (7% inode=99%);| /mnt=823060
MB;938549;938550;0;938551
```

Das Plugin sollte aber nicht nur funktionieren, wenn alles in Ordnung ist. Wichtig ist natürlich, dass es den Fehlerfall erkennt, im konkreten Beispiel ein hängendes NFS-Filesystem. Deshalb wird zum Testen das Storage-Device kurzerhand abgeschaltet. Sollte man kein Gerät haben, das man beliebig außer Betrieb setzen kann, genügt es auch mit Hilfe einer manipulierten Route die IP-Adresse des NFS-Servers unerreichbar werden zu lassen.

```
nagsrv$ check_disk -w 2G -c 1G /mnt
DISK CRITICAL - /mnt hangs: Timeout

solsrv1$ check_disk -w 2G -c 1G -p /mnt
DISK CRITICAL - /mnt hangs: Timeout
```

Sinnvollerweise finden die Tests in einem dritten Verzeichnis statt, da durch die **configure**- und **make**-Läufe eine Menge Dateien verändert werden, die den Patch unnötig aufblähen würden. Erst wenn alles fehlerfrei funktioniert, kopiert man die o. g. Dateien *plugins/check_disk.c*, *configure.in* und *plugins/Makefile.am* in das Verzeichnis *nagios-plugins-1.4.13.nfs*, das man durch Kopieren der Original-Quellen erzeugt hat. Dadurch hat man zwei Verzeichnisbäume. Einen „sauberen", der die Originalquellen enthält und einen modifizierten mit den eigenen Anpassungen. Danach kann man die Unterschiede zwischen den beiden Releases mit dem **diff**-Befehl ermitteln. Dabei wechselt man in das Verzeichnis, in dem Original- und modifizierte Sourcen nebeneinander liegen. Der Aufruf lautet **diff -Naur <originalverzeichnis> <modifiziertes verzeichnis>**

```
nagsrv$ diff -Naur nagios-plugins-1.4.13 nagios-plugins-1.4.13.nfs
diff -Naur nagios-plugins-1.4.13/configure.in nagios-plugins-1.4.13.nfs/con-
figure.in
--- nagios-plugins-1.4.13/configure.in  2008-09-25 10:15:58.000000000 +0200
+++ nagios-plugins-1.4.13.nfs/configure.in      2009-04-30
17:55:06.000000000 +0200
@@ -453,6 +453,11 @@
        with_gnutls="no"
 fi

+dnl Check for POSIX thread libraries
+AC_CHECK_HEADERS(pthread.h)
+AC_CHECK_LIB(pthread,pthread_create,THRLIBS="-lpthread")
+AC_SUBST(THRLIBS)
+
 dnl
 dnl Checks for header files.
 dnl
diff -Naur nagios-plugins-1.4.13/plugins/check_disk.c nagios-plugins-1.4.13.
nfs/plugins/check_disk.c
--- nagios-plugins-1.4.13/plugins/check_disk.c  2008-07-10
12:03:55.000000000 +0200
+++ nagios-plugins-1.4.13.nfs/plugins/check_disk.c      2009-04-30
17:55:01.000000000 +0200
@@ -55,6 +55,9 @@
 # include <limits.h>
 #endif
 #include "regex.h"
```

```
+#if HAVE_PTHREAD_H
+# include <pthread.h>
+#endif


 /* If nonzero, show inode information. */
@@ -129,6 +132,7 @@
 void print_usage (void);
 double calculate_percent(uintmax_t, uintmax_t);
 void stat_path (struct parameter_list *p);
+void do_stat_path (struct parameter_list *p);

 double w_dfp = -1.0;
 double c_dfp = -1.0;
@@ -993,6 +997,42 @@
 void
 stat_path (struct parameter_list *p)
 {
+#ifdef HAVE_PTHREAD_H
+  pthread_t stat_thread;
+  int status;
+  int statdone = 0;
+  int timer = timeout_interval;
+  struct timespec req, rem;
+  req.tv_sec = 0;
+  pthread_create(&stat_thread, NULL, do_stat_path, p);
+  while (timer-- > 0) {
+    req.tv_nsec = 10000000;
+    nanosleep(&req, &rem);
+    if (pthread_kill(stat_thread, 0)) {
+      statdone = 1;
+      break;
+    } else {
+      req.tv_nsec = 990000000;
+      nanosleep(&req, &rem);
+    }
+  }
+  if (statdone == 1) {
+    pthread_join(stat_thread, (void *)&status);
+  } else {
+    pthread_detach(stat_thread);
+    if (verbose >= 3)
+      printf("stat did not return within %ds on %s\n", timeout_interval,
p->name);
+    printf("DISK %s - ", _("CRITICAL"));
+    die (STATE_CRITICAL, _("%s %s: %s\n"), p->name, _("hangs"), _("Time-
out"));
+  }
+#else
+  do_stat_path(p);
+#endif
+}
+
+void
+do_stat_path (struct parameter_list *p)
+{
```

```
   /* Stat entry to check that dir exists and is accessible */
   if (verbose >= 3)
     printf("calling stat on %s\n", p->name);
diff -Naur nagios-plugins-1.4.13/plugins/Makefile.am nagios-plugins-1.4.13.
nfs/plugins/Makefile.am
--- nagios-plugins-1.4.13/plugins/Makefile.am   2008-07-08
11:31:04.000000000 +0200
+++ nagios-plugins-1.4.13.nfs/plugins/Makefile.am       2009-04-30
17:54:54.000000000 +0200
@@ -54,7 +54,7 @@
 check_apt_LDADD = $(BASEOBJS) runcmd.o
 check_cluster_LDADD = $(BASEOBJS)
 check_dig_LDADD = $(NETLIBS) runcmd.o
-check_disk_LDADD = $(BASEOBJS) popen.o
+check_disk_LDADD = $(BASEOBJS) popen.o $(THRLIBS)
 check_dns_LDADD = $(NETLIBS) runcmd.o
 check_dummy_LDADD = $(BASEOBJS)
 check_fping_LDADD = $(NETLIBS) popen.o
```

Die Ausgabe dieses Kommandos zeigt in einem speziellen Format die Unterschiede zwischen den beiden Dateibäumen. Sie enthält Anweisungen, wie man durch Löschen und Einfügen von Zeilen von der alten Version zur neuen kommt. Zeilen, die mit einem Minuszeichen beginnen, müssen entfernt werden, wohingegen Zeilen, die mit einem Pluszeichen beginnen, neu sind und eingefügt werden müssen. Wie man an den mit *diff* beginnenden Zeilen sieht, sind Änderungen für die drei besprochenen Dateien *plugins/check_disk.c*, *configure.in* und *plugins/Makefile.am* enthalten. So kann man also beliebig viele Files in einem Projekt anpassen und die Änderungen in einer einzigen Datei zusammenfassen. Man leitet diese Ausgabe in eine Datei um und kann sie dann später zusammen mit dem **patch**-Kommando verwenden, um die Modifikationen an den Originalsourcen vorzunehmen.

```
diff -Naur nagios-plugins-1.4.13 nagios-plugins-1.4.13.nfs \
    > nagios-plugins-1.4.13.check_disks.nfs.patch
```

Diese Patchdateien sind auch die im OpenSource-Bereich übliche Art, um Korrekturen an die Projektverantwortlichen zu schicken. Das Schöne daran ist, dass man auf einen Blick sieht, an welchen Stellen etwas geändert wurde. Natürlich muss man jetzt noch prüfen, ob die Anwendung des Patches fehlerfrei funktioniert. Dazu wechselt man in das Verzeichnis mit den Originalsourcen.

```
nagsrv$ cd ../nagios-plugins-1.4.13
```

Der **patch**-Befehl liest dann die Liste der Änderungsanweisungen ein und nimmt die geforderten Lösch- und Einfügeaktionen an den betreffenden Dateien vor. Am besten ruft man ihn zuerst einmal mit dem Parameter *--dry-run* auf. Damit werden die Änderungen nicht tatsächlich vollzogen, sondern nur geprüft, ob die Informationen in der Patchdatei sich auch wirklich anwenden lassen oder ob es zu Konflikten kommen wird.

```
nagsrv$ patch --dry-run --strip 1 \
    <../nagios-plugins-1.4.13.check_disks.nfs.patch
patching file configure.in
patching file plugins/check_disk.c
patching file plugins/Makefile.am
```

Nachdem der Trockenlauf fehlerfrei war, wiederholt man das Kommando, aber diesmal ohne *--dry-run*. Dadurch wurde das Verzeichnis mit den Originalsourcen auf den neuen Stand gebracht und enthält alle vorgenommenen Änderungen an den genannten drei Dateien. Nun muss man noch den Patch einreichen. Er wird dann begutachtet, auf Herz und Nieren getestet und hoffentlich Bestandteil des nächsten Releases der Nagios-Plugins sein.

Dazu öffnet man die SourceForge-Webseite *http://sourceforge.net/projects/nagiosplug*, wo das Projekt *Nagios Plugin Development* beheimatet ist. Dort findet man unter dem Menüpunkt *Tracker* eine Anwendung, in der man Bugs melden, neue Features vorschlagen und Patches hochladen kann.

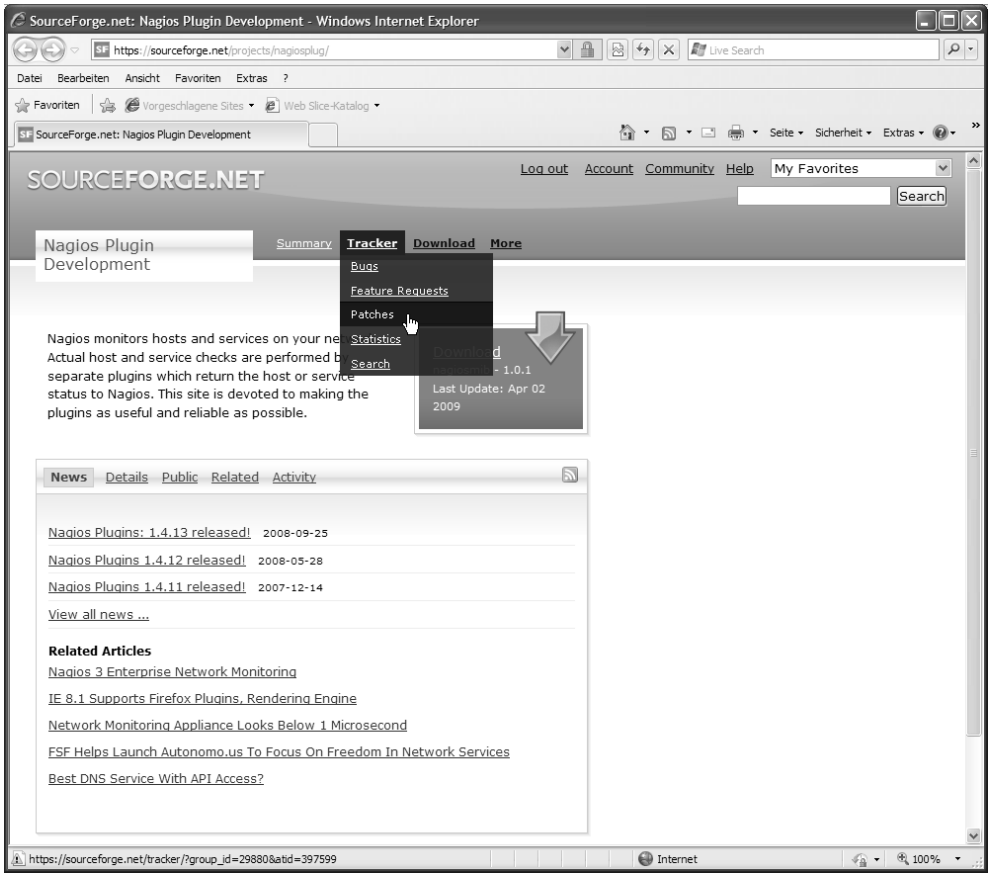


Abbildung 3.5: **Der Tracker bietet die Möglichkeit, Patches hochzuladen**

Danach klickt man auf den Link *Add new* und füllt ein Formular aus. Die Beschreibung sollte die erzielte Verbesserung umfassen, sowie die Plattformen, auf denen der Patch getestet wurde. Wenn alles gut geht, erscheint irgendwann ein neues Release der Nagios-Plugins (oder eines anderen Plugins, das man erweitert und dem Autor einen Patch geschickt hat). Dadurch erreicht man, dass allen anderen Benutzern ebenfalls die neuen Funktionen zur Verfügung stehen. Für Firmen ist noch wichtig, dass man statt einer individuellen Anpassung wieder auf die „offizielle" Version eines Plugins zurückgreifen kann.

```
From: Armin Admin
To: Bernd Berserker
Subject: Re: NFS-Probleme

Hallo Bernd,

wenn wir Glück haben, wird mein Patch akzeptiert und wandert in die of-
fiziellen Nagios-Plugins. Damit brauchen wir keine Services mehr, die auf
check_fs_ping aufsetzen, sondern können unsere Anforderung mit check_disk
abdecken. Wir hätten dann zweierlei erreicht: Naprax steht als Wohltäter da
und wir können unsere Nagios-Konfiguration einfach halten.

Gruss,
Armin

p.s. Das Einsenden von Patches ist aber nur dann sinnvoll, wenn möglichst
viele andere Anwender auch etwas davon haben. Glaub jetzt aber nicht, ich
könne alle unsere teils abstrusen, firmeninternen Anforderungen in ein offi-
zielles Release reindrücken. Die werden mir was pfeifen.
```