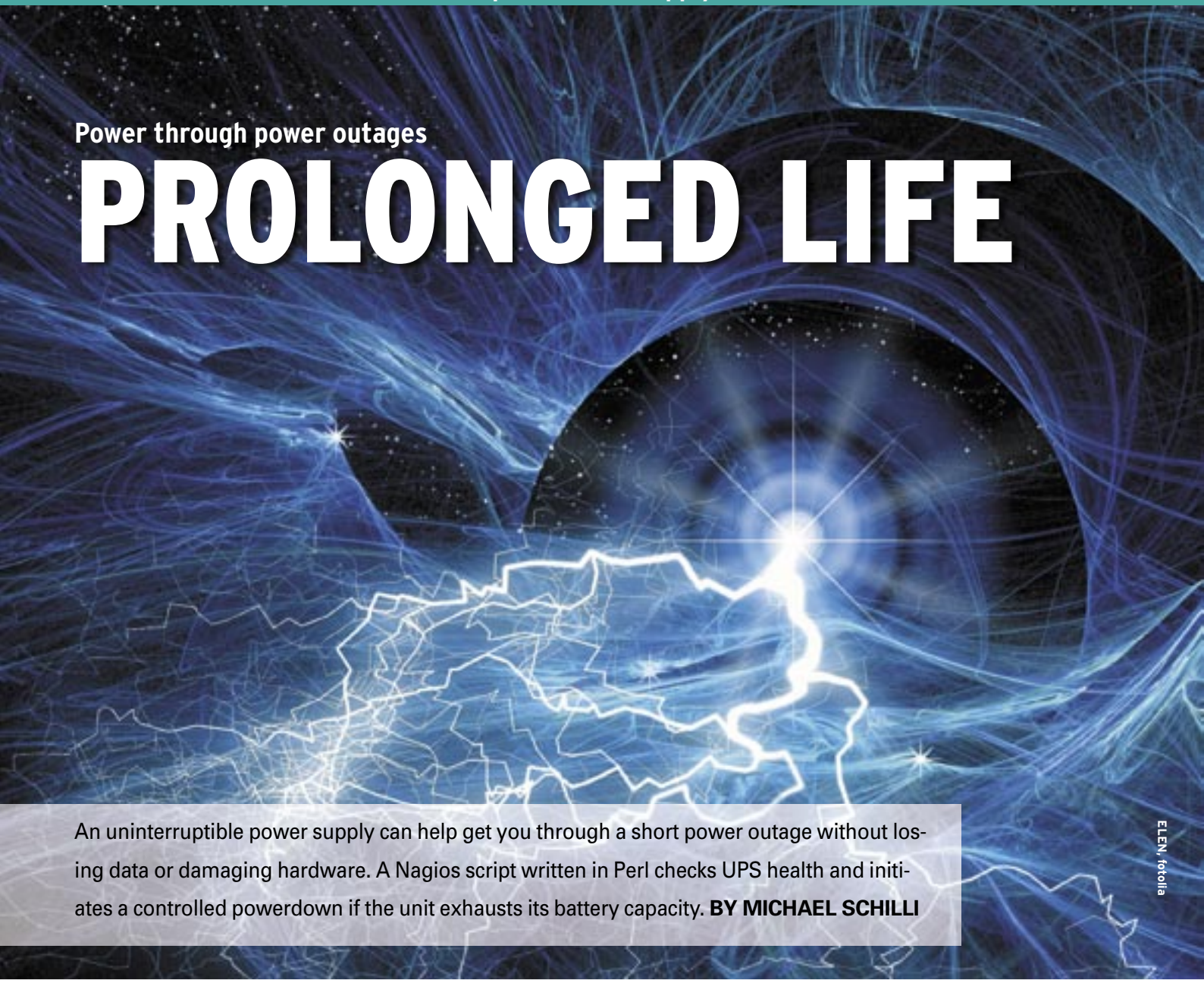


Power through power outages

PROLONGED LIFE

An uninterruptible power supply can help get you through a short power outage without losing data or damaging hardware. A Nagios script written in Perl checks UPS health and initiates a controlled powerdown if the unit exhausts its battery capacity. **BY MICHAEL SCHILLI**

ELEN, fotolia

While going through my drawers full of old computer hardware, I found a cheap, old, uninterruptible power supply (UPS). I must have bought it because it was on sale, since the "Cyberpower 325SL" only provides 185 watts for about five minutes. But I figured that's good enough for helping a desktop PC, a DSL modem, and a router go through a short power outage unharmed.

Send Like It's 1999

The UPS features a serial interface jack (Figure 1) that can be connected to a PC with a serial cable. That's what I call retro! The unit also came with Windows-only software that I remember discarding way back, right after opening the package. When searching the web, however, I found the NUT project [1], which offers drivers for all kinds of UPS systems and a daemon to communicate with the UPS unit through the drivers.

Installing the NUT software was a breeze; the documentation that came with the distribution is stellar. After compiling and running *make install*, three configuration files must be created. First, the *ups.conf* file sets the parameters for the UPS used (Figure 2). In my case, I chose the *genericups* driver, which works with cheap UPS units and just provides basic online/offline status without fancy UPS stats or battery capacity left. Looking at the driver documentation revealed that, for CyberPower units, "type 7" has to be used. I called the device "elcheapo," and that's how it is going to be referenced later on when its status is checked.

Because I plugged the serial cable into the second serial port of my PC, the configured port is */dev/ttyS1*. Had I plugged it into the first serial jack, */dev/ttyS0* would have been the port.

Limit Access

The daemon configuration file *upsd.conf* defines rules on who can access the UPS data from the NUT daemon (Figure 3). I opened the file for my PC's static IP address, and the */32* at the end defines that only status data can be read.

For home use, defining user-based access via *upsd.users* is probably overkill, but the file needs to be there, so an empty file will do.

You could define a new user "nut" with an associated group, but I decided to let the daemon run as the default user *nobody*, on whose behalf we need to create a state directory:

```
# mkdir /var/state/ups
# chown nobody /var/state/ups
```

Figure 1: The UPS system with two power plugs on top and one serial cable plugged in on the right. The "Kill-A-Watt" meter indicates that PC, DSL modem, and router combined are using about 114 watts.

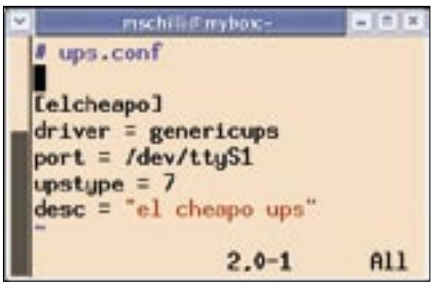


Figure 2: UPS configuration in /usr/local/ups/etc/ups.conf.

Figure 3: NUT daemon configuration in /usr/local/ups/etc/upsd.conf.

```
# chmod 700 /var/state/ups
```

Next, the driver daemon and then the NUT daemon must be started as root:

```
# /usr/local/ups/bin/➥
upsdrvctl start
# /usr/local/ups/sbin/upsd
```

If the output from these commands indicates success, a quick test with the *upsc* utility (which comes with NUT) will reveal that the UPS is online, drawing juice from the power outlet:

```
$ upsc elcheapo@localhost➥
ups.status
OL
```

When unplugging the UPS so that it powers the PC from its battery, the above *upsc* call will return *OB* instead of returning *OL*.

Over in Nagios Land

How should you monitor the UPS? Regular readers will remember that I've talked about Nagios in this column before [2]. Nagios watches over all kinds of systems in my home, including room temperature, hard-disk capacity, and the performance of the hosting service I'm using for my websites. So, I decided to add a UPS watcher as just another Nagios task to my existing setup (Figure 4).

The script in Listing 1, *check_myups*, uses the *upsc* utility mentioned above to query the UPS status but adds a wrapper so that the script can be used as a Nagios plugin. It uses a few extra Perl modules, which can be downloaded from CPAN. If the UPS is up and the check goes well, the script prints *UPS OK - OL* and returns an exit code of 0. If the UPS is on battery power, the script returns *UPS CRITICAL - OB* and exits with exit code 2 to tell Nagios about the problem.

And ... Action!

Nagios follows the notion of "soft" and "hard" status changes. If a check indicates a critical condition for the first time but the parameter *max_check_attempts* is set to 3, Nagios makes a note of the problem but sets the status to SOFT first. When subsequent checks *retry_check_interval* seconds later also fail and *max_check_attempts* is finally reached, the status is set to HARD and the notification mechanisms kick in. It's possible to configure emails to be sent out to

Listing 1: check_myups

```
01 #!/usr/bin/perl
02 ##################################
03 # check_myups
04 # Mike Schilli, 2007
05 ##################################
06 use strict;
07 use Log::Log4perl qw(:easy);
08 use Nagios::Clientstatus;
09
10 my $version = "0.01";
11 my $ncli    =
12   Nagios::Clientstatus->new(
13   help_subref => sub {
14    print "usage: $0\n";
15   },
16   version       => $version,
17   mandatory_args => [],
18    );
19
20 my $data = `upsc elcheapo\
     @localhost ups.status`;
21
22 chomp $data;
23 my $status = "ok";
24
25 if ($data eq "OB") {
26   $status = "critical";
27 }
28
29 print "UPS ", uc($status),
30    " - $data\n";
31
32 exit $ncli->exitvalue(
33   $status);
```

provide a rude awakening to the system administrator on call.

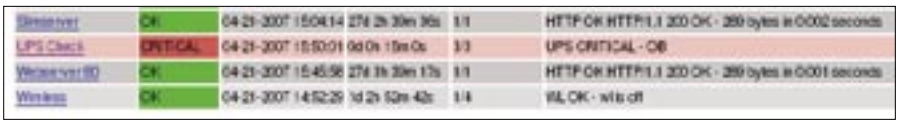
Nagios also allows you to initiate actions when status changes occur. This way, scripts can try to fix the problem, and if the solution is as simple as restarting a web server, it's probably a good idea to do that instead of bothering a system administrator. In the case of a UPS at home running out of battery power, we just want the Linux system to shut itself down properly to avoid a hard landing. The event handler in Listing 2, *powerdown*, does exactly that.

```
##############################
# ups Plugin
##############################
define command{
  command_name  ups
  command_line  $USER1$/check_myups
}

##############################
# ups Service
##############################
define service{
  use                  ez-service
  host_name            mybox
  normal_check_interval  1
  retry_check_interval   1
  event_handler        powerdown
  max_check_attempts     2
  service_description UPS Check
  check_command        ups
}

##############################
# powerdown
##############################
define command{
    command_name         powerdown
    command_line         $USER1$/powerdo
wn $SERVICESTATE$ $SERVICESTATETYPE$
                                254,1         53%
```

Figure 4: Nagios configuration for the UPS check and its powerdown event handler.



Webserver	OK	04-21-2007 15:04:14	27d 2h 39m 36s	1/1	HTTP OK HTTP/1.1 200 OK - 289 bytes in 0.002 seconds
UPS Check	CRITICAL	04-21-2007 15:50:01	0d 0h 15m 0s	1/3	UPS CRITICAL - OB
Webserver80	OK	04-21-2007 15:45:58	27d 1h 39m 17s	1/1	HTTP OK HTTP/1.1 200 OK - 289 bytes in 0.001 seconds
Wireless	OK	04-21-2007 14:52:29	1d 2h 52m 42s	1/4	WL OK - wlo off

Figure 5: Nagios shows that the UPS is running on battery power.

But be careful, Nagios calls the script defined with the event handler directive with every status change: first, when the status changes from OK to CRITICAL/SOFT, then when it changes from CRITICAL/SOFT to CRITICAL/HARD, and finally after it recovers from CRITICAL HARD to OK.

The *powerdown* script, which gets the status information that is passed in, makes sure that a poweroff of the machine only happens in case a critical hard state is encountered and ignores all other requests.

Per the configuration in Figure 4, Nagios passes in the two variables SERVICESTATE (ok/critical) and SERVICESTATETYPE (soft/hard) so *powerdown* can make an informed decision.

If I'm out of battery power, the event handler will want to run the Linux *poweroff* script, but only root can do that. To make sure that the *nagios* user running the Nagios application has permission to do that, the following entry is added to the *sudoers* file:

```
# /etc/sudoers
nagios ALL= NOPASSWD:\
/usr/bin/poweroff
```

With this setting, the *nagios* user can run *sudo /usr/bin/poweroff* as root without having to type a password. Both scripts, *check_myups* and *powerdown*, have to be installed in the Nagios plugin directory, */usr/local/nagios/libexec*. When the changes to the Nagios configuration have been made, Nagios needs to be restarted.

Now you'll be ready when the next power outage comes! ■

INFO

[1] Network UPS Tools (NUT): *http://www.networkupstools.org/*

[2] "The Watcher" by Michael Schilli. *Linux Magazine*, June 2006: *http://www.linux-magazine.com/issue/67/Perl_Nagios_Plugins.pdf*

[3] Turnbull, James. *Pro Nagios 2.0*. Apress, 2006.

[4] Listings for this article: *http://www.linux-magazine.com/Magazine/Downloads/80*

THE AUTHOR

Michael Schilli works as a Software Developer at Yahoo!, Sunnyvale, California. He wrote "Perl Power" for Addison-Wesley and can be contacted at *mschilli@perlmeister.com*. His homepage is at *http://perlmeister.com*.

Listing 2: powerdown

```
01 #!/usr/bin/perl -w
02 ##############################
03 # powerdown event handler
04 # Mike Schilli, 2007
05 ##############################
06 use strict;
07 use Sysadm::Install qw(:all);
08 use Log::Log4perl qw(:easy);
09
10 Log::Log4perl->easy_init({
11   file =>
12       ">>/tmp/powerdown.log",
13   level => $DEBUG
14 });
15
16 my ($state, $softhard) =
17   @ARGV;
18
19 LOGDIE
20  "usage: $0 state SOFT|HARD"
21  if !$softhard
22  or $softhard !~ /SOFT|HARD/;
23
24 DEBUG
25 "Called $0 $state $softhard";
26
27 if ($state eq "OK") {
28  DEBUG "Ignoring OK";
29  exit 0;
30 }
31
32 if ($softhard eq "SOFT") {
33  DEBUG "Ignoring soft mode";
34  exit 0;
35 }
36
37 # Shut PC off
38 INFO "Shutting down";
39 tap("sudo",
40     "/usr/bin/poweroff");
```