Version 5.1.1

NAME

Monit - utility for monitoring services on a Unix system

SYNOPSIS

monit [options] {arguments}

DESCRIPTION

monit is a utility for managing and monitoring processes, files, directories and filesystems on a Unix system. Monit conducts automatic maintenance and repair and can execute meaningful causal actions in error situations. E.g. Monit can start a process if it does not run, restart a process if it does not respond and stop a process if it uses too much resources. You may use Monit to monitor files, directories and filesystems for changes, such as timestamps changes, checksum changes or size changes.

Monit is controlled via an easy to configure control file based on a free-format, token-oriented syntax. Monit logs to syslog or to its own log file and notifies you about error conditions via customizable alert messages. Monit can perform various TCP/IP network checks, protocol checks and can utilize SSL for such checks. Monit provides a http(s) interface and you may use a browser to access the Monit program.

GENERAL OPERATION

The behavior of Monit is controlled by command-line options *and* a run control file, *~/.monitrc*, the syntax of which we describe in a later section. Command-line options override *.monitrc* declarations.

The following options are recognized by monit. However, it is recommended that you set options (when applicable) directly in the *.monitrc* control file.

General Options and Arguments

-c *file* Use this control file

-d *n* Run as a daemon once per *n* seconds

-g Set group name for start, stop, restart, monitor and unmonitor.

-l *logfile* Print log information to this file

-p *pidfile* Use this lock file in daemon mode

-s *statefile* Write state information to this file

-I Do not run in background (needed for run from init)

-t Run syntax check for the control file

-v Verbose mode, work noisy (diagnostic output)

-H *[filename]* Print MD5 and SHA1 hashes of the file or of stdin if the filename is omitted; Monit will exit afterwards

-V Print version number and patch level

-h Print a help text

In addition to the options above, Monit can be started with one of the following action arguments; Monit will then execute the action and exit without transforming itself to a daemon.

start all Start all services listed in the control file and enable monitoring for them. If the group option is set, only start and enable monitoring of services in the named group (no "all" verb is required in this case).

start name Start the named service and enable monitoring for it. The name is a service entry name from the monitrc file.

stop all Stop all services listed in the control file and disable their monitoring. If the group option is set, only stop and disable monitoring of the services in the named group (no "all" verb is required in this case).

stop name Stop the named service and disable its monitoring. The name is a service entry name from the monitrc file.

restart all Stop and start *all* services. If the group option is set, only restart the services in the named group (no "all" verb is required in this case).

restart name Restart the named service. The name is a service entry name from the monitrc file.

monitor all Enable monitoring of all services listed in the control file. If the group option is set, only start monitoring of services in the named group (no "all" verb is required in this case).

monitor name Enable monitoring of the named service. The name is a service entry name from the monitrc file. Monit will also enable monitoring of all services this service depends on.

unmonitor all Disable monitoring of all services listed in the control file. If the group option is set, only disable monitoring of services in the named group (no "all" verb is required in this case).

unmonitor name Disable monitoring of the named service. The name is a service entry name from the monitrc file. Monit will also disable monitoring of all services that depends on this service.

status Print full status information for each service.

summary Print short status information for each service.

reload Reinitialize a running Monit daemon, the daemon will reread its configuration, close and reopen log files.

quit Kill a Monit daemon process

validate Check all services listed in the control file. This action is also the default behavior when Monit runs in daemon mode.

WHAT TO MONITOR

You may use Monit to monitor daemon processes or similar programs running on localhost. Monit is particular useful for monitoring daemon processes, such as those started at system boot time from /etc/init.d/. For instance sendmail, sshd, apache and mysql. In difference to many monitoring systems, Monit can act if an error situation should occur, e.g.; if sendmail is not running, monit can start sendmail or if apache is using too much resources (e.g. if a DoS attack is in progress) Monit can stop or restart apache and send you an alert message. Monit can also monitor process characteristics, such as; if a process has become a zombie and how much memory or cpu cycles a process is using.

You may also use Monit to monitor files, directories and filesystems on localhost. Monit can monitor these items for changes, such as timestamps changes, checksum changes or size changes. This is also useful for security reasons - you can monitor the md5 checksum of files that should not change.

You may even use Monit to monitor remote hosts. First and foremost Monit is a utility for monitoring and mending services on localhost, but if a service depends on a remote service, e.g. a database server or an application server, it might by useful to be able to test a remote host as well.

You may monitor the general system-wide resources such as cpu usage, memory and load average.

HOW TO MONITOR

Monit is configured and controlled via a control file called **monitrc**. The default location for this file is ~/.monitrc. If this file does not exist, Monit will try /etc/monitrc, then @sysconfdir@/monitrc and finally ./monitrc.

A Monit control file consists of a series of service entries and global option statements in a free-format, token-oriented syntax. Comments begin with a # and extend through the end of the line. There are three kinds of tokens in the control file: grammar keywords, numbers and strings.

On a semantic level, the control file consists of three types of statements:

1. **Global set-statements**

   A global set-statement starts with the keyword *set* and the item to configure.

2. **Global include-statement**

   The include statement consists of the keyword *include* and a glob string.

3. **One or more service entry statements.**

   A service entry starts with the keyword *check* followed by the service type.

A Monit control file example:

```
#
# Monit control file
#

set daemon 120 # Poll at 2-minute intervals
set logfile syslog facility log_daemon
set alert foo@bar.baz
set httpd port 2812 and use address localhost
    allow localhost   # Allow localhost to connect
    allow admin:Monit # Allow Basic Auth

check system myhost.mydomain.tld
   if loadavg (1min) > 4 then alert
   if loadavg (5min) > 2 then alert
   if memory usage > 75% then alert
   if cpu usage (user) > 70% then alert
   if cpu usage (system) > 30% then alert
   if cpu usage (wait) > 20% then alert

check process apache
   with pidfile "/usr/local/apache/logs/httpd.pid"
   start program = "/etc/init.d/httpd start" with timeout 60 seconds
   stop program = "/etc/init.d/httpd stop"
   if 2 restarts within 3 cycles then timeout
   if totalmem > 100 Mb then alert
   if children > 255 for 5 cycles then stop
   if cpu usage > 95% for 3 cycles then restart
   if failed port 80 protocol http then restart
   group server
   depends on httpd.conf, httpd.bin

check file httpd.conf
    with path /usr/local/apache/conf/httpd.conf
    # Reload apache if the httpd.conf file was changed
    if changed checksum
       then exec "/usr/local/apache/bin/apachectl graceful"

check file httpd.bin
    with path /usr/local/apache/bin/httpd
    # Run /watch/dog in the case that the binary was changed
    # and alert in the case that the checksum value recovered
    # later
```

```
    if failed checksum then exec "/watch/dog"
       else if recovered then alert

include /etc/monit/mysql.monitrc
include /etc/monit/mail/*.monitrc
```

The above example illustrates a service entry for monitoring the apache web server process as well as related files. The meaning of the various statements will be explained in the following sections.

LOGGING

Monit will log status and error messages to a log file. Use the *set logfile* statement in the monitrc control file. To setup Monit to log to its own logfile, use e.g. *set logfile /var/log/monit.log*. If **syslog** is given as a value for the *-l* command-line switch (or the keyword *set logfile syslog* is found in the control file) Monit will use the **syslog** system daemon to log messages with a priority assigned to each message based on the context. To turn off logging, simply do not set the logfile in the control file (and of course, do not use the -l switch)

DAEMON MODE

The *-d interval* command-line switch runs Monit in daemon mode. You must specify a numeric argument which is a polling interval in seconds.

In daemon mode, Monit detaches from the console, puts itself in the background and runs continuously, monitoring each specified service and then goes to sleep for the given poll interval.

Simply invoking

```
Monit -d 300
```

will poll all services described in your *~/.monitrc* file every 5 minutes.

It is strongly recommended to set the poll interval in your ~/.monitrc file instead, by using *set daemon **n***, where **n** is an integer number of seconds. If you do this, Monit will always start in daemon mode (as long as no action arguments are given). Example (check every 5 minutes):

```
set daemon 300
```

If you need Monit to wait some time at startup before it start checking services you can use the delay statement. Example (check every 5 minutes, wait 1 minute on start before first monitoring cycle):

```
set daemon 300 with start delay 60
```

Monit makes a per-instance lock-file in daemon mode. If you need more Monit instances, you will need more configuration files, each pointing to its own lock-file.

Calling *monit* with a Monit daemon running in the background sends a wake-up signal to the daemon, forcing it to check services immediately.

The *quit* argument will kill a running daemon process instead of waking it up.

INIT SUPPORT

Monit can run and be controlled from *init*. If Monit should crash, *init* will re-spawn a new Monit process. Using init to start Monit is probably the best way to run Monit if you want to be certain that you always have a running Monit daemon on your system. (It's obvious, but never the less worth to stress; Make sure that the control file does not have any syntax errors before you start Monit from init. Also, make sure that if you run monit from init, that you do not start Monit from a startup scripts as well).

To setup Monit to run from init, you can either use the 'set init' statement in monit's control file or use the -I option from the command line and here is what you must add to /etc/inittab:

```
  # Run Monit in standard run-levels
  mo:2345:respawn:/usr/local/bin/monit -Ic /etc/monitrc
```

After you have modified init's configuration file, you can run the following command to re-examine /etc/inittab and start monit:

```
  telinit q
```

```
For systems without telinit:
```

```
  kill -1 1
```

If Monit is used to monitor services that are also started at boot time (e.g. services started via SYSV init rc scripts or via inittab) then, in some cases, a race condition could occur. That is; if a service is slow to start, Monit can assume that the service is not running and possibly try to start it and raise an alert, while, in fact the service is already about to start or already in its startup sequence. Please see the FAQ for solutions to this problem.

INCLUDE FILES

The Monit control file, *monitrc*, can include additional configuration files. This feature helps to maintain a certain structure or to place repeating settings into one file. Include statements can be placed at virtually any spot. The syntax is the following:

```
  INCLUDE globstring
```

The globstring is any kind of string as defined in `glob(7)`. Thus, you can refer to a single file or you can load several files at once. In case you want to use whitespace in your string the globstring need to be embedded into quotes (') or double quotes ("). For example,

```
 INCLUDE "/etc/monit/Monit configuration files/printer.*.monitrc"
```

loads any file matching the single globstring. If the globstring matches a directory instead of a file, it is silently ignored.

INCLUDE statements in included files are parsed as in the main control file.

If the globstring matches several results, the files are included in a non sorted manner. If you need to rely on a certain order, you might need to use single *include* statements.

GROUP SUPPORT

Service entries in the control file, *monitrc*, can be grouped together by the *group* statement. The syntax is simply (keyword in capital):

```
  GROUP groupname
```

With this statement it is possible to group similar service entries together and manage them as a whole. Monit provides functions to start, stop, restart, monitor and unmonitor a group of services, like so:

To start a group of services from the console:

```
  Monit -g <groupname> start
```

To stop a group of services:

```
  Monit -g <groupname> stop
```

To restart a group of services:

```
Monit -g <groupname> restart
```

Note: the *status* and *summary* commands don't support the -g option and will print the state of all services.

Service can be added to multiple groups by adding group statement multiple times:

```
group www
group filesystem
```

MONITORING MODE

Monit supports three monitoring modes per service: *active*, *passive* and *manual*. See also the example section below for usage of the mode statement.

In *active* mode, Monit will monitor a service and in case of problems Monit will act and raise alerts, start, stop or restart the service. Active mode is the default mode.

In *passive* mode, Monit will passively monitor a service and specifically **not** try to fix a problem, but it will still raise alerts in case of a problem.

For use in clustered environments there is also a *manual* mode. In this mode, Monit will enter *active* mode **only** if a service was brought under monit's control, for example by executing the following command in the console:

```
Monit start sybase
(Monit will call sybase's start method and enable monitoring)
```

If a service was not started by Monit or was stopped or disabled for example by:

```
Monit stop sybase
(Monit will call sybase's stop method and disable monitoring)
```

Monit will then not monitor the service. This allows for having services configured in monitrc and start it with Monit only if it should run. This feature can be used to build a simple failsafe cluster. To see how, read more about how to setup a cluster with Monit using the *heartbeat* system in the examples sections below.

A service's monitoring state is persistent across Monit restart. This means that you probably would like to make certain that services in manual mode are stopped or in unmonitored mode at server shutdown. Do for instance the following in a server shutdown script:

```
Monit stop sybase
```

or

```
Monit unmonitor sybase
```

If you use Monit in a HA-cluster you should place the state file in a temporary filesystem so if the machine should crash and the stand-by machine take over services, any manual monitoring mode services that were started on the crashed machine won't be started on reboot. Use for example:

```
set statefile /tmp/monit.state
```

ALERT MESSAGES

Monit will raise an email alert in the following situations:

```
o A service timed out
o A service does not exist
o A service related data access problem
o A service related program execution problem
o A service is of invalid object type
o A icmp problem
o A port connection problem
```

```
 o A resource statement match
 o A file checksum problem
 o A file size problem
 o A file/directory timestamp problem
 o A file/directory/filesystem permission problem
 o A file/directory/filesystem uid problem
 o A file/directory/filesystem gid problem
 o An action is done per administrator's request
```

Monit will send an alert each time a monitored object changed. This involves:

```
 o Monit started, stopped or reloaded
 o A file checksum changed
 o A file size changed
 o A file content match
 o A file/directory timestamp changed
 o A filesystem mount flags changed
 o A process PID changed
 o A process PPID changed
```

You use the alert statement to notify Monit that you want alert messages sent to an email address. If you do not specify an alert statement, Monit will not send alert messages.

There are two forms of alert statement:

```
 o Global - common for all services
 o Local  - per service
```

In both cases you can use more than one alert statement. In other words, you can send many different emails to many different addresses.

Recipients in the global and in the local lists are alerted when a service failed, recovered or changed. If the same email address is in the global and in the local list, Monit will only send one alert. Local (per service) defined alert email addresses override global addresses in case of a conflict. Finally, you may choose to only use a global alert list (recommended), a local per service list or both.

It is also possible to disable the global alerts locally for particular service(s) and recipients.

Setting a global alert statement

If a change occurred on a monitored services, Monit will send an alert to all recipients in the global list who has registered interest for the event type. Here is the syntax for the global alert statement:

SET ALERT mail-address [[NOT] {events}] [MAIL-FORMAT {mail-format}] [REMINDER number]

Simply using the following in the global section of monitrc:

```
 set alert foo@bar
```

will send a default email to the address foo@bar whenever an event occurred on any service. Such an event may be that a service timed out, a service doesn't exist and so on. If you want to send alert messages to more email addresses, add a *set alert 'email'* statement for each address.

For explanations of the *events, MAIL-FORMAT and REMINDER* keywords above, please see below.

You can also use the NOT option ahead of the events list which will reverse the meaning of the list. That is, only send alerts for events *not* in the list. This can save you some configuration bytes if you are interested in most events except a few.

Setting a local alert statement

Each service can also have its own recipient list.

ALERT mail-address [[NOT] {events}] [MAIL-FORMAT {mail-format}] [REMINDER number]

or

NOALERT mail-address

If you only want an alert message sent for certain events and for certain service(s), for example only for timeout events or only if a service died, then postfix the alert-statement with a filter block:

```
check process myproc with pidfile /var/run/my.pid
  alert foo@bar only on { timeout, nonexist }
  ...
```

(*only* and *on* are noise keywords, ignored by Monit. As a side note; Noise keywords are used in the control file grammar to make an entry resemble English and thus make it easier to read (or, so goes the philosophy). The full set of available noise keywords are listed below in the Control File section).

You can also setup to send alerts for all events except some by putting the word *not* ahead of the list. For example, if you want to receive alerts for all events except Monit instance events, you can write (note that the noise words 'but' and 'on' are optional):

```
check system myserver
  alert foo@bar but not on { instance }
  ...
```

instead of:

```
  alert foo@bar on { action
                     checksum
                     content
                     data
                     exec
                     gid
                     icmp
                     invalid
                     fsflags
                     nonexist
                     permission
                     pid
                     ppid
                     size
                     timeout
                     timestamp }
```

This will send alerts for all events to foo@bar, except Monit instance events. An instance event BTW, is an event fired whenever the Monit program start or stop.

Event filtering can be used to send an email to different email addresses depending on the events that occurred. For instance:

```
alert foo@bar { nonexist, timeout, resource, icmp, connection }
alert security@bar on { checksum, permission, uid, gid }
alert manager@bar
```

This will send an alert message to foo@bar whenever a nonexist, timeout, resource or connection problem occurs and a message to security@bar if a checksum, permission, uid or gid problem occurs. And finally, a message to manager@bar whenever any error event occurs.

Here is the list of events you can use in a mail-filter: *uid, gid, size, nonexist, data, icmp, instance, invalid, exec, content, timeout, resource, checksum, fsflags, timestamp, connection, permission, pid, ppid, action*

You can also disable the alerts localy using the NOALERT statement. This is useful if you have lots of services monitored and are using the global alert statement, but don't want to receive alerts for some minor subset of services:

```
noalert appadmin@bar
```

For example, if you stick the noalert statement in a 'check system' entry, you won't receive system related alerts (such as Monit instance started/stopped/reloaded alert, system overloaded alert, etc.) but will receive alerts for all other monitored services.

The following example will alert foo@bar on all events on all services by default, except the service mybar which will send an alert only on timeout. The trick is based on the fact that local definition of the same recipient overrides the global setting (including registered events and mail format):

```
set alert foo@bar

check process myfoo with pidfile /var/run/myfoo.pid
  ...
check process mybar with pidfile /var/run/mybar.pid
  alert foo@bar only on { timeout }
```

Alert message layout

Monit provides a default mail message layout that is short and to the point. Here's an example of a standard alert mail sent by monit:

```
From: monit@tildeslash.com
Subject: Monit alert -- Does not exist apache
To: hauk@tildeslash.com
Date: Thu, 04 Sep 2003 02:33:03 +0200

Does not exist Service apache

       Date:   Thu, 04 Sep 2003 02:33:03 +0200
       Action: restart
       Host:   www.tildeslash.com

Your faithful employee,
monit
```

If you want to, you can change the format of this message with the optional *mail-format* statement. The syntax for this statement is as follows:

```
mail-format {
     from: monit@localhost
  subject: $SERVICE $EVENT at $DATE
  message: Monit $ACTION $SERVICE at $DATE on $HOST: $DESCRIPTION.
           Yours sincerely,
           monit
}
```

Where the keyword *from:* is the email address Monit should pretend it is sending from. It does not have to be a real mail address, but it must be a proper formated mail address, on the form: name@domain. The keyword *subject:* is for the email subject line. The subject must be on only *one* line. The *message:* keyword denotes the mail body. If used, this keyword should always be the last in a mail-format statement. The mail body can be as long as you want, but must **not** contain the '}' character.

All of these format keywords are optional, but if used, you must provide at least one. Thus if you only want to change the from address Monit is using you can do:

```
set alert foo@bar with mail-format { from: bofh@bar.baz }
```

From the previous example you will notice that some special $XXX variables were used. If used, they will be substituted and expanded into the text with these values:

- ***$EVENT***

  ```
  A string describing the event that occurred. The values are
  fixed and are:

  Event:    | Failure state:           | Success state:
  ------------------------------------------------------------------
  ACTION    | "Action done"            | "Action done"
  CHECKSUM  | "Checksum failed"        | "Checksum succeeded"
  CONNECTION| "Connection failed"      | "Connection succeeded"
  CONTENT   | "Content failed",        | "Content succeeded"
  DATA      | "Data access error"      | "Data access succeeded"
  EXEC      | "Execution failed"       | "Execution succeeded"
  FSFLAG    | "Filesystem flags failed"| "Filesystem flags succeeded"
  GID       | "GID failed"             | "GID succeeded"
  ```

```
ICMP      | "ICMP failed"            | "ICMP succeeded"
INSTANCE  | "Monit instance changed" | "Monit instance changed not"
INVALID   | "Invalid type"           | "Type succeeded"
NONEXIST  | "Does not exist"         | "Exists"
PERMISSION| "Permission failed"      | "Permission succeeded"
PID       | "PID failed"             | "PID succeeded"
PPID      | "PPID failed"            | "PPID succeeded"
RESOURCE  | "Resource limit matched" | "Resource limit succeeded"
SIZE      | "Size failed"            | "Size succeeded"
TIMEOUT   | "Timeout"                | "Timeout recovery"
TIMESTAMP | "Timestamp failed"       | "Timestamp succeeded"
UID       | "UID failed"             | "UID succeeded"
```

- ***$SERVICE***

  ```
  The service entry name in monitrc
  ```

- ***$DATE***

  ```
  The current time and date (RFC 822 date style).
  ```

- ***$HOST***

  ```
  The name of the host Monit is running on
  ```

- ***$ACTION***

  ```
  The name of the action which was done. Action names are fixed
  and are:

  Action:  | Name:
  --------------------
  ALERT    | "alert"
  EXEC     | "exec"
  MONITOR  | "monitor"
  RESTART  | "restart"
  START    | "start"
  STOP     | "stop"
  UNMONITOR| "unmonitor"
  ```

- ***$DESCRIPTION***

  ```
  The description of the error condition
  ```

Setting a global mail format

It is possible to set a standard mail format with the following global set-statement (keywords are in capital):

SET MAIL-FORMAT {mail-format}

Format set with this statement will apply to every alert statement that does *not* have its own specified mail-format. This statement is most useful for setting a default from address for messages sent by monit, like so:

```
set mail-format { from: monit@foo.bar.no }
```

Setting an error reminder

Monit by default sends just one error notification if a service failed and another when it recovered. If you want to be notified more then once if a service remains in a failed state, you can use the reminder option to the alert statement (keywords are in capital):

ALERT ... [WITH] REMINDER [ON] number [CYCLES]

For example if you want to be notified each tenth cycle if a service remains in a failed state, you can use:

```
alert foo@bar with reminder on 10 cycles
```

Likewise if you want to be notified on each failed cycle, you can use:

```
alert foo@bar with reminder on 1 cycle
```

Setting a mail server for alert messages

The mail server Monit should use to send alert messages is defined with a global set statement (keywords are in capital and optional statements in [brackets]):

```
SET MAILSERVER {hostname|ip-address [PORT port]
               [USERNAME username] [PASSWORD password]
               [using SSLV2|SSLV3|TLSV1] [CERTMD5 checksum]}+
               [with TIMEOUT X SECONDS]
               [using HOSTNAME hostname]
```

The port statement allows to use SMTP servers other then those listening on port 25. If omitted, port 25 is used unless ssl or tls is used, in which case port 465 is used by default.

Monit support plain smtp authentication - you can set a username and a password using the USERNAME and PASSWORD options.

To use secure communication, use the SSLV2, SSLV3 or TLSV1 options, you can also specify the server certificate checksum using CERTMD5 option.

As you can see, it is possible to set several SMTP servers. If Monit cannot connect to the first server in the list it will try the second server and so on. Monit has a default 5 seconds connection timeout and if the SMTP server is slow, Monit could timeout when connecting or reading from the server. If this is the case, you can use the optional timeout statement to explicit set the timeout to a higher value if needed. Here is an example for setting several mail servers:

```
set mailserver mail.tildeslash.com, mail.foo.bar port 10025
    username "Rabbi" password "Loewe" using tlsv1, localhost
    with timeout 15 seconds
```

Here Monit will first try to connect to the server "mail.tildeslash.com", if this server is down Monit will try "mail.foo.bar" on port 10025 using the given credentials via tls and finally "localhost". We also set an explicit connect and read timeout; If Monit cannot connect to the first SMTP server in the list within 15 seconds it will try the next server and so on. The *set mailserver* .. statement is optional and if not defined Monit will not send email alerts. Not setting a mail server is recommended only if alert notification is delegated to M/Monit.

Monit, by default, use the local host name in SMTP HELO/EHLO and in the Message-ID header. Some mail servers check this information against DNS for spam protection and can reject the email if the DNS and the hostname used in the transaction does not match. If this is the case, you can override the default local host name by using the HOSTNAME option:

```
set mailserver mail.tildeslash.com using hostname
    "myhost.example.org"
```

Event queue

If the MTA (mail server) for sending alerts is not available, Monit *can* queue events on the local file-system until the MTA recover. Monit will then post queued events in order with their original timestamp so the events are not lost. This feature is most useful if Monit is used together with M/Monit and when event history is important.

The event queue is persistent across monit restarts and provided that the back-end filesystem is persistent too, across system restart as well.

By default, the queue is disabled and if the alert handler fails, Monit will simply drop the alert message. To enable the event queue, add the following statement to the Monit control file:

```
SET EVENTQUEUE BASEDIR <path> [SLOTS <number>]
```

The <path> is the path to the directory where events will be stored. Optionally if you want to limit the queue size, use the slots option to only store up to *number* event messages. If the slots option is not used, Monit will store as many events as the backend filesystem allows.

Example:

```
  set eventqueue
```

```
basedir /var/monit
slots 5000
```

Events are stored in a binary format, with one file per event. The file size is ca. 130 bytes or a bit more (depending on the message length). The file name is composed of the unix timestamp, underscore and the service name, for example:

```
/var/monit/1131269471_apache
```

If you are running more then one Monit instance on the same machine, you **must** use separated event queue directories to avoid sending wrong alerts to the wrong addresses.

If you want to purge the queue by hand, that is, remove queued event-files, Monit should be stopped before the removal.

SERVICE TIMEOUT

monit provides a service timeout mechanism for situations where a service simply refuses to start or respond over a longer period.

The timeout mechanism is based on number if service restarts and number of poll-cycles. For example, if a service had *x* restarts within *y* poll-cycles (where *x* <= *y*) then Monit will perform an action (for example unmonitor the service). If a timeout occurs Monit will send an alert message if you have register interest for this event.

The syntax for the timeout statement is as follows (keywords are in capital):

IF <number> RESTART <number> CYCLE(S) THEN <action>

Here is an example where Monit will unmonitor the service if it was restarted 2 times within 3 cycles:

```
if 2 restarts within 3 cycles then unmonitor
```

To have Monit check the service again after a monitoring was disabled, run 'monit monitor <servicename>' from the command line.

Example for setting custom exec on timeout:

```
if 5 restarts within 5 cycles then exec "/foo/bar"
```

Example for stopping the service:

```
if 7 restarts within 10 cycles then stop
```

SERVICE TESTS

Monit provides several tests you may utilize in a service entry to test a service. There are two classes of tests: variable and constant tests. That is, the condition we test is either constant e.g. a number or it can vary.

A constant test has this general format:

IF <TEST> [[<X>] [TIMES WITHIN] <Y> CYCLES] THEN ACTION [ELSE IF SUCCEEDED [[<X>] [TIMES WITHIN] <Y> CYCLES] THEN ACTION]

If the <TEST> condition should evaluate to true, then the selected action is executed each cycle the test condition remains true. The comparison value is constant. Recovery action is evaluated only once (on a failed to succeeded state change only). The 'ELSE IF SUCCEEDED' part is optional, if omitted, Monit will send an alert on recovery. The alert is sent only once on each state change unless overridden by the 'reminder' alert option.

A variable test has this general format:

IF CHANGED <TEST> [[<X>] [TIMES WITHIN] <Y> CYCLES] THEN ACTION

If the <TEST> should evaluate to true, then the selected action is executed once. The comparison value is a variable where the last result becomes the new value and is compared in future cycles. The alert is delivered each time the condition becomes true.

You can use this test for alerts or for some automatic action, for example to reload monitored process after its configuration file was changed. Variable tests are supported for 'checksum', 'size', 'pid, 'ppid' and 'timestamp' tests only.

... [[<X>] [TIMES WITHIN] <Y> CYCLES] ...

If a test match, its action is executed at once. This behaviour can optionally be changed and you can for instance require that a test must match over several poll cycles before the action is executed by using the statement above. You can use this in several ways. For example:

```
 if failed port 80 for 3 times within 5 cycles then alert
```

or

```
 if failed port 80 for 10 cycles then unmonitor
```

If you don't specify <X> times, it equals <Y> by default, thus the test match if it evaluate to true for <Y> consecutive cycles.

It is possible to use this option for failed, succeeded, recovered or changed rules. Here is a more complex example:

```
check filesystem rootfs with path /dev/hda1
 if space usage > 80% for 5 times within 15 cycles
    then alert else if succeeded for 10 cycles then alert
 if space usage > 90% for 5 cycles then
    exec '/try/to/free/the/space'
 if space usage > 99% then exec '/stop/processess'
```

In each test you must select the action to be executed from this list:

- **ALERT** sends the user an alert event on each state change (for constant tests) or on each change (for variable tests).
- **RESTART** restarts the service *and* sends an alert. Restart is conducted by first calling the service's registered stop method and then the service's start method.
- **START** starts the service by calling the service's registered start method *and* send an alert.
- **STOP** stops the service by calling the service's registered stop method *and* send an alert. If Monit stops a service it will not be checked by Monit anymore nor restarted again later. To reactivate monitoring of the service again you must explicitly enable monitoring from the web interface or from the console, e.g. 'monit monitor apache'.
- **EXEC** can be used to execute an arbitrary program *and* send an alert. If you choose this action you must state the program to be executed and if the program require arguments you must enclose the program and its arguments in a quoted string. You may optionally specify the uid and gid the executed program should switch to upon start. For instance:

  ```
   exec "/usr/local/tomcat/bin/startup.sh"
        as uid nobody and gid nobody
  ```

  The uid and gid switch can be useful if the program to be started cannot change to a lesser privileged user and group. This is typically needed for Java Servers. Remember, if Monit is run by the superuser, then all programs executed by Monit will be started with superuser privileges unless the uid and gid extension was used.
- **MONITOR** will enable monitoring of the service *and* send an alert.
- **UNMONITOR** will disable monitoring of the service *and* send an alert. The service will not be checked by Monit anymore nor restarted again later. To reactivate monitoring of the service you must explicitly enable monitoring from monit's web interface or from the console using the monitor argument.

RESOURCE TESTING

Monit can examine how much system resources a service are using. This test can only be used within a system or process service entry in the Monit control file.

Depending on system or process characteristics, services can be stopped or restarted and alerts can be generated. Thus it is possible to utilize systems which are idle and to spare system under high load.

The full syntax for the resource-statements used for resource testing is as follows (keywords are in capital and optional statements in [brackets]),

IF resource operator value [[<X>] <Y> CYCLES] THEN action [ELSE IF SUCCEEDED [[<X>] <Y> CYCLES] THEN action]

resource is a choice of "CPU", "TOTALCPU", "CPU([user|system|wait])", "MEMORY", "CHILDREN", "TOTALMEMORY", "LOADAVG([1min|5min|15min])". Some resource tests can be used inside a check system entry, some in a check process entry and some in both:

System only resource tests:

CPU([user|system|wait]) is the percent of time the system spend in user or system/kernel space. Some systems such as linux 2.6 supports a 'wait' indicator as well.

Process only resource tests:

CPU is the CPU usage of the process itself (percent).

TOTALCPU is the total CPU usage of the process and its children in (percent). You will want to use TOTALCPU typically for services like Apache webserver where one master process forks the child processes as workers.

CHILDREN is the number of child processes of the process.

TOTALMEMORY is the memory usage of the process and its child processes in either percent or as an amount (Byte, kB, MB, GB).

System and process resource tests:

MEMORY is the memory usage of the system or of a process (without children) in either percent (of the systems total) or as an amount (Byte, kB, MB, GB).

LOADAVG([1min|5min|15min]) refers to the system's load average. The load average is the number of processes in the system run queue, averaged over the specified time period.

operator is a choice of "<", ">", "!=", "==" in C notation, "gt", "lt", "eq", "ne" in shell sh notation and "greater", "less", "equal", "notequal" in human readable form (if not specified, default is EQUAL).

value is either an integer or a real number (except for CHILDREN). For CPU, TOTALCPU, MEMORY and TOTALMEMORY you need to specify a *unit*. This could be "%" or if applicable "B" (Byte), "kB" (1024 Byte), "MB" (1024 KiloByte) or "GB" (1024 MegaByte).

action is a choice of "ALERT", "RESTART", "START", "STOP", "EXEC", "MONITOR" or "UNMONITOR".

To calculate the cycles, a counter is raised whenever the expression above is true and it is lowered whenever it is false (but not below 0). All counters are reset in case of a restart.

The following is an example to check that the CPU usage of a service is not going beyond 50% during five poll cycles. If it does, Monit will restart the service:

```
 if cpu is greater than 50% for 5 cycles then restart
```

See also the example section below.

FILE CHECKSUM TESTING

The checksum statement may only be used in a file service entry. If specified in the control file, Monit will compute a md5 or sha1 checksum for a file.

The checksum test in constant form is used to verify that a file does not change. Syntax (keywords are in capital):

IF FAILED [MD5|SHA1] CHECKSUM [EXPECT checksum] [[<X>] <Y> CYCLES] THEN action [ELSE IF SUCCEEDED [[<X>] <Y> CYCLES] THEN action]

The checksum test in variable form is used to watch for file changes. Syntax (keywords are in capital):

IF CHANGED [MD5|SHA1] CHECKSUM [[<X>] <Y> CYCLES] THEN action

The choice of MD5 or SHA1 is optional. MD5 features a 256 bit and SHA1 a 320 bit checksum. If this option is omitted Monit tries to guess the method from the EXPECT string or uses MD5 as default.

expect is optional and if used it specifies a md5 or sha1 string Monit should expect when testing a file's checksum. If *expect* is used, Monit will not compute an initial checksum for the file, but instead use the string you submit. For example:

```
if failed checksum and
   expect the sum 8f7f419955cefa0b33a2ba316cba3659
then alert
```

You can, for example, use the GNU utility *md5sum(1)* or *sha1sum(1)* to create a checksum string for a file and use this string in the expect-statement.

action is a choice of "ALERT", "RESTART", "START", "STOP", "EXEC", "MONITOR" or "UNMONITOR".

The checksum statement in variable form may be used to check a file for changes and if changed, do a specified action. For instance to reload a server if its configuration file was changed. The following illustrates this for the apache web server:

```
check file httpd.conf path /usr/local/apache/conf/httpd.conf
    if changed sha1 checksum
       then exec "/usr/local/apache/bin/apachectl graceful"
```

If you plan to use the checksum statement for security reasons, (a very good idea, by the way) and to monitor a file or files which should not change, then please use the constant form and also read the DEPENDENCY TREE section below to see a detailed example on how to do this properly.

Monit can also test the checksum for files on a remote host via the HTTP protocol. See the CONNECTION TESTING section below.

TIMESTAMP TESTING

The timestamp statement may only be used in a file, fifo or directory service entry.

The timestamp test in constant form is used to verify various timestamp conditions. Syntax (keywords are in capital):

IF TIMESTAMP [[operator] value [unit]] [[<X>] <Y> CYCLES] THEN action [ELSE IF SUCCEEDED [[<X>] <Y> CYCLES] THEN action]

The timestamp statement in variable form is simply to test an existing file or directory for timestamp changes and if changed, execute an action. Syntax (keywords are in capital):

IF CHANGED TIMESTAMP [[<X>] <Y> CYCLES] THEN action

operator is a choice of "<", ">", "!=", "==" in C notation, "GT", "LT", "EQ", "NE" in shell sh notation and "GREATER", "LESS", "EQUAL", "NOTEQUAL" in human readable form (if not specified, default is EQUAL).

value is a time watermark.

unit is either "SECOND", "MINUTE", "HOUR" or "DAY" (it is also possible to use "SECONDS", "MINUTES", "HOURS", or "DAYS").

action is a choice of "ALERT", "RESTART", "START", "STOP", "EXEC", "MONITOR" or "UNMONITOR".

The variable timestamp statement is useful for checking a file for changes and then execute an action. This version was written particularly with configuration files in mind. For instance, if you monitor the apache web server you can use this statement to reload apache if the *httpd.conf* (apache's configuration file) was changed. Like so:

```
check file httpd.conf with path /usr/local/apache/conf/httpd.conf
  if changed timestamp
     then exec "/usr/local/apache/bin/apachectl graceful"
```

The constant timestamp version is useful for monitoring systems able to report its state by changing the timestamp of certain state files. For instance the *iPlanet Messaging server stored process* system updates the timestamp of the following files:

```
o stored.ckp
o stored.lcu
o stored.per
```

If a task should fail, the system keeps the timestamp. To report stored problems you can use the following statements:

```
check file stored.ckp with path /msg-foo/config/stored.ckp
  if timestamp > 1 minute then alert

check file stored.lcu with path /msg-foo/config/stored.lcu
  if timestamp > 5 minutes then alert

check file stored.per with path /msg-foo/config/stored.per
  if timestamp > 1 hour then alert
```

As mentioned above, you can also use the timestamp statement for monitoring directories for changes. If files are added or removed from a directory, its timestamp is changed:

```
check directory mydir path /foo/directory
 if timestamp > 1 hour then alert
```

or

```
check directory myotherdir path /foo/secure/directory
 if timestamp < 1 hour then alert
```

The following example is a hack for restarting a process after a certain time. Sometimes this is a necessary workaround for some third-party applications, until the vendor fix a problem:

```
check file server.pid path /var/run/server.pid
      if timestamp > 7 days
         then exec "/usr/local/server/restart-server"
```

FILE SIZE TESTING

The size statement may only be used in a file service entry. If specified in the control file, Monit will compute a size for a file.

The size test in constant form is used to verify various size conditions. Syntax (keywords are in capital):

IF SIZE [[operator] value [unit]] [[<X>] <Y> CYCLES] THEN action [ELSE IF SUCCEEDED [[<X>] <Y> CYCLES] THEN action]

The size statement in variable form is simply to test an existing file for size changes and if changed, execute an action. Syntax (keywords are in capital):

IF CHANGED SIZE [[<X>] <Y> CYCLES] THEN action

operator is a choice of "<", ">", "!=", "==" in C notation, "GT", "LT", "EQ", "NE" in shell sh notation and "GREATER", "LESS", "EQUAL", "NOTEQUAL" in human readable form (if not specified, default is EQUAL).

value is a size watermark.

unit is a choice of "B","KB","MB","GB" or long alternatives "byte", "kilobyte", "megabyte", "gigabyte". If it is not specified, "byte" unit is assumed by default.

action is a choice of "ALERT", "RESTART", "START", "STOP", "EXEC", "MONITOR" or "UNMONITOR".

The variable size test form is useful for checking a file for changes and send an alert or execute an action. Monit will register the size of the file at startup and monitor the file for changes. As soon as the value changes, Monit will perform the specified action, reset the registered value to the new value and continue monitoring and test if the size changes again.

One example of use for this statement is to conduct security checks, for instance:

```
check file su with path /bin/su
      if changed size then exec "/sbin/ifconfig eth0 down"
```

which will "cut the cable" and stop a possible intruder from compromising the system further. This test is just one of many you may use to increase the security awareness on a system. If you plan to use Monit for security reasons we recommend that you use this test in combination with other supported tests like checksum, timestamp, and so on.

The constant form of this test can be useful in similar or different contexts. It can, for instance, be used to test if a certain file size was exceeded and then alert you or Monit may execute a certain action specified by you. An example is to use this statement to rotate log files after they have reached a certain size or to check that a database file does not grow beyond a specified threshold.

To rotate a log file:

```
check file myapp.log with path /var/log/myapp.log
   if size > 50 MB then
      exec "/usr/local/bin/rotate /var/log/myapp.log myapp"
```

where /usr/local/bin/rotate may be a simple script, such as:

```
#/bin/bash
/bin/mv $1 $1.`date +%y-%m-%d`
/usr/bin/pkill -HUP $2
```

Or you may use this statement to trigger the `logrotate(8)` program, to do an "emergency" rotate. Or to send an alert if a file becomes a known bottleneck if it grows behind a certain size because of limits in a database engine:

```
check file mydb with path /data/mydatabase.db
      if size > 1 GB then alert
```

This is a more restrictive form of the first example where the size is explicitly defined (note that the real su size is system dependent):

```
check file su with path /bin/su
      if size != 95564 then exec "/sbin/ifconfig eth0 down"
```

FILE CONTENT TESTING

The match statement allows you to test the content of a text file by using regular expressions. This is a great feature if you need to periodically test files, such as log files, for certain patterns. If a pattern match, Monit defaults to raise an alert, other actions are also possible.

The syntax (keywords in capital) for using this test is:

IF [NOT] MATCH {regex|path} [[<X>] <Y> CYCLES] THEN action

regex is a string containing the extended regular expression. See also `regex(7)`.

path is an absolute path to a file containing extended regular expression on every line. See also `regex(7)`.

action is a choice of "ALERT", "RESTART", "START", "STOP", "EXEC", "MONITOR" or "UNMONITOR".

You can use the *NOT* statement to invert a match.

The content is only being checked every cycle. If content is being added and removed between two checks they are unnoticed.

On startup the read position is set to the end of the file and Monit continue to scan to the end of file on each cycle. But if the file size should decrease or inode change the read position is set to the start of the file.

Only lines ending with a newline character are inspected. Thus, lines are being ignored until they have been completed with this character. Also note that only the first 511 characters of a line are inspected.

IGNORE [NOT] MATCH {regex|path}

Lines matching an *IGNORE* are not inspected during later evaluations. *IGNORE MATCH* has always precedence over *IF MATCH*.

All *IGNORE MATCH* statements are evaluated first, in the order of their appearance. Thereafter, all the *IF MATCH* statements are evaluated.

A real life example might look like this:

```
check file syslog with path /var/log/syslog
  ignore match
      "^\w{3} [ :0-9]{11} [._[:alnum:]-]+ monit\[[0-9]+\]:"
  ignore match /etc/monit/ignore.regex
  if match
      "^\w{3} [ :0-9]{11} [._[:alnum:]-]+ mrcoffee\[[0-9]+\]:"
  if match /etc/monit/active.regex then alert
```

FILESYSTEM FLAGS TESTING

Monit can test the flags of a filesystem for changes. This test is implicit and Monit will send alert in case of failure by default.

This test is useful for detecting changes of the filesystem flags such as when the filesystem became read-only based on disk errors or the mount flags were changed (such as nosuid). Each platform provides different set of flags. POSIX define the RDONLY and NOSUID flags which should work on all platforms. Some platforms (such as FreeBSD) has additonal flags.

The syntax for the fsflags statement is:

IF CHANGED FSFLAGS [[<X>] <Y> CYCLES] THEN action

action is a choice of "ALERT", "RESTART", "START", "STOP", "EXEC", "MONITOR" or "UNMONITOR".

Example:

```
check filesystem rootfs with path /
      if changed fsflags then exec "/my/script"
      alert root@localhost
```

SPACE TESTING

Monit can test file systems for space usage. This test may only be used within a filesystem service entry in the Monit control file.

Monit will check a filesystem's total space usage. If you only want to check available space for non-superuser, you must set the watermark appropriately (i.e. total space minus reserved blocks for the superuser).

You can obtain (and set) the superuser's reserved blocks size, for example by using the tune2fs utility on Linux. On Linux 5% of available blocks are reserved for the superuser by default. On solaris 10% of the blocks are reserved. You can also use tunefs on solaris to change values on a live filesystem.

The full syntax for the space statement is:

IF SPACE operator value unit [[<X>] <Y> CYCLES] THEN action [ELSE IF SUCCEEDED [[<X>] <Y> CYCLES] THEN action]

operator is a choice of "<",">","!=","==" in c notation, "gt", "lt", "eq", "ne" in shell sh notation and "greater", "less", "equal", "notequal" in human readable form (if not specified, default is EQUAL).

unit is a choice of "B","KB","MB","GB", "%" or long alternatives "byte", "kilobyte", "megabyte", "gigabyte", "percent".

action is a choice of "ALERT", "RESTART", "START", "STOP", "EXEC", "MONITOR" or "UNMONITOR".

INODE TESTING

If supported by the file-system, you can use Monit to test for inodes usage. This test may only be used within a filesystem service entry in the Monit control file.

If the filesystem becomes unavailable, Monit will call the service's registered start method, if it is defined and if Monit is running in active mode. If Monit runs in passive mode or the start methods is not defined, Monit will just send an error alert.

The syntax for the inode statement is:

IF INODE(S) operator value [unit] [[<X>] <Y> CYCLES] THEN action [ELSE IF SUCCEEDED [[<X>] <Y> CYCLES] THEN action]

operator is a choice of "<",">","!=","==" in c notation, "gt", "lt", "eq", "ne" in shell sh notation and "greater", "less", "equal", "notequal" in human readable form (if not specified, default is EQUAL).

unit is optional. If not specified, the value is an absolute count of inodes. You can use the "%" character or the longer alternative "percent" as a unit.

action is a choice of "ALERT", "RESTART", "START", "STOP", "EXEC", "MONITOR" or "UNMONITOR".

PERMISSION TESTING

Monit can monitor the permission of file objects. This test may only be used within a file, fifo, directory or filesystem service entry in the Monit control file.

The syntax for the permission statement is:

IF FAILED PERM(ISSION) octalnumber [[<X>] <Y> CYCLES] THEN action [ELSE IF SUCCEEDED [[<X>] <Y> CYCLES] THEN action]

octalnumber defines permissions for a file, a directory or a filesystem as four octal digits (0-7). Valid range: 0000 - 7777 (you can omit the leading zeros, Monit will add the zeros to the left thus for example "640" is valid value and matches "0640").

action is a choice of "ALERT", "RESTART", "START", "STOP", "EXEC", "MONITOR" or "UNMONITOR".

The web interface will show a permission warning if the test failed.

We recommend that you use the UNMONITOR action in a permission statement. The rationale for this feature is security and that Monit does not start a possible cracked program or script. Example:

```
check file monit.bin with path "/usr/local/bin/monit"
      if failed permission 0555 then unmonitor
```

If the test fails, Monit will simply send an alert and stop monitoring the file and propagate an unmonitor action upward in a depend tree.

UID TESTING

Monit can monitor the owner user id (uid) of a file object. This test may only be used within a file, fifo, directory or filesystem service entry in the Monit control file.

The syntax for the uid statement is:

IF FAILED UID user [[<X>] <Y> CYCLES] THEN action [ELSE IF SUCCEEDED [[<X>] <Y> CYCLES] THEN action]

user defines a user id either in numeric or in string form.

action is a choice of "ALERT", "RESTART", "START", "STOP", "EXEC", "MONITOR" or "UNMONITOR".

The web interface will show a uid warning if the test should fail.

We recommend that you use the UNMONITOR action in a uid statement. The rationale for this feature is security and that Monit does not start a possible cracked program or script. Example:

```
check file passwd with path /etc/passwd
      if failed uid root then unmonitor
```

If the test fails, Monit will simply send an alert and stop monitoring the file and propagate an unmonitor action upward in a depend tree.

GID TESTING

Monit can monitor the owner group id (gid) of file objects. This test may only be used within a file, fifo, directory or filesystem service entry in the Monit control file.

The syntax for the gid statement is:

IF FAILED GID user [[<X>] <Y> CYCLES] THEN action [ELSE IF SUCCEEDED [[<X>] <Y> CYCLES] THEN action]

user defines a group id either in numeric or in string form.

action is a choice of "ALERT", "RESTART", "START", "STOP", "EXEC", "MONITOR" or "UNMONITOR".

The web interface will show a gid warning if the test should fail.

We recommend that you use the UNMONITOR action in a gid statement. The rationale for this feature is security and that Monit does not start a possible cracked program or script. Example:

```
check file shadow with path /etc/shadow
      if failed gid root then unmonitor
```

If the test fails, Monit will simply send an alert and stop monitoring the file and propagate an unmonitor action upward in a depend tree.

PID TESTING

Monit can test the process identification number (pid) of a process for changes. This test is implicit and Monit will send a alert in the case of failure by default.

The syntax for the pid statement is:

IF CHANGED PID [[<X>] <Y> CYCLES] THEN action

action is a choice of "ALERT", "RESTART", "START", "STOP", "EXEC", "MONITOR" or "UNMONITOR".

This test is useful to detect possible process restarts which has occurred in the timeframe between two Monit testing cycles. In the case that the restart was fast and the process provides expected service (i.e. all tests succeeded) you will be notified that the process was replaced.

For example sshd daemon can restart very quickly, thus if someone changes its configuration and do sshd restart outside of Monit's control you will be notified that the process was replaced by a new instance (or you can optionally do some other action such as preventively stop sshd).

Another example is a MySQL Cluster which has its own watchdog with process restart ability. You can use Monit for redundant monitoring.

Example:

```
check process sshd with pidfile /var/run/sshd.pid
      if changed pid then exec "/my/script"
```

PPID TESTING

Monit can test the process parent process identification number (ppid) of a process for changes. This test is implicit and Monit will send alert in the case of failure by default.

The syntax for the ppid statement is:

IF CHANGED PPID [[<X>] <Y> CYCLES] THEN action

action is a choice of "ALERT", "RESTART", "START", "STOP", "EXEC", "MONITOR" or "UNMONITOR".

This test is useful for detecting changes of a process parent.

Example:

```
check process myproc with pidfile /var/run/myproc.pid
      if changed ppid then exec "/my/script"
```

CONNECTION TESTING

Monit is able to perform connection testing via networked ports or via Unix sockets. A connection test may only be used within a process or within a host service entry in the Monit control file.

If a service listens on one or more sockets, Monit can connect to the port (using either tcp or udp) and verify that the service will accept a connection and that it is possible to write and read from the socket. If a connection is not accepted or if there is a problem with socket i/o, Monit will assume that something is wrong and execute a specified action. If Monit is compiled with openssl, then ssl based network services can also be tested.

The full syntax for the statement used for connection testing is as follows (keywords are in capital and optional statements in [brackets]),

IF FAILED [host] port [type] [protocol|{send/expect}+] [timeout] [[<X>] <Y> CYCLES] THEN action [ELSE IF SUCCEEDED [[<X>] <Y> CYCLES] THEN action]

or for Unix sockets,

IF FAILED [unixsocket] [type] [protocol|{send/expect}+] [timeout] [[<X>] <Y> CYCLES] THEN action [ELSE IF SUCCEEDED [[<X>] <Y> CYCLES] THEN action]

host:HOST hostname. Optionally specify the host to connect to. If the host is not given then localhost is assumed if this test is used inside a process entry. If this test was used inside a remote host entry then the entry's remote host is assumed. Although *host* is intended for testing name based virtual host in a HTTP server running on local or remote host, it does allow the connection statement to be used to test a server running on another machine. This may be useful; For instance if you use Apache httpd as a front-end and an application-server as the back-end running on another machine, this statement may be used to test that the back-end server is running and if not raise an alert.

port:PORT number. The port number to connect to

unixsocket:UNIXSOCKET PATH. Specifies the path to a Unix socket. Servers based on Unix sockets, always runs on the local machine and does not use a port.

type:TYPE {TCP|UDP|TCPSSL}. Optionally specify the socket type Monit should use when trying to connect to the port. The different socket types are; TCP, UDP or TCPSSL, where TCP is a regular stream based socket, UDP is a datagram socket and TCPSSL specify that Monit should use a TCP socket with SSL when connecting to a port. The default socket type is TCP. If TCPSSL is used you may optionally specify the SSL/TLS protocol to be used and the md5 sum of the server's certificate. The TCPSSL options are:

```
TCPSSL [SSLAUTO|SSLV2|SSLV3|TLSV1] [CERTMD5 md5sum]
```

proto(col):PROTO {protocols}. Optionally specify the protocol Monit should speak when a connection is established. At the moment Monit knows how to speak: *APACHE-STATUS DNS DWP FTP GPS HTTP IMAP CLAMAV LDAP2 LDAP3 LMTP MYSQL NNTP NTP3 POP POSTFIX-POLICY RADIUS RDATE RSYNC SIP SMTP SSH TNS PGSQL* If you have compiled Monit with ssl support, Monit can also speak the SSL variants such as: *HTTPS FTPS POPS IMAPS* To use the SSL protocol support you need to define the socket as SSL and use the general protocol name (for example in the case of HTTPS) : TYPE TCPSSL PROTOCOL HTTP If the server's protocol is not found in this list, simply do not specify the protocol and Monit will utilize a default test, including test if it is possible to read and write to the port. This default test is in most cases more than good enough to deduce if the server behind the port is up or not.

The protocol statement is:

```
PROTO(COL) {name}
```

HTTP protocol supports additional options: o REQUEST o HOSTHEADER o CHECKSUM

```
PROTO(COL) HTTP [REQUEST {"/path"} [with HOSTHEADER "string"] [with CHECKSUM checksum]
```

When the Host header option is not specified, for HTTP protocol, by default the content of host option which specifies the target host to connect to is used. The Host header can be used when you need to test

Examples:

```
  if failed host 192.168.1.100 port 8080 protocol http and request '/testing' hostheader 'exampl
  if failed host 192.168.1.101 port 8080 protocol http and request '/testing' hostheader 'exampl
  if failed host 192.168.1.102 port 8080 protocol http and request '/testing' hostheader 'exampl
```

In addition to the standard protocols, the *APACHE-STATUS* protocol is a test of a specific server type, rather than a generic protocol. Server performance is examined using the status page generated by Apache's mod_status, which is expected to be at its default address of http://www.example.com/server-status. Currently the *APACHE-STATUS* protocol examines the percentage of Apache child processes which are

```
o logging (loglimit)
o closing connections (closelimit)
o performing DNS lookups (dnslimit)
o in keepalive with a client (keepalivelimit)
o replying to a client (replylimit)
o receiving a request (requestlimit)
o initialising (startlimit)
o waiting for incoming connections (waitlimit)
o gracefully closing down (gracefullimit)
o performing cleanup procedures (cleanuplimit)
```

Each of these quantities can be compared against a value relative to the total number of active Apache child processes. If the comparison expression is true the chosen action is performed.

The apache-status protocol statement is formally defined as (keywords in uppercase):

```
PROTO(COL) {limit} OP PERCENT [OR {limit} OP PERCENT]*
```

where {limit} is one or more of: loglimit, closelimit, dnslimit, keepalivelimit, replylimit, requestlimit, startlimit, waitlimit gracefullimit or cleanuplimit. The operator OP is one of: [<|=|>].

You can combine all of these test into one expression or you can choose to test a certain limit. If you combine the limits you must or' them together using the OR keyword.

Here's an example were we test for a loglimit more than 10 percent, a dnslimit over 25 percent and a wait limit less than 20 percent of processes. See also more examples below in the example section.

```
protocol apache-status
               loglimit > 10% or
               dnslimit > 50% or
               waitlimit < 20%
then alert
```

Obviously, do not use this test unless the httpd server you are testing is Apache Httpd and mod_status is activated on the server.

send/expect: {SEND|EXPECT} "string" .... If Monit does not support the protocol spoken by the server, you can write your own protocol-test using *send* and *expect* strings. The *SEND* statement sends a string to the server port and the *EXPECT* statement compares a string read from the server with the string given in the expect statement. If your system supports POSIX regular expressions, you can use regular expressions in the expect string, see `regex(7)` to learn more about the types of regular expressions you can use in an expect string. Otherwise the string is used as it is. The send/expect statement is:

```
[{SEND|EXPECT} "string"]+
```

Note that Monit will send a string as it is, and you **must** remember to include CR and LF in the string sent to the server if the protocol expect such characters to terminate a string (most text based protocols used over Internet does). Likewise monit will read up to 256 bytes from the server and use this string when comparing the expect string. If the server sends strings terminated by CRLF, (i.e. "\r\n") you *may* remember to add the same terminating characters to the string you expect from the server.

As mentioned above, Monit limits the expect input to 255 bytes. You can override the default value by using this set statement at the top of the Monit configuration file:

```
SET EXPECTBUFFER <number> ["b"|"kb"|"mb"]
```

For example, to set the expect buffer to read 10 kilobytes:

```
set expectbuffer 10 kb
```

Note, if you want to test the number of bytes returned from the server you need to work around a bound check limit in POSIX regex. You cannot use something like expect ".{5000}" as the max number in a boundary check usually is {255}. However this should work, expect "(.{250}){20}"

You can use non-printable characters in a send string if needed. Use the hex notation, \0xHEXHEX to send any char in the range \0x00-\0xFF, that is, 0-255 in decimal. This may be useful when testing some network protocols, particularly those over UDP. For example, to test a quake 3 server you can use the following,

```
send "\0xFF\0xFF\0xFF\0xFFgetstatus"
expect "sv_floodProtect|sv_maxPing"
```

Finally, send/expect can be used with any socket type, such as TCP sockets, UNIX sockets and UDP sockets.

timeout:with TIMEOUT x SECONDS. Optionally specifies the connect and read timeout for the connection. If Monit cannot connect to the server within this time it will assume that the connection failed and execute the specified action. The default connect timeout is 5 seconds.

action is a choice of "ALERT", "RESTART", "START", "STOP", "EXEC", "MONITOR" or "UNMONITOR".

Connection testing using the URL notation

You can test a HTTP server using the compact URL syntax. This test also allow you to use POSIX regular expressions to test the content returned by the HTTP server.

The full syntax for the URL statement is as follows (keywords are in capital and optional statements in [brackets]):

```
IF FAILED URL ULR-spec
   [CONTENT {==|!=} "regular-expression"]
   [TIMEOUT number SECONDS] [[<X>] <Y> CYCLES]
   THEN action
   [ELSE IF SUCCEEDED [[<X>] <Y> CYCLES] THEN action]
```

Where URL-spec is an URL on the standard form as specified in RFC 2396:

```
<protocol>://<authority><path>?<query>
```

Here is an example of an URL where all components are used:

```
http://user:password@www.foo.bar:8080/document/?querystring#ref
```

If a username and password is included in the URL Monit will attempt to login at the server using **Basic Authentication**.

Testing the content returned by the server is optional. If used, you can test if the content **match** or does **not match** a regular expression. Here's an example on how the URL statement can be used in a *check service*:

```
check host FOO with address www.foo.bar
     if failed url
        http://user:password@www.foo.bar:8080/?querystring
        and content == 'action="j_security_check"'
     then ...
```

Monit will look at the content-length header returned by the server and download this amount before testing the content. That is, if the content-length is more than 1Mb or this header is not set by the server Monit will default to download up to 1 Mb and not more.

Only the http(s) protocol is supported in an URL statement. If the protocol is **https** Monit will use SSL when connecting to the server.

Remote host ping test

In addition Monit can perform ICMP Echo tests in remote host checks. The icmp test may only be used in a check host entry and Monit must run with super user privileges, that is, the root user must run monit. The reason is that the icmp test utilize a raw socket to send the icmp packet and only the super user is allowed to create a raw socket.

The full syntax for the ICMP Echo statement used for ping testing is as follows (keywords are in capital and optional statements in [brackets]):

```
IF FAILED ICMP TYPE ECHO
   [COUNT number] [WITH] [TIMEOUT number SECONDS]
     [[<X>] <Y> CYCLES]
   THEN action
   [ELSE IF SUCCEEDED [[<X>] <Y> CYCLES] THEN action]
```

The rules for action and timeout are the same as those mentioned above in the CONNECTION TESTING section. The count parameter specifies how many consecutive echo requests will be send to the host in one cycle. In the case that no reply came within timeout frame, Monit reports error. When at least one reply was received, the test will pass. Monit sends by default three echo requests in one cycle to prevent the random packet loss from generating false alarm (i.e. up to 66% packet loss is tolerated). You can set the count option to a value between 1 and 20, which can serve as an error ratio. For example if you require 100% ping success, set the count to 1 (i.e. just one request will be sent, and if the packet was lost an error will be reported).

An icmp ping test is useful for testing if a host is up, before testing ports at the host. If an icmp ping test is used in a check host entry, this test is run first and if the ping test should fail we assume that the connection to the host is down and Monit does *not* continue to test any ports. Here's an example:

```
check host xyzzy with address xyzzy.org
      if failed icmp type echo count 5 with timeout 15 seconds
         then alert
      if failed port 80 proto http then alert
      if failed port 443 type TCPSSL proto http then alert
      alert foo@bar
```

In this case, if the icmp test should fail you will get *one* alert and only one alert as long as the host is down, and equally important, Monit will *not* test port 80 and port 443. Likewise if the icmp ping test should succeed (again) Monit will continue to test both port 80 and 443.

Keep in mind though that some firewalls can block icmp packages and thus render the test useless.

Examples

To check a port connection and receive an alert if Monit cannot connect to the port, use the following statement:

```
  if failed port 80 then alert
```

In this case the machine in question is assumed to be the default host. For a process entry it's *localhost* and for a remote host entry it's the *address* of the remote host. Monit will conduct a tcp connection to the host at port 80 and use tcp by default. If you want to connect with udp, you can specify this after the port-statement;

```
 if failed port 53 type udp protocol dns then alert
```

Monit will stop trying to connect to the port after 5 seconds and assume that the server behind the port is down. You may increase or decrease the connect timeout by explicit add a connection timeout. In the following example the timeout is increased to 15 seconds and if Monit cannot connect to the server within 15 seconds the test will fail and an alert message is sent.

```
  if failed port 80 with timeout 15 seconds then alert
```

If a server is listening to a Unix socket the following statement can be used:

```
 if failed unixsocket /var/run/sophie then alert
```

A Unix socket is used by some servers for fast (interprocess) communication on localhost only. A Unix socket is specified by a path and in the example above the path, /var/run/sophie, specifies a Unix socket.

If your machine answers for several virtual hosts you can prefix the port statement with a host-statement like so:

```
 if failed host www.sol.no port 80 then alert
 if failed host 80.69.226.133 port 443 then alert
 if failed host kvasir.sol.no port 80 then alert
```

And as mentioned above, if you do not specify a host-statement, *localhost* or *address* is assumed.

Monit also knows how to speak some of the more popular Internet protocols. So, besides testing for connections, Monit can also speak with the server in question to verify that the server works. For example, the following is used to test a http server:

```
 if failed host www.tildeslash.com port 80 proto http
    then restart
```

Some protocols also support a request statement. This statement can be used to ask the server for a special document entity.

Currently **only** the *HTTP* protocol module supports the request statement, such as:

```
 if failed host www.myhost.com port 80 protocol http
    and request "/data/show.php?a=b&c=d"
 then restart
```

The request must contain an URI string specifying a document from the http server. The string will be URL encoded by Monit before it sends the request to the http server, so it's okay to use URL unsafe characters in the request. If the request statement isn't specified, the default web server page will be requested.

You can override default Host header in HTTP request:

```
 if failed host 192.168.1.100 port 80 protocol http
    hostheader "example.com"
 then alert
```

You can also test the checksum for documents returned by a http server. You can use either MD5 sums:

```
 if failed port 80 protocol http
    and request "/page.html"
        with checksum 8f7f419955cefa0b33a2ba316cba3659
 then alert
```

Or you can use SHA1 sums:

```
 if failed port 80 protocol http
```

```
    and request "/page.html"
        with checksum e428302e260e0832007d82de853aa8edf19cd872
 then alert
```

Monit will compute a checksum (either MD5 or SHA1 is used, depending on length of the hash) for the document (in the above case, /page.html) and compare the computed checksum with the expected checksum. If the sums does not match then the if-tests action is performed, in this case alert. Note that Monit will **not** test the checksum for a document if the server does not set the HTTP *Content-Length* header. A HTTP server should set this header when it server a static document (i.e. a file). A server will often use chunked transfer encoding instead when serving dynamic content (e.g. a document created by a CGI-script or a Servlet), but to test the checksum for dynamic content is not very useful. There are no limitation on the document size, but keep in mind that Monit will use time to download the document over the network so it's probably smart not to ask monit to compute a checksum for documents larger than 1Mb or so, depending on you network connection of course. Tip; If you get a checksum error even if the document has the correct sum, the reason may be that the download timed out. In this case, explicit set a longer timeout than the default 5 seconds.

As mentioned above, if the server protocol is not supported by Monit you can write your own protocol test using send/expect strings. Here we show a protocol test using send/expect for an imaginary "Ali Baba and the Forty Thieves" protocol:

```
 if failed host cave.persia.ir port 4040
    send "Open, Sesame!\r\n"
    expect "Please enter the cave\r\n"
    send "Shut, Sesame!\r\n"
    expect "See you later [A-Za-z ]+\r\n"
 then restart
```

The *TCPSSL* statement can optionally test the md5 sum of the server's certificate. You must state the md5 certificate string you expect the server to deliver and upon a connect to the server, the server's actual md5 sum certificate string is tested. Any other symbol but [A-Fa-f0-9] is being ignored in that sting. Thus it is possible to copy and paste the output of e.g. openssl. If they do not match, the connection test fails. If the ssl version handshake does not work properly you can also force a specific ssl version, as we demonstrate in this example:

```
 if failed host shop.sol.no port 443
    type TCPSSL SSLV3 # Force Monit to use ssl version 3
    # We expect the server to return this  md5 certificate sum
    # as either 12-34-56-78-90-AB-CD-EF-12-34-56-78-90-AB-CD-EF
    # or e.g.   1234567890ABCDEF1234567890ABCDEF
    # or e.g.   1234567890abcdef1234567890abcdef
    # what ever come in more handy (see text above)
    CERTMD5 12-34-56-78-90-AB-CD-EF-12-34-56-78-90-AB-CD-EF
    protocol http
 then restart
```

Here's an example where a connection test is used inside a process entry:

```
 check process apache with pidfile /var/run/apache.pid
       start program = "/etc/init.d/httpd start"
       stop program = "/etc/init.d/httpd stop"
       if failed host www.tildeslash.com port 80 then restart
```

Here, a connection test is used in a remote host entry:

```
 check host up2date with address ftp.redhat.com
       if failed port 21 and protocol ftp then alert
```

Since we did not explicit specify a host in the above test, monit will connect to port 21 at ftp.redhat.com. Apropos, the host address can be specified as a dotted IP address string or as hostname in the DNS. The following is exactly[*] the same test, but here an ip address is used instead:

```
 check host up2date with address 66.187.232.30
       if failed port 21 and protocol ftp then alert
```

[*] Well, not quite, since we specify an ip-address directly we will bypass any DNS round-robin setup, but that's another story.

Testing the SIP protocol

The SIP protocol is used by communication platform servers such as Asterisk and FreeSWITCH.

The SIP test is similar to the other protocol tests, but in addition allows extra optional parameters.

IF FAILED [host] [port] [type] PROTOCOL sip [AND] [TARGET valid@uri] [AND] [MAXFORWARD n] THEN action [ELSE IF SUCCEEDED [[<X>] <Y> CYCLES] THEN action]

TARGET : you may specify an alternative recipient for the message, by adding a valid sip uri after this keyword.

MAXFORWARD : Limit the number of proxies or gateways that can forward the request to the next server. It's value is an integer in the range 0-255, set by default to 70. If max-forward = 0, the next server may respond 200 OK (test succeeded) or send a 483 Too Many Hops (test failed)

SIP examples:

```
 check host openser_all with address 127.0.0.1
   if failed port 5060 type udp protocol sip
      with target "localhost:5060" and maxforward 6
   then alert

If sips is supported, that is, sip over ssl, specify tcpssl as
the connection type.

 check host fwd.pulver.com with address fwd.pulver.com
   if failed port 5060 type tcpssl protocol SIP
      and target 613@fwd.pulver.com maxforward 10
   then alert
```

For more examples, see the example section below.

Testing the RADIUS protocol

The SIP test is similar to the other protocol tests, but in addition allows extra optional parameters.

IF FAILED [host] [port] [type] PROTOCOL radius [SECRET string] THEN action [ELSE IF SUCCEEDED [[<X>] <Y> CYCLES] THEN action]

SECRET: you may specify an alternative secret, default is "testing123".

RADIUS example:

```
 check process radiusd with pidfile /var/run/radiusd.pid
       start program = "/etc/init.d/freeradius start"
       stop program = "/etc/init.d/freeradius stop"
       if failed host 127.0.0.1 port 1812 type udp protocol radius secret testing123 then alert
       if 5 restarts within 5 cycles then timeout
```

MONIT HTTPD

If specified in the control file, Monit will start a Monit daemon with http support. From a Browser you can then start and stop services, disable or enable service monitoring as well as view the status of each service. Also, if Monit logs to its own file, you can view the content of this logfile in a Browser.

The control file statement for starting a Monit daemon with http support is a global set-statement:

set httpd port 2812

And you can use this URL, *http://localhost:2812/*, to access the daemon from a browser. The port number, in this case 2812, can be any number that you are allowed to bind to.

If you have compiled Monit with openssl, you can also start the httpd server with ssl support, using the following expression:

```
 set httpd port 2812
```

```
    ssl enable
    pemfile /etc/certs/monit.pem
```

And you can use this URL, *https://localhost:2812/*, to access the Monit web server over an ssl encrypted connection.

The pemfile, in the example above, holds both the server's private key and certificate. This file should be stored in a safe place on the filesystem and should have strict permissions, that is, no more than 0700.

In addition, if you want to check for client certificates you can use the CLIENTPEMFILE statement. In this case, a connecting client has to provided a certificate known by Monit in order to connect. This file also needs to have all necessary CA certificates. A configuration could look like:

```
set httpd port 2812
    ssl enable
    pemfile /etc/certs/monit.pem
    clientpemfile /etc/certs/monit-client.pem
```

By default self signed client certificates are not allowed. If you want to use a self signed certificate from a client it has to be allowed explicitly with the ALLOWSELFCERTIFICATION statement.

For more information on how to use Monit with SSL and for more information about certificates and generating pem files, please consult the README.SSL file accompanying the software.

If you only want the http server to accept connect requests to one host addresses you can specify the bind address either as an IP number string or as a hostname. In the following example we bind the http server to the loopback device. In other words the http server will only be reachable from localhost:

```
  set httpd port 2812 and use the address 127.0.0.1
```

or

```
  set httpd port 2812 and use the address localhost
```

If you do not use the ADDRESS statement the http server will accept connections on any/all local addresses.

It is possible to hide monit's httpd server version, which usually is available in httpd header responses and in error pages.

```
  set httpd port 2812
    ...
    signature {enable|disable}
```

Use *disable* to hide the server signature - Monit will only report its name (e.g. 'monit' instead of for example 'monit 4.2'). By default the version signature is enabled. It is worth to stress that this option provides no security advantage and falls into the "security through obscurity" category.

If you remove the httpd statement from the config file, monit will stop the httpd server on configuration reload. Likewise if you change the port number, Monit will restart the http server using the new specified port number.

The status page displayed by the Monit web server is automatically refreshed with the same poll time set for the monit daemon.

Note:

We strongly recommend that you start Monit with http support (and bind the server to localhost, only, unless you are behind a firewall). The built-in web-server is small and does not use much resources, and more *importantly*, Monit can use the http server for interprocess communication between a Monit client and a monit daemon.

For instance, you *must* start a Monit daemon with http support if you want to be able to use most of the available console commands. I.e. 'Monit stop all', 'Monit start all' etc.

If a Monit daemon is running in the background we will ask the daemon (via the HTTP protocol) to execute the above commands. That is, the daemon is requested to start and stop the services. This ensures that a daemon will not restart a service that you requested to stop and that (any) timeout lock will be removed from a service when you start it.

Monit HTTPD Authentication

Monit supports two types of authentication schema's for connecting to the httpd server, (three, if you count SSL client certificate validation). Both schema's can be used together or by itself. You **must** choose at least one.

Host and network allow list

The http server maintains an access-control list of hosts and networks allowed to connect to the server. You can add as many hosts as you want to, but only hosts with a valid domain name or its IP address are allowed. Networks require a network IP and a netmask to be accepted.

The http server will query a name server to check any hosts connecting to the server. If a host (client) is trying to connect to the server, but cannot be found in the access list or cannot be resolved, the server will shutdown the connection to the client promptly.

Control file example:

```
set httpd port 2812
    allow localhost
    allow my.other.work.machine.com
    allow 10.1.1.1
    allow 192.168.1.0/255.255.255.0
    allow 10.0.0.0/8
```

Clients, not mentioned in the allow list, trying to connect to the server are logged with their ip-address.

Basic Authentication

This authentication schema is HTTP specific and described in more detail in RFC 2617.

In short; a server challenge a client (e.g. a Browser) to send authentication information (username and password) and if accepted, the server will allow the client access to the requested document.

The biggest weakness with Basic Authentication is that the username and password is sent in clear-text (i.e. base64 encoded) over the network. It is therefor recommended that you do not use this authentication method unless you run the Monit http server with *ssl* support. With ssl support it is completely safe to use Basic Authentication since **all** http data, including Basic Authentication headers will be encrypted.

Monit will use Basic Authentication if an allow statement contains a username and a password separated with a single ':' character, like so: *allow username:password*. The username and password must be written in clear-text. Special characters can be used but the password has to be quoted.

PAM is supported as well on platforms which provide PAM (such as Linux, Mac OS X, FreeBSD, NetBSD). The syntax is: *allow @mygroup* which provides access to the user of group called *mygroup*. Monit uses PAM service called *monit* for PAM authentication, see PAM manual page for detailed instructions how to set the PAM service and PAM authentication plugins. Example Monit PAM for Mac OS X - /etc/pam.d/monit:

```
# monit: auth account password session
auth       sufficient     pam_securityserver.so
auth       sufficient     pam_unix.so
auth       required       pam_deny.so
account    required       pam_permit.so
```

And configuration part for monitrc which allows only group admins authenticated using via PAM to access the http interface:

```
set httpd port 2812 allow @admin
```

Alternatively you can use files in "htpasswd" format (one user:passwd entry per line), like so: *allow [cleartext|crypt|md5] /path [users]*. By default cleartext passwords are read. In case the passwords are digested it is necessary to specify the cryptographic method. If you do not want all users in the password file to have access to Monit you can specify only those users that should have access, in the allow statement. Otherwise all users are added.

Example1:

```
set httpd port 2812
    allow hauk:password
    allow md5 /etc/httpd/htpasswd john paul ringo george
```

If you use this method together with a host list, then only clients from the listed hosts will be allowed to connect to the Monit http server and each client will be asked to provide a username and a password.

Example2:

```
set httpd port 2812
    allow localhost
    allow 10.1.1.1
    allow hauk:"password"
```

If you only want to use Basic Authentication, then just provide allow entries with username and password or password files as in example 1 above.

Finally it is possible to define some users as read-only. A read-only user can read the Monit web pages but will *not* get access to push-buttons and cannot change a service from the web interface.

```
set httpd port 2812
    allow admin:password
    allow hauk:password read-only
    allow @admins
    allow @users read-only
```

A user is set to read-only by using the *read-only* keyword **after** username:password. In the above example the user *hauk* is defined as a read-only user, while the *admin* user has all access rights.

If you use Basic Authentication it is a good idea to set the access permission for the control file (~/.monitrc) to only readable and writable for the user running monit, because the password is written in clear-text. (Use this command, /bin/chmod 600 ~/.monitrc). In fact, since Monit **version 3.0**, Monit will complain and exit if the control file is readable by others.

Clients trying to connect to the server but supply the wrong username and/or password are logged with their ip-address.

If the Monit command line interface is being used, at least one cleartext password is necessary. Otherwise, the Monit command line interface will not be able to connect to the Monit daemon server.

DEPENDENCIES

If specified in the control file, Monit can do dependency checking before start, stop, monitoring or unmonitoring of services. The dependency statement may be used within any service entries in the Monit control file.

The syntax for the depend statement is simply:

DEPENDS on service[, service [,...]]

Where **service** is a service entry name, for instance **apache** or **datafs**.

You may add more than one service name of any type or use more than one depend statement in an entry.

Services specified in a *depend* statement will be checked during stop/start/monitor/unmonitor operations. If a service is stopped or unmonitored it will stop/unmonitor any services that depends on itself. Likewise, if a service is started, it will first stop any services that depends on itself and after it is started, start all depending services again. If the service is to be monitored (enable monitoring), all services which this service depends on will be monitored before enabling monitoring of this service.

Here is an example where we set up an apache service entry to depend on the underlying apache binary. If the binary should change an alert is sent and apache is not monitored anymore. The rationale is security and that Monit should not execute a possibly cracked apache binary.

```
(1) check process apache
(2)    with pidfile "/usr/local/apache/logs/httpd.pid"
(3)    ...
(4)    depends on httpd
(5)
(6) check file httpd with path /usr/local/apache/bin/httpd
(7)    if failed checksum then unmonitor
```

The first entry is the process entry for apache shown before (abbreviated for clarity). The fourth line sets up a dependency between this entry and the service entry named httpd in line 6. A depend tree works as follows, if an action is conducted in a lower branch it will propagate upward in the tree and for every dependent entry execute the same action. In this case, if the checksum should fail in line 7 then an unmonitor action is executed and the apache binary is not checked anymore. But since the apache process entry depends on the httpd entry this entry will also execute the unmonitor action. In short, if the checksum test for the httpd binary file should fail, both the check file httpd entry and the check process apache entry is set in un-monitoring mode.

A dependency tree is a general construct and can be used between all types of service entries and span many levels and propagate any supported action (except the exec action which will not propagate upward in a dependency tree for obvious reasons).

Here is another different example. Consider the following common server setup:

```
WEB-SERVER -> APPLICATION-SERVER -> DATABASE -> FILESYSTEM
    (a)              (b)             (c)          (d)
```

You can set dependencies so that the web-server depends on the application server to run before the web-server starts and the application server depends on the database server and the database depends on the file-system to be mounted before it starts. See also the example section below for examples using the depend statement.

Here we describe how Monit will function with the above dependencies:

If no servers are running

Monit will start the servers in the following order: *d*, *c*, *b*, *a*

If all servers are running

When you run 'Monit stop all' this is the stop order: *a*, *b*, *c*, *d*. If you run 'Monit stop d' then *a*, *b* and *c* are also stopped because they depend on *d* and finally *d* is stopped.

If *a* does not run

When Monit runs it will start *a*

If *b* does not run

When Monit runs it will first stop *a* then start *b* and finally start *a* again.

If *c* does not run

When Monit runs it will first stop *a* and *b* then start *c* and finally start *b* then *a*.

If *d* does not run

When Monit runs it will first stop *a*, *b* and *c* then start *d* and finally start *c*, *b* then *a*.

If the control file contains a depend loop.

A depend loop is for example; a->b and b->a or a->b->c->a.

When Monit starts it will check for such loops and complain and exit if a loop was found. It will also exit with a complaint if a depend statement was used that does not point to a service in the control file.

THE RUN CONTROL FILE

The preferred way to set up Monit is to write a *.monitrc* file in your home directory. When there is a conflict between the command-line arguments and the arguments in this file, the command-line arguments take precedence. To protect the security of your control file and passwords the control file must have permissions *no more than 0700* (u=xrw,g=,o=); Monit will complain and exit otherwise.

Run Control Syntax

Comments begin with a '#' and extend through the end of the line. Otherwise the file consists of a series of service entries or global option statements in a free-format, token-oriented syntax.

There are three kinds of tokens: grammar keywords, numbers (i.e. decimal digit sequences) and strings. Strings can be either quoted or unquoted. A quoted string is bounded by double quotes and may contain whitespace (and quoted digits are treated as a string). An unquoted string is any whitespace-delimited token, containing characters and/or numbers.

On a semantic level, the control file consists of two types of entries:

1. **Global set-statements**

   A global set-statement starts with the keyword *set* and the item to configure.

2. **One or more service entry statements.**

   Each service entry consists of the keywords `check', followed by the service type. Each entry requires a <unique> descriptive name, which may be freely chosen. This name is used by monit to refer to the service internally and in all interactions with the user.

Currently, six types of check statements are supported:

1. **CHECK PROCESS <unique name> PIDFILE <path>**

   <path> is the absolute path to the program's pidfile. If the pidfile does not exist or does not contain the pid number of a running process, Monit will call the entry's start method if defined, If Monit runs in passive mode or the start methods is not defined, Monit will just send alerts on errors.

2. **CHECK FILE <unique name> PATH <path>**

   <path> is the absolute path to the file. If the file does not exist or disappeared, Monit will call the entry's start method if defined, if <path> does not point to a regular file type (for instance a directory), Monit will disable monitoring of this entry. If Monit runs in passive mode or the start methods is not defined, Monit will just send alerts on errors.

3. **CHECK FIFO <unique name> PATH <path>**

   <path> is the absolute path to the fifo. If the fifo does not exist or disappeared, Monit will call the entry's start method if defined, if <path> does not point to a fifo type (for instance a directory), Monit will disable monitoring of this entry. If Monit runs in passive mode or the start methods is not defined, Monit will just send alerts on errors.

4. **CHECK FILESYSTEM <unique name> PATH <path>**

   <path> is the path to the filesystem block special device, mount point, file or a directory which is part of a filesystem. It is recommended to use a block special file directly (for example /dev/hda1 on Linux or /dev/dsk /c0t0d0s1 on Solaris, etc.) If you use a mount point (for example /data), be careful, because if the filesystem is unmounted the test will still be true because the mount point exist.

   If the filesystem becomes unavailable, Monit will call the entry's start method if defined. if <path> does not point to a filesystem, Monit will disable monitoring of this entry. If Monit runs in passive mode or the start methods is not defined, Monit will just send alerts on errors.

5. **CHECK DIRECTORY <unique name> PATH <path>**

   <path> is the absolute path to the directory. If the directory does not exist or disappeared, Monit will call the entry's start method if defined, if <path> does not point to a directory, monit will disable monitoring of this entry. If Monit runs in passive mode or the start methods is not defined, Monit will just send alerts on errors.

6. **CHECK HOST <unique name> ADDRESS <host address>**

   The host address can be specified as a hostname string or as an ip-address string on a dotted decimal format. Such as, tildeslash.com or "64.87.72.95".

7. **CHECK SYSTEM <unique name>**

   The system name is usualy hostname, but any descriptive name can be used. This test allows to check general system resources such as CPU usage (percent of time spent in user, system and wait), total memory usage or load average.

You can use noise keywords like 'if', `and', `with(in)', `has', `using', 'use', 'on(ly)', `usage' and `program(s)' anywhere in an entry to make it resemble English. They're ignored, but can make entries much easier to read at a glance. The punctuation characters ';' ',' and '=' are also ignored. Keywords are case insensitive.

```
Here are the legal global keywords:

Keyword         Function
-----------------------------------------------------------------
set daemon      Set a background poll interval in seconds.
set init        Set Monit to run from init. Monit will not
                transform itself into a daemon process.
set logfile     Name of a file to dump error- and status-
                messages to. If syslog is specified as the
                file, Monit will utilize the syslog daemon
                to log messages. This can optionally be
                followed by 'facility <facility>' where
                facility is 'log_local0' - 'log_local7' or
                'log_daemon'. If no facility is specified,
                LOG_USER is used.
set mailserver  The mailserver used for sending alert
                notifications. If the mailserver is not
                defined, Monit will try to use 'localhost'
                as the smtp-server for sending mail. You
                can add more mail servers, if Monit cannot
                connect to the first server it will try the
                next server and so on.
set mail-format Set a global mail format for all alert
                messages emitted by monit.
set idfile      Explicit set the location of the Monit id
                file. E.g. set idfile /var/monit/id.
set pidfile     Explicit set the location of the Monit lock
                file. E.g. set pidfile /var/run/xyzmonit.pid.
set statefile   Explicit set the location of the file Monit
                will write state data to. If not set, the
                default is $HOME/.monit.state.
set httpd port  Activates Monit http server at the given
                port number.
ssl enable      Enables ssl support for the httpd server.
                Requires the use of the pemfile statement.
ssl disable     Disables ssl support for the httpd server.
                It is equal to omitting any ssl statement.
pemfile         Set the pemfile to be used with ssl.
clientpemfile   Set the pemfile to be used when client
                certificates should be checked by monit.
address         If specified, the http server will only
                accept connect requests to this addresses
                This statement is an optional part of the
                set httpd statement.
allow           Specifies a host or IP address allowed to
                connect to the http server. Can also specify
                a username and password allowed to connect
                to the server. More than one allow statement
                are allowed. This statement is also an
                optional part of the set httpd statement.
read-only       Set the user defined in username:password
                to read only. A read-only user cannot change
```

```
                a service from the Monit web interface.
include         include a file or files matching the globstring

Here are the legal service entry keywords:

Keyword         Function
------------------------------------------------------------
check           Starts an entry and must be followed by the type
                of monitored service {filesystem|directory|file|host
                process|system} and a descriptive name for the
                service.
pidfile         Specify the  process pidfile. Every
                process must create a pidfile with its
                current process id. This statement should only
                be used in a process service entry.
path            Must be followed by a path to the block
                special file for filesystem, regular
                file, directory or a process's pidfile.
group           Specify a groupname for a service entry.
start           The program used to start the specified
                service. Full path is required. This
                statement is optional, but recommended.
stop            The program used to stop the specified
                service. Full path is required. This
                statement is optional, but recommended.
pid and ppid    These keywords may be used as standalone
                statements in a process service entry to
                override the alert action for change of
                process pid and ppid.
uid and gid     These keywords are either 1) an optional part of
                a start, stop or exec statement. They may be
                used to specify a user id and a group id the
                program (process) should switch to upon start.
                This feature can only be used if the superuser
                is running monit. 2) uid and gid may also be
                used as standalone statements in a file service
                entry to test a file's uid and gid attributes.
host            The hostname or IP address to test the port
                at. This keyword can only be used together
                with a port statement or in the check host
                statement.
port            Specify a TCP/IP service port number which
                a process is listening on. This statement
                is also optional. If this statement is not
                prefixed with a host-statement, localhost is
                used as the hostname to test the port at.
type            Specifies the socket type Monit should use when
                testing a connection to a port. If the type
                keyword is omitted, tcp is used. This keyword
                must be followed by either tcp, udp or tcpssl.
tcp             Specifies that Monit should use a TCP
                socket type (stream) when testing a port.
tcpssl          Specifies that Monit should use a TCP socket
                type (stream) and the secure socket layer (ssl)
                when testing a port connection.
udp             Specifies that Monit should use a UDP socket
                type (datagram) when testing a port.
certmd5         The md5 sum of a certificate a ssl forged
                server has to deliver.
proto(col)      This keyword specifies the type of service
                found at the port. See CONNECTION TESTING
                for list of supported protocols.
                You're welcome to write new protocol test
                modules. If no protocol is specified Monit will
                use a default test which in most cases are good
                enough.
request         Specifies a server request and must come
                after the protocol keyword mentioned above.
                 - for http it can contain an URL and an
                   optional query string.
                 - other protocols does not support this
                   statement yet
send/expect     These keywords specify a generic protocol.
                Both require a string whether to be sent or
                to be matched against (as extended regex if
                supported).  Send/expect can not be used
                together with the proto(col) statement.
```

unix(socket)	Specifies a Unix socket file and used like the port statement above to test a Unix domain network socket connection.
URL	Specify an URL string which Monit will use for connection testing.
content	Optional sub-statement for the URL statement. Specifies that Monit should test the content returned by the server against a regular expression.
timeout x sec.	Define a network port connection timeout. Must be followed by a number in seconds and the keyword, seconds.
timeout	Define a service timeout. Must be followed by two digits. The first digit is max number of restarts for the service. The second digit is the cycle interval to test restarts. This statement is optional.
alert	Specifies an email address for notification if a service event occurs. Alert can also be postfixed, to only send a message for certain events. See the examples above. More than one alert statement is allowed in an entry. This statement is also optional.
noalert	Specifies an email address which don't want to receive alerts. This statement is also optional.
restart, stop unmonitor, start and exec	These keywords may be used as actions for various test statements. The exec statement is special in that it requires a following string specifying the program to be execute. You may also specify an UID and GID for the exec statement. The program executed will then run using the specified user id and group id.
mail-format	Specifies a mail format for an alert message This statement is an optional part of the alert statement.
checksum	Specify that Monit should compute and monitor a file's md5/sha1 checksum. May only be used in a check file entry.
expect	Specifies a md5/sha1 checksum string Monit should expect when testing the checksum. This statement is an optional part of the checksum statement.
timestamp	Specifies an expected timestamp for a file or directory. More than one timestamp statement are allowed. May only be used in a check file or check directory entry.
changed	Part of a timestamp statement and used as an operator to simply test for a timestamp change.
every	Validate this entry only at every n poll cycle. Useful in daemon mode when the cycle is short and a service takes some time to start.
mode	Must be followed either by the keyword active, passive or manual. If active, Monit will restart the service if it is not running (this is the default behavior). If passive, Monit will not (re)start the service if it is not running - it will only monitor and send alerts (resource related restart and stop options are ignored in this mode also). If manual, Monit will enter active mode only if a service was started under monit's control otherwise the service isn't monitored.
cpu	Must be followed by a compare operator, a number with "%" and an action. This statement is used to check the cpu usage in percent of a process with its children over a number of cycles. If the compare expression matches then the specified action is executed.
mem	The equivalent to the cpu token for memory of a process (w/o children!). This token must be followed by a compare operator a number with unit {B|KB|MB|GB|%|byte|kilobyte|megabyte| gigabyte|percent} and an action.
loadavg	Must be followed by [1min,5min,15min] in (), a compare operator, a number and an action. This statement is used to check the system load

```
                average over a number of cycles. If the compare
                expression matches then the specified action is
                executed.
 children       This is the number of child processes spawn by a
                process. The syntax is the same as above.
 totalmem       The equivalent of mem, except totalmem is an
                aggregation of memory, not only used by a
                process but also by all its child
                processes. The syntax is the same as above.
 space          Must be followed by a compare operator, a
                number, unit {B|KB|MB|GB|%|byte|kilobyte|
                megabyte|gigabyte|percent} and an action.
 inode(s)       Must be followed by a compare operator, integer
                number, optionally by percent sign (if not, the
                limit is absolute) and an action.
 perm(ission)   Must be followed by an octal number describing
                the permissions.
 size           Must be followed by a compare operator, a
                number, unit {B|KB|MB|GB|byte|kilobyte|
                megabyte|gigabyte} and an action.
 depends (on)   Must be followed by the name of a service this
                service depends on.
```

Here's the complete list of reserved **keywords** used by monit:

if, *then*, *else*, *set*, *daemon*, *logfile*, *syslog*, *address*, *httpd*, *ssl*, *enable*, *disable*, *pemfile*, *allow*, *read-only*, *check*, *init*, *count*, *pidfile*, *statefile*, *group*, *start*, *stop*, *uid*, *gid*, *connection*, *port(number)*, *unix(socket)*, *type*, *proto(col)*, *tcp*, *tcpssl*, *udp*, *alert*, *noalert*, *mail-format*, *restart*, *timeout*, *checksum*, *resource*, *expect*, *send*, *mailserver*, *every*, *mode*, *active*, *passive*, *manual*, *depends*, *host*, *default*, *http*, *ftp*, *smtp*, *pop*, *ntp3*, *nntp*, *imap*, *clamav*, *ssh*, *dwp*, *ldap2*, *ldap3*, *tns*, *request*, *cpu*, *mem*, *totalmem*, *children*, *loadavg*, *timestamp, changed, second(s), minute(s), hour(s), day(s)*, *space*, *inode*, *pid*, *ppid*, *perm(ission)*, *icmp*, *process*, *file*, *directory*, *filesystem*, *size*, *action*, *unmonitor*, *rdate*, *rsync*, *data*, *invalid*, *exec*, *nonexist*, *policy*, *reminder*, *instance*, *eventqueue*, *basedir*, *slot(s)*, *system*, *idfile*, *gps*, *radius*, *secret*, *target*, *maxforward*, *hostheader* and *failed*

And here is a complete list of **noise keywords** ignored by monit:

is, *as*, *are*, *on(ly)*, *with(in)*, *and*, *has*, *using*, *use*, *the*, *sum*, *program(s)*, *than*, *for*, *usage*, *was*, *but*, *of*.

Note: If the *start* or *stop* programs are shell scripts, then the script must begin with `#!` and the remainder of the first line must specify an interpreter for the program. E.g. `#!/bin/sh`

It's possible to write scripts directly into the *start* and *stop* entries by using a string of shell-commands. Like so:

```
start="/bin/bash -c 'echo $$ > pidfile; exec program'"
stop="/bin/bash -c 'kill -s SIGTERM `cat pidfile`'"
```

CONFIGURATION EXAMPLES

The simplest form is just the check statement. In this example we check to see if the server is running and log a message if not:

```
check process resin with pidfile /usr/local/resin/srun.pid
```

To have Monit start the server if it's not running, add a start statement:

```
check process resin with pidfile /usr/local/resin/srun.pid
      start program = "/usr/local/resin/bin/srun.sh start"
```

Here's a more advanced example for monitoring an apache web-server listening on the default port number for HTTP and HTTPS. In this example Monit will restart apache if it's not accepting connections at the port numbers. The method Monit use for a process restart is to first execute the stop-program, wait up to 30s for the process to stop and then execute the start-program and wait up to 30s for it to start. The length of start or stop timeout can be overriden using the 'timeout' option. If Monit was unable to stop or start the service a failed alert message will be sent if you have requested alert messages to be sent.

```
check process apache with pidfile /var/run/httpd.pid
      start program = "/etc/init.d/httpd start" with timeout 60 seconds
      stop program  = "/etc/init.d/httpd stop"
```

```
      if failed port 80 then restart
      if failed port 443 with timeout 15 seconds then restart
```

This example demonstrate how you can run a program as a specified user (uid) and with a specified group (gid). Many daemon programs will do the uid and gid switch by them self, but for those programs that does not (e.g. Java programs), monit's ability to start a program as a certain user can be very useful. In this example we start the Tomcat Java Servlet Engine as the standard *nobody* user and group. Please note that Monit will only switch uid and gid for a program if the super-user is running monit, otherwise Monit will simply ignore the request to change uid and gid.

```
check process tomcat with pidfile /var/run/tomcat.pid
      start program = "/etc/init.d/tomcat start"
            as uid nobody and gid nobody
      stop program  = "/etc/init.d/tomcat stop"
            # You can also use id numbers instead and write:
            as uid 99 and with gid 99
      if failed port 8080 then alert
```

In this example we use udp for connection testing to check if the name-server is running and also use timeout and alert:

```
check process named with pidfile /var/run/named.pid
      start program = "/etc/init.d/named start"
      stop program  = "/etc/init.d/named stop"
      if failed port 53 use type udp protocol dns then restart
      if 3 restarts within 5 cycles then timeout
```

The following example illustrates how to check if the service 'sophie' is answering connections on its Unix domain socket:

```
check process sophie with pidfile /var/run/sophie.pid
      start program = "/etc/init.d/sophie start"
      stop  program = "/etc/init.d/sophie stop"
      if failed unix /var/run/sophie then restart
```

In this example we check an apache web-server running on localhost that answers for several IP-based virtual hosts or vhosts, hence the host statement before port:

```
check process apache with pidfile /var/run/httpd.pid
      start "/etc/init.d/httpd start"
      stop  "/etc/init.d/httpd stop"
      if failed host www.sol.no port 80 then alert
      if failed host shop.sol.no port 443 then alert
      if failed host chat.sol.no port 80 then alert
      if failed host www.tildeslash.com port 80 then alert
```

To make sure that Monit is communicating with a http server a protocol test can be added:

```
check process apache with pidfile /var/run/httpd.pid
      start "/etc/init.d/httpd start"
      stop  "/etc/init.d/httpd stop"
      if failed host www.sol.no port 80
         protocol HTTP
         then alert
```

This example shows a different way to check a webserver using the send/expect mechanism:

```
check process apache with pidfile /var/run/httpd.pid
      start "/etc/init.d/httpd start"
      stop  "/etc/init.d/httpd stop"
      if failed host www.sol.no port 80
         send "GET / HTTP/1.0\r\nHost: www.sol.no\r\n\r\n"
         expect "HTTP/[0-9\.]{3} 200 .*\r\n"
         then alert
```

To make sure that Apache is logging successfully (i.e. no more than 60 percent of child servers are logging), use its mod_status page at www.sol.no/server-status with this special protocol test:

```
check process apache with pidfile /var/run/httpd.pid
      start "/etc/init.d/httpd start"
      stop  "/etc/init.d/httpd stop"
      if failed host www.sol.no port 80
      protocol apache-status loglimit > 60% then restart
```

This configuration can be used to alert you if 25 percent or more of Apache child processes are stuck performing DNS lookups:

```
check process apache with pidfile /var/run/httpd.pid
      start "/etc/init.d/httpd start"
      stop  "/etc/init.d/httpd stop"
      if failed host www.sol.no port 80
      protocol apache-status dnslimit > 25% then alert
```

Here we use an icmp ping test to check if a remote host is up and if not send an alert:

```
check host www.tildeslash.com with address www.tildeslash.com
      if failed icmp type echo count 5 with timeout 15 seconds
         then alert
```

In the following example we ask Monit to compute and verify the checksum for the underlying apache binary used by the start and stop programs. If the the checksum test should fail, monitoring will be disabled to prevent possibly starting a compromised binary:

```
check process apache with pidfile /var/run/httpd.pid
      start program = "/etc/init.d/httpd start"
      stop program  = "/etc/init.d/httpd stop"
      if failed host www.tildeslash.com port 80 then restart
      depends on apache_bin

check file apache_bin with path /usr/local/apache/bin/httpd
      if failed checksum then unmonitor
```

In this example we ask Monit to test the checksum for a document on a remote server. If the checksum was changed we send an alert:

```
check host tildeslash with address www.tildeslash.com
      if failed port 80 protocol http
         and request "/monit/dist/monit-4.0.tar.gz"
             with checksum f9d26b8393736b5dfad837bb13780786
      then alert
```

Here are a couple of tests for some popular communication servers, using the SIP protocol. First we test a FreeSWITCH server and then an Asterisk server

```
check process freeswitch
   with pidfile /usr/local/freeswitch/log/freeswitch.pid
 start program = "/usr/local/freeswitch/bin/freeswitch -nc -hp"
 stop program = "/usr/local/freeswitch/bin/freeswitch -stop"
 if totalmem > 1000.0 MB for 5 cycles then alert
 if totalmem > 1500.0 MB for 5 cycles then alert
 if totalmem > 2000.0 MB for 5 cycles then restart
 if cpu > 60% for 5 cycles then alert
 if failed port 5060 type udp protocol SIP
    target me@foo.bar and maxforward 10
 then restart
 if 5 restarts within 5 cycles then timeout

check process asterisk
  with pidfile /var/run/asterisk/asterisk.pid
  start program = "/usr/sbin/asterisk"
  stop program = "/usr/sbin/asterisk -r -x 'shutdown now'"
  if totalmem > 1000.0 MB for 5 cycles then alert
  if totalmem > 1500.0 MB for 5 cycles then alert
  if totalmem > 2000.0 MB for 5 cycles then restart
  if cpu > 60% for 5 cycles then alert
  if failed port 5060 type udp protocol SIP
    and target me@foo.bar maxforward 10
  then restart
  if 5 restarts within 5 cycles then timeout
```

Some servers are slow starters, like for example Java based Application Servers. So if we want to keep the poll-cycle low (i.e. < 60 seconds) but allow some services to take its time to start, the **every** statement is handy:

```
check process dynamo with pidfile /etc/dynamo.pid
      start program = "/etc/init.d/dynamo start"
      stop program  = "/etc/init.d/dynamo stop"
      if failed port 8840 then alert
```

```
      every 2 cycles
```

Here is an example where we group together two database entries so you can manage them together, e.g.; 'Monit -g database start all'. The mode statement is also illustrated in the first entry and have the effect that Monit will not try to (re)start this service if it is not running:

```
check process sybase with pidfile /var/run/sybase.pid
      start = "/etc/init.d/sybase start"
      stop  = "/etc/init.d/sybase stop"
      mode passive
      group database

check process oracle with pidfile /var/run/oracle.pid
      start program = "/etc/init.d/oracle start"
      stop program  = "/etc/init.d/oracle stop"
      mode active # Not necessary really, since it's the default
      if failed port 9001 then restart
      group database
```

Here is an example to show the usage of the resource checks. It will send an alert when the CPU usage of the http daemon and its child processes raises beyond 60% for over two cycles. Apache is restarted if the CPU usage is over 80% for five cycles or the memory usage over 100Mb for five cycles or if the machines load average is more than 10 for 8 cycles:

```
check process apache with pidfile /var/run/httpd.pid
      start program = "/etc/init.d/httpd start"
      stop program  = "/etc/init.d/httpd stop"
      if cpu > 40% for 2 cycles then alert
      if totalcpu > 60% for 2 cycles then alert
      if totalcpu > 80% for 5 cycles then restart
      if mem > 100 MB for 5 cycles then stop
      if loadavg(5min) greater than 10.0 for 8 cycles then stop
```

This examples demonstrate the timestamp statement with exec and how you may restart apache if its configuration file was changed.

```
check file httpd.conf with path /etc/httpd/httpd.conf
      if changed timestamp
         then exec "/etc/init.d/httpd graceful"
```

In this example we demonstrate usage of the extended alert statement and a file check dependency:

```
check process apache with pidfile /var/run/httpd.pid
     start = "/etc/init.d/httpd start"
     stop  = "/etc/init.d/httpd stop"
     alert admin@bar on {nonexist, timeout}
       with mail-format {
             from:     bofh@$HOST
             subject:  apache $EVENT - $ACTION
             message:  This event occurred on $HOST at $DATE.
             Your faithful employee,
             monit
     }
     if failed host www.tildeslash.com  port 80 then restart
     if 3 restarts within 5 cycles then timeout
     depend httpd_bin
     group apache

check file httpd_bin with path /usr/local/apache/bin/httpd
      alert security@bar on {checksum, timestamp,
                  permission, uid, gid}
            with mail-format {subject: Alaaarrm! on $HOST}
      if failed checksum
         and expect 8f7f419955cefa0b33a2ba316cba3659
             then unmonitor
      if failed permission 755 then unmonitor
      if failed uid root then unmonitor
      if failed gid root then unmonitor
      if changed timestamp then alert
      group apache
```

In this example, we demonstrate usage of the depend statement. In this case, we want to start oracle and apache. However, we've set up apache to use oracle as a back end, and if oracle is restarted, apache must be restarted as well.

```
check process apache with pidfile /var/run/httpd.pid
      start = "/etc/init.d/httpd start"
      stop  = "/etc/init.d/httpd stop"
      depends on oracle

check process oracle with pidfile /var/run/oracle.pid
      start = "/etc/init.d/oracle start"
      stop  = "/etc/init.d/oracle stop"
      if failed port 9001 then restart
```

Next, we have 2 services, oracle-import and oracle-export that need to be restarted if oracle is restarted, but are independent of each other.

```
check process oracle with pidfile /var/run/oracle.pid
      start = "/etc/init.d/oracle start"
      stop  = "/etc/init.d/oracle stop"
      if failed port 9001 then restart

check process oracle-import
     with pidfile /var/run/oracle-import.pid
      start = "/etc/init.d/oracle-import start"
      stop  = "/etc/init.d/oracle-import stop"
      depends on oracle

check process oracle-export
     with pidfile /var/run/oracle-export.pid
      start = "/etc/init.d/oracle-export start"
      stop  = "/etc/init.d/oracle-export stop"
      depends on oracle
```

Finally an example with all statements:

```
check process apache with pidfile /var/run/httpd.pid
      start program = "/etc/init.d/httpd start"
      stop program  = "/etc/init.d/httpd stop"
      if 3 restarts within 5 cycles then timeout
      if failed host www.sol.no  port 80 protocol http
         and use the request "/login.cgi"
             then alert
      if failed host shop.sol.no port 443 type tcpssl
         protocol http and with timeout 15 seconds
             then restart
      if cpu is greater than 60% for 2 cycles then alert
      if cpu > 80% for 5 cycles then restart
      if totalmem > 100 MB then stop
      if children > 200 then alert
      alert bofh@bar with mail-format {from: monit@foo.bar.no}
      every 2 cycles
      mode active
      depends on weblogic
      depends on httpd.pid
      depends on httpd.conf
      depends on httpd_bin
      depends on datafs
      group server

check file httpd.pid with path /usr/local/apache/logs/httpd.pid
      group server
      if timestamp > 7 days then restart
      every 2 cycles
      alert bofh@bar with mail-format {from: monit@foo.bar.no}
      depends on datafs

check file httpd.conf with path /etc/httpd/httpd.conf
      group server
      if timestamp was changed
         then exec "/usr/local/apache/bin/apachectl graceful"
      every 2 cycles
      alert bofh@bar with mail-format {from: monit@foo.bar.no}
      depends on datafs

check file httpd_bin with path /usr/local/apache/bin/httpd
      group server
      if failed checksum and expect the sum
         8f7f419955cefa0b33a2ba316cba3659 then unmonitor
      if failed permission 755 then unmonitor
```

```
       if failed uid root then unmonitor
       if failed gid root then unmonitor
       if changed size then alert
       if changed timestamp then alert
       every 2 cycles
       alert bofh@bar with mail-format {from: monit@foo.bar.no}
       alert foo@bar on { checksum, size, timestamp, uid, gid }
       depends on datafs

 check filesystem datafs with path /dev/sdb1
       group server
       start program  = "/bin/mount /data"
       stop program  =  "/bin/umount /data"
       if failed permission 660 then unmonitor
       if failed uid root then unmonitor
       if failed gid disk then unmonitor
       if space usage > 80 % then alert
       if space usage > 94 % then stop
       if inode usage > 80 % then alert
       if inode usage > 94 % then stop
       alert root@localhost

 check host ftp.redhat.com with address ftp.redhat.com
       if failed icmp type echo with timeout 15 seconds
          then alert
       if failed port 21 protocol ftp
          then exec "/usr/X11R6/bin/xmessage -display
                     :0 ftp connection failed"
       alert foo@bar.com

 check host www.gnu.org with address www.gnu.org
       if failed port 80 protocol http
          and request "/pub/gnu/bash/bash-2.05b.tar.gz"
              with checksum 8f7f419955cefa0b33a2ba316cba3659
       then alert
       alert rms@gnu.org with mail-format {
            subject: The gnu server may be hacked again! }
```

Note; only the **check statement** is mandatory, the other statements are optional and the order of the optional statements is not important.

FILES

~/.monitrc Default run control file

/etc/monitrc If the control file is not found in the default location and /etc contains a *monitrc* file, this file will be used instead.

./monitrc If the control file is not found in either of the previous two locations, and the current working directory contains a *monitrc* file, this file is used instead.

~/.monit.pid Lock file to help prevent concurrent runs (non-root mode).

/var/run/monit.pid Lock file to help prevent concurrent runs (root mode, Linux systems).

/etc/monit.pid Lock file to help prevent concurrent runs (root mode, systems without /var/run).

~/.monit.state Monit save its state to this file and utilize information found in this file to recover from a crash. This is a binary file and its content is only of interest to monit. You may set the location of this file in the Monit control file or by using the -s switch when Monit is started.

~/.monit.id Monit save its unique id to this file.

ENVIRONMENT

No environment variables are used by Monit. However, when Monit execute a script or a program Monit will set several environment variables which can be utilized by the executable. The following and *only* the following environment variables are available:

MONIT_EVENT

The event that occurred on the service

MONIT_DESCRIPTION

A description of the error condition

MONIT_SERVICE

The name of the service (from monitrc) on which the event occurred.

MONIT_DATE

The time and date (rfc 822 style) the event occurred

MONIT_HOST

The host the event occurred on

The following environment variables are only available for process service entries:

MONIT_PROCESS_PID

The process pid. This may be 0 if the process was (re)started,

MONIT_PROCESS_MEMORY

Process memory. This may be 0 if the process was (re)started,

MONIT_PROCESS_CHILDREN

Process children. This may be 0 if the process was (re)started,

MONIT_PROCESS_CPU_PERCENT

Process cpu%. This may be 0 if the process was (re)started,

In addition the following spartan PATH environment variable is available:

PATH=/bin:/usr/bin:/sbin:/usr/sbin

Scripts or programs that depends on other environment variables or on a more verbose PATH must provide means to set these variables by them self.

SIGNALS

If a Monit daemon is running, SIGUSR1 wakes it up from its sleep phase and forces a poll of all services. SIGTERM and SIGINT will gracefully terminate a Monit daemon. The SIGTERM signal is sent to a Monit daemon if Monit is started with the *quit* action argument.

Sending a SIGHUP signal to a running Monit daemon will force the daemon to reinitialize itself, specifically it will reread configuration, close and reopen log files.

Running Monit in foreground while a background Monit daemon is running will wake up the daemon.

NOTES

This is a very silent program. Use the -v switch if you want to see what Monit is doing, and tail -f the logfile. Optionally for testing purposes; you can start Monit with the -Iv switch. Monit will then print debug information to the console, to stop monit in this mode, simply press CTRL^C (i.e. SIGINT) in the same console.

The syntax (and parser) of the control file was inspired by Eric S. Raymond et al. excellent fetchmail program. Some portions of this man page does also receive inspiration from the same authors.

AUTHORS

Jan-Henrik Haukeland <hauk@tildeslash.com>, Martin Pala <martinp@tildeslash.com>, Christian Hopp <chopp@iei.tu-clausthal.de>, Rory Toma <rory@digeo.com>

See also http://mmonit.com/monit/who/

COPYRIGHT

SEE ALSO

GNU text utilities; md5sum(1); sha1sum(1); `openssl(1)`; `glob(7)`; `regex(7)`; *http://mmonit.com/*