Version 5.16

NAME

Monit - utility for monitoring services on a Unix system

SYNOPSIS

monit [options] <arguments>

DESCRIPTION

Monit is a utility for managing and monitoring processes, programs, 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 can 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 customisable alert messages. Monit can perform various TCP/IP network checks, protocol checks and can utilise SSL for such checks. Monit provides a HTTP(S) interface and you may use a browser to access the Monit program.

WHAT TO MONITOR?

You can use Monit to monitor daemon **processes** or similar programs running on localhost. Monit is particularly useful for monitoring daemon processes, such as those started at system boot time. For instance sendmail, sshd, apache and mysql. In contrast to many other monitoring systems, Monit can act if an error situation should occur, e.g.; if sendmail is not running, monit can start sendmail again automatically or if apache is using too many 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 how much memory or cpu cycles a process is using.

You can 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 or sha1 checksum of files that should not change and get an alert or perform an action if they should change.

Monit can monitor **network connections** to various servers, either on localhost or on remote hosts. TCP, UDP and Unix Domain Sockets are supported. Network test can be performed on a protocol level; Monit has built-in tests for the main Internet protocols, such as HTTP, SMTP etc. Even if a protocol is not supported you can still test the server because you can configure Monit to send any data and test the response from the server.

Monit can be used to test **programs** or scripts at certain times, much like cron, but in addition, you can test the exit value of a program and perform an action or send an alert if the exit value indicates an error. This means that you can use Monit to perform any type of check you can write a script for.

Finally, Monit can be used to monitor general **system** resources on localhost such as overall CPU usage, Memory and System Load.

GENERAL OPERATION

The behaviour 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 default location for *monitrc* is *~/.monitrc*. If this file does not exist, Monit will try */etc/monitrc* and a few other places. See FILES for details. You can also specify the control file directly by using the *-c* command-line switch to monit. For instance,

```
$ monit -c /var/monit/monitrc
```

Before Monit is started the first time, you can test the control file for syntax errors:

```
$ monit -t
$ Control file syntax OK
```

If there was an error, Monit will print an error message to the console, including the line number in the control file from where the error was found.

Once you have a working Monit control file, simply start Monit from the console, like so:

```
$ monit
```

You can change some configuration directives via command-line switches, but for simplicity it is recommended that you put these in the control file.

Monit will detach from the terminal and run as a background process, i.e. as a daemon process. As a daemon, Monit runs in cycles; It monitor services, then goes to sleep for a configured period, then wakes up and start monitoring again in an endless loop.

Options

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

-c *file* Use this control file

-d *n* Run Monit as a daemon once per *n* seconds. Or use *"set daemon"* in monitrc.

-g *name* Set group name for start, stop, restart, monitor, unmonitor, status and summary action.

-l *logfile* Print log information to this file. Or use *"set logfile"* in monitrc.

-p *pidfile* Use this lock file in daemon mode. Or use *"set pidfile"* in monitrc.

-s *statefile* Write state information to this file. Or use *"set statefile"* in monitrc.

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

-i Print Monit's unique ID

-r Reset Monit's unique ID. Use with caution

-t Run syntax check for the control file

-v Verbose mode, work noisy (diagnostic output)

-vv Very verbose mode, same as -v plus log stack-trace on error

-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

Arguments

Once you have Monit running as a daemon process, you can call Monit with one of the following arguments. Monit will then connect to the Monit daemon (on TCP port 127.0.0.1:2812 by default) and ask the Monit daemon to perform the requested action. In other words; calling monit without arguments starts the Monit daemon, and calling monit *with* arguments enables you to communicate with the Monit daemon process.

start all

> Start all services listed in the control file and enable monitoring for them. If the group option is set (*-g*), only start and enable monitoring of services in the named group ("all" is not 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 ("all" is not 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 ("all" is not 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 ("all" is not 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 ("all" is not 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 [name]

Print service status information.

summary [name]

Print a short status summary.

reload

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

quit

Kill the Monit daemon process

validate

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

procmatch regex

Allows for easy testing of pattern for process match check. The command takes regular expression as an argument and displays all running processes matching the pattern.

THE MONIT CONTROL FILE

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. If you build Monit from source, the value of @sysconfdir@ can be given at configure time as ./configure --sysconfdir. For instance, using *./configure --sysconfdir /var/monit/etc* will make Monit search for *monitrc* in */var/monit/etc*

To protect the security of your control file and passwords the control file must have read-write permissions *no more than 0700* (u=xrw,g=,o=); Monit will complain and exit

otherwise.

When there is a conflict between the command-line arguments and the arguments in this file, the command-line arguments takes precedence.

Monit uses its own Domain Specific Language (DSL); The control file consists of a series of service entries and global option statements.

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.

You can use noise keywords like `'if'`, `'and'`, `'with(in)'`, `'has'`, `'us(ing|e)'`, `'on(ly)'`, `'then'`, `'for'`, `'of'` anywhere in an entry to make it resemble English. They're ignored, but can make entries much easier to read at a glance. Keywords are case insensitive.

There are three kinds of tokens: *grammar*, *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 three types of entries:

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. This statement is used to include configure directives from separate files.

3. 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, nine types of check statements are supported:

1. CHECK PROCESS <unique name> <PIDFILE <path> | MATCHING <regex>>

   <path> is the absolute path to the program's pid-file. A pid-file is a file, containing a Process's unique ID. If the pid-file does not exist or does not contain the PID number of a running process, Monit will call the entry's start method if defined.

   <regex> is an alternative to using PID files and uses process name pattern matching to find the process to monitor. The first match is used so this form of check is most useful if the process name is unique. Pid-file should be used where possible as it defines expected pid exactly (pattern matching won't be useful for processes which start a child process using fork/clone as the child will match the same pattern temporarily). You can test if a process match a pattern from the

command-line using `monit procmatch "regex-pattern"`. This will lists all processes matching or not, the regex-pattern.

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

   <path> is the absolute path to the file. If the file does not exist, 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 method is not defined, Monit will just send an alert on error.

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

   <path> is the absolute path to the fifo. If the fifo does not exist, 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 method is not defined, Monit will just send an alert on error.

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

   <path> is the path to the device/disk, 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), note 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 method is not defined, Monit will just send an alert on error.

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

   <path> is the absolute path to the directory. If the directory does not exist, 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 an alert on error.

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 *unique name* is usually the local host name, but any descriptive name can be used. If you use the variable $HOST as the name, it will expand to the hostname. This check allows one to monitor general system resources such as CPU usage, total memory usage or load average. The *unique name* is used as the system hostname in mail alerts and as the initial name of the host entry in M/Monit.

8. CHECK PROGRAM <unique name> PATH <executable file> [TIMEOUT <number> SECONDS]

<path> is the absolute path to the executable program or script. The status test allows one to check the program's exit status. If the program does not finish executing within <number> seconds, Monit will terminate it. The default program timeout is 300 seconds (5 minutes). The output of the program is recorded and made available in the User Interface and in alerts (by default up to 512B, you can customize the limit using the set limits statement).

9. CHECK NETWORK <unique name> <ADDRESS <ipaddress> | INTERFACE <name>>

<ipaddress> is the IPv4 or IPv6 address of the monitored network interface. It is also possible to use interface name, such as "eth0" on Linux.

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)

The format for logfile is:

```
[date] priority : message
```

for example:

```
[CET Jan  5 18:49:29] info : 'localhost' Monit started
```

DAEMON MODE

Use

```
SET DAEMON <seconds>
    [[WITH] START DELAY <seconds>]
```

to specify Monit's poll cycle length and run 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, wakes up and start monitoring again in an endless cycle.

Alternatively, you can use the `-d` command line switch to set the poll interval, but it is strongly recommended to set the poll interval in your *~/.monitrc* file, by using *set daemon*.

Monit will then always start in daemon mode. If you do not use this statement and do not start monit with the -d option, Monit will just run through the service checks once

and then exit. This might be useful in some situations, but Monit is primarily designed to run as a daemon process.

Calling `monit` with a Monit daemon running in the background sends a wake-up signal to the daemon, forcing it to check services immediately. Calling `monit` with the quit argument will kill a running Monit daemon process instead of waking it up.

The start delay option can be used to wait (once) before Monit starts checking services. This can be useful for example when the system boots. Monit will by default start checking services immediately at startup.

INIT SUPPORT

The `set init` statement prevents Monit from transforming itself into a daemon process. Instead Monit will run as a foreground process. (You should still use `set daemon` to specify the poll cycle).

This is required to run Monit from init. 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. Another option is to run Monit from crontab. In any case, you should make sure that the control file does not have any syntax errors before you start Monit from init or crontab (use `monit -t` to check).

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. 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 a solution to this problem. The short version is to start Monit on a higher run-level after system processes.

INCLUDE FILES

The Monit control file, `monitrc`, can include additional configuration files. This feature

helps one to organise configuration into separate files instead of having everything in one file, if you like this kind of thing. Include statements can be placed at virtually any place in `monitrc` though the convention is at the bottom. 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. If you want to use whitespace in your string the globstring needs to be embedded into quotes (') or double quotes ("). If the globstring matches a directory instead of a file, it is silently ignored.

Any *include* statements in an included file 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 should avoid wild-card globbing and instead specify the full path of files included.

An example,

```
include /etc/monit.d/*.cfg
```

This will load any file matching the globstring. That is, all files in */etc/monit.d* that ends with the prefix *.cfg*.

SSL OPTIONS

Common SSL/TLS options can be set using the following statement and will apply to all SSL connections made through Monit:

```
SET <SSL | TLS> [OPTIONS] {
    VERSION: <AUTO | SSLV2 | SSLV3 | TLSV1 | TLSV11 | TLSV12>
    VERIFY: <ENABLE | DISABLE>
    SELFSIGNED: <ALLOW | REJECT>
    CLIENTPEMFILE: <path>
    CACERTIFICATEFILE: <path>
    CACERTIFICATEPATH: <path>
}
```

VERSION set the specific SSL/TLS version to use. By default Monit uses AUTO. In AUTO mode, only TLS is used, SSLv2 and SSLv3 is considered obsolete. If you have to use SSLv2 or SSLv3, you must explicitly set the version.

VERIFY enable SSL server certificate verification. This will verify and report an error if the server certificate is not trusted, not valid or has expired. By default certificate verification is disabled, though we recommend enabling it, otherwise there is no guarantee that Monit speaks with the server you think it speaks with.

SELFSIGNED self-signed certificates are rejected by default. Use this option to allow self-signed certificates.

CLIENTPEMFILE set the path to the SSL client certificate "database-file" in PEM format. If an SSL server requires client certificate authentication, Monit will try to find a public key

certificate in this file which match the server's Certificate Authority and if found, use this certificate for client authentication.

CACERTIFICATEFILE set the path to the PEM encoded file containing Certificate Authority (CA) certificates. Monit uses OpenSSL's default CA certificates if this options is not used (*openssl version -d* can be used to get the default CA certificates). Many distributions comes with SSL and CA certificates already setup and using this option is normally not necessary.

CACERTIFICATEPATH set the path to the directory containing Certificate Authority (CA) certificates. Monit uses OpenSSL's default CA certificates if this options is not used. Many distributions comes with SSL and CA certificates already setup and using this option is normally not necessary.

The SSL options statement will globally apply to all SSL/TLS connection made through Monit. SSL options can also be set in a local check, in *mailserver* settings or in the *mmonit* statement, and will then override or extend the global settings.

To set global SSL options, put this statement near the top of your *.monitrc* file:

```
set ssl options {...}
```

Here is an example of setting both global and local SSL options:

```
# Enable certificate verification for all SSL connections
# Self-signed certificates are not allowed by default
set ssl options {
        verify: enable
}

# Verify certificate (via global setting)
# Allow self-signed certificate for this check
check host example with address example.com
        if failed
                port 443
                protocol https
                with ssl options {selfsigned: allow}
        then alert

# Do not verify example2.com's certificate (override global setting)
check host example2 with address example2.com
        if failed
                port 443
                protocol https
                with ssl options {verify: disable}
        then alert
```

MONIT HTTPD

If specified in the control file, Monit will start with HTTP support. You can then use Monit

CLI to start and stop services, disable or enable service monitoring as well as view the status of each service.

If HTTP support is enabled over TCP rather than over a Unix Socket, you can also view Monit's informative dashboard in your web browser.

Note that if HTTP support is disabled, the Monit CLI interface will have reduced functionality, as most CLI commands (such as "monit status") need to communicate with the Monit background process via the HTTP interface. We strongly recommend having HTTP support enabled. If security is a concern, bind the HTTP interface to local host only or use Unix Socket so Monit is not accessible from the outside.

Syntax for TCP port:

```
SET HTTPD PORT <number> [ADDRESS <hostname | IP-address>]
    [SSL <ENABLE | DISABLE>]
    [PEMFILE <path>]
    [CLIENTPEMFILE <path>]
    [ALLOWSELFCERTIFICATION]
    [SIGNATURE <ENABLE | DISABLE>]
    ALLOW <user:password | IP-address | IP-range>+
```

Example:

```
set httpd port 2812
    allow username:password
```

You can now use http://localhost:2812/ to access Monit's web interface from a browser, after you have entered username and password as credentials.

Syntax for Unix Socket:

```
SET HTTPD UNIXSOCKET <path>
    ALLOW <user:password>+
```

Example:

```
set httpd unixsocket /var/run/monit.sock
    allow username:password
```

Options:

UNIXSOCKET set the path to the Unix Socket Monit should bind to and listen on.

PORT set the port Monit should bind to and listen on. Monit is usually setup on port 2812.

ADDRESS make Monit listen on a specific interface only. For example if you *don't* want to expose Monit's web interface to the network, bind it to localhost only. Monit will accept connections on any address by default (if ADDRESS option is missing).

For example to limit the web interface to localhost only:

```
set httpd
    port 2812
```

```
    use address 127.0.0.1
    allow username:password
```

SSL enable TLS for Monit's web interface. The *PEMFILE* option 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, no more than 0700.

For example:

```
set httpd
    port 2812
    ssl enable
    pemfile /etc/certs/monit.pem
    allow myuser:mypassword
```

You can now use https://localhost:2812/ to access the Monit web server over a TLS encrypted connection.

OpenSSL FIPS is supported. To enable FIPS mode (provided your OpenSSL library supports it), add this statement to Monit control file:

```
SET FIPS
```

CLIENTPEMFILE client certificate based authentication. A connecting client has to provide a certificate known to Monit in order to connect. This file also needs to have all necessary CA certificates. 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 example:

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

SIGNATURE can be used to hide Monit version from the HTTP response header and error pages. For example:

```
set httpd
  port 2812
  signature disable
  allow myuser:mypassword
```

Authentication

Access to the Monit web interface is controlled primarily via the **ALLOW** option which is used to specify authentication and authorise only specific clients to connect.

If the Monit command line interface is being used, at least one cleartext password is necessary (see bellow), otherwise the Monit command line interface will not be able to connect to the Monit web interface.

Clients trying to connect to Monit, but submit a wrong username and/or password are logged with their IP-address.

Client certificates

This authentication method is a strong authentication mechanism and employ HTTPS client certificates to verify the authenticity of a connecting client. Clients must posses a Public Key Certificate known by Monit. The client must connect to Monit over SSL and Monit ask the client to send its certificate. Upon receiving the certificate Monit compares the certificate to certificates located in the *CLIENTPEMFILE* file. Access is granted if the client certificate is in this file. See *CLIENTPEMFILE* above for details.

Host and network allow list

Monit maintains an access-control list of hosts and networks allowed to connect. You can add as many hosts as you want to, but only hosts with a valid domain name or its IP address are allowed.

Monit will query a name server to check any hosts trying to connect. If a host (client) is trying to connect, but cannot be found in the access list or cannot be resolved, Monit 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 Monit will be denied access and are logged with their IP-address.

Basic Authentication

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

Cleartext user and password

Monit will use Basic Authentication if an allow statement contains a username and a password separated with a single ':' character. Special characters can be used, but for non-alphanumerics the password has to be quoted.

Syntax:

```
ALLOW <username>:<password>
```

PAM

PAM is supported on platforms which provide PAM (such as Linux, Mac OS X, FreeBSD, NetBSD).

Syntax:

```
ALLOW @<group>
```

where `group` is the group name allowed to access Monit's web interface. Monit uses a PAM service called *monit* for PAM authentication, see the PAM manual page for detailed instructions on how to set the PAM service and PAM authentication plugins.

Sample PAM service for Monit on Mac OS X (store as "/etc/pam.d/monit" file):

```
# 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 monitrc config which only allows group `admin` authenticated via PAM to access the web interface:

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

htpasswd file

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:"passw@rd"
```

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.

Read-only users

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.

ALERT MESSAGES

Monit will raise an alert in the following situations:

```
o A service does not exist (e.g. process is not running)
o Cannot read service data (e.g. cannot get filesystem usage)
o Execution of a service related script failed (e.g. start failed)
o Invalid service type (e.g. if path points to directory instead of file)
o Custom test script returned error
o Ping test failed
o TCP/UDP connection and/or port test failed
o Resource usage test failed (e.g. cpu usage too high)
o Checksum mismatch or change (e.g. file changed)
o File size test failed (e.g. file too large)
o Timestamp test failed (e.g. file is older then expected)
o Permission test failed (e.g. file mode doesn't match)
o An UID test failed (e.g. file owned by different user)
o A GID test failed (e.g. file owned by different group)
o A process' PID changed out of Monit's control
o A process' PPID changed out of Monit control
o Too many service recovery attempts failed
o A file content test found a match
o Filesystem flags changed
o A service action was performed by administrator
o A network link failed
o A network link capacity changed
```

```
o A network link saturation failed
o A network link upload/download rate failed
o Monit was started, stopped or reloaded
```

To get an alert via e-mail, set the alert target using the global `set alert` statement (for all services) or the `alert` statement in the context of a service entry (for a single service).

Setting an alert recipient

If an event occurs, Monit will send an alert. There are two kinds of alert statement: global and local.

Global syntax:

```
SET ALERT mail-address [[NOT] {event, ...}] [REMINDER cycles]
```

Example:

```
set alert foo@bar
```

will send a default email to the address foo@bar whenever any event occurs on any service.

If you want to send alert messages to more email addresses, add a `set alert 'email'` statement for each address.

It is also possible to use the local alert statement in the context of a service check to enable alert for the given service only:

```
ALERT mail-address [[NOT] {event, ...}] [REMINDER cycles]
```

Local alert example:

```
check host myhost with address 1.2.3.4
    if failed port 3306 protocol mysql then alert
    if failed port 80 protocol http then alert
    alert foo@baz # Local service alert
```

You can combine global and local alert statements. If there is a conflict, the local alert has precedence and overrides the global statement.

Setting an event filter

If you only want an alert message sent for certain events, list them in an `{event, ...}` block, e.g.:

```
set alert foo@bar only on { timeout, nonexist }
```

The event list can also be negated to send alerts for all events *except* those which are listed, by prepending the list with the word `not`. For example, to receive all alerts except notification about Monit program start and stop:

```
set alert foo@bar but not on { instance }
```

Here is a list of all possible event types emitted by Monit. Values from the first column can be used in the event filter list mentioned above:

```
Event:      | Failure state:               | Success state:
-------------------------------------------------------------------
action      | "Action done"                | "Action done"
checksum    | "Checksum failed"            | "Checksum succeeded"
bytein      | "Download bytes exceeded"    | "Download bytes ok"
byteout     | "Upload bytes exceeded"      | "Upload bytes ok"
connection  | "Connection failed"          | "Connection succeeded"
content     | "Content failed",            | "Content succeeded"
data        | "Data access error"          | "Data access succeeded"
exec        | "Execution failed"           | "Execution succeeded"
fsflags     | "Filesystem flags failed"    | "Filesystem flags succeeded"
gid         | "GID failed"                 | "GID succeeded"
icmp        | "Ping failed"                | "Ping succeeded"
instance    | "Monit instance changed"     | "Monit instance changed not"
invalid     | "Invalid type"               | "Type succeeded"
link        | "Link down"                  | "Link up"
nonexist    | "Does not exist"             | "Exists"
packetin    | "Download packets exceeded"  | "Download packets ok"
packetout   | "Upload packets exceeded"    | "Upload packets ok"
permission  | "Permission failed"          | "Permission succeeded"
pid         | "PID failed"                 | "PID succeeded"
ppid        | "PPID failed"                | "PPID succeeded"
resource    | "Resource limit matched"     | "Resource limit succeeded"
saturation  | "Saturation exceeded"        | "Saturation ok"
size        | "Size failed"                | "Size succeeded"
speed       | "Speed failed"               | "Speed ok"
status      | "Status failed"              | "Status succeeded"
timeout     | "Timeout"                    | "Timeout recovery"
timestamp   | "Timestamp failed"           | "Timestamp succeeded"
uid         | "UID failed"                 | "UID succeeded"
uptime      | "Uptime failed"              | "Uptime succeeded"
```

Each alert recipient can have it's own filter, for example:

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

Setting an error reminder

Monit by default sends just *one* notification if a service failed and another when/if it recovers. If you want to be notified that the service is still in a failed state, you can use the reminder option in the alert statement:

```
SET ALERT mail-address [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
```

Disabling alerts for some service

To suppress alerts for some user and service, add the `noalert` statement in the context of a service check.

```
NOALERT mail-address
```

Example (send all alerts to foo@bar except for service p3):

```
set alert foo@bar

check process p1 with pidfile /var/run/p1.pid

check process p2 with pidfile /var/run/p2.pid

check process p3 with pidfile /var/run/p3.pid
    noalert foo@bar
```

Message format

The alert message format can be modified by using the `set mail-format` statement:

```
SET MAIL-FORMAT {mail-format}
```

Example:

```
set mail-format {
     from: monit@foo.bar
 reply-to: support@domain.com
  subject: $SERVICE $EVENT at $DATE
  message: Monit $ACTION $SERVICE at $DATE on $HOST: $DESCRIPTION.
           Yours sincerely,
           monit
}
```

The *from:* option is the sender's email address. That is, the email address Monit will pretend it is sending alerts from. It does not have to be a real email-address only a proper formatted address.

The *reply-to:* option can be used to set the reply-to mail header.

The *subject:* option sets the message subject and must be on only *one* line.

The *message:* option sets the mail body. This option should always be the last in a mail-format statement. The mail body can be as long as needed, but must *not* contain the block-closing '}' character.

You need not use all options, only the option which you want to override. For example to globally change the senders address only:

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

The subject and body may contain $NAME variables, which are expanded by Monit. Here is a list of variables that can be used when composing an alert message.

- *$EVENT*

  A string describing the event that occurred.

- *$SERVICE*

  The service name

- *$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 by Monit.

- *$DESCRIPTION*

  The description of the error condition

Setting a mail server for alert delivery

The mail server Monit should use to send alert messages is defined with a `set mailserver` statement:

```
SET MAILSERVER
       <hostname|ip-address>
       [PORT number]
       [USERNAME string] [PASSWORD string]
       [using SSL [with options {...}]
       [CERTIFICATE CHECKSUM [MD5|SHA1] <hash>],
       ...
  [with TIMEOUT X SECONDS]
  [using HOSTNAME hostname]
```

Multiple mail servers can be set by using a comma separated list. If Monit cannot connect to the first server, it will try the next in the list and so on.

The port statement allows to override the default SMTP port (465 for SSL, or 25 for TLS and non secure connection).

Monit supports AUTH PLAIN and AUTH LOGIN for SMTP authentication. You can set a username and a password using the USERNAME and PASSWORD options.

You can set SSL/TLS options for the connection and also check a SSL certificate checksum.

The default connection timeout is 5 seconds. You can rise this limit using the TIMEOUT option.

Example (setting two mail servers for failover):

```
set mailserver smtp.gmail.com, smtp.other.host
```

By default, Monit uses the local host name in SMTP HELO/EHLO and in the Message-ID header. You can override this using the HOSTNAME option.

Event queue

If no mail server is available, Monit *can* queue events in the local file-system for retry until the mail server recovers.

If Monit is used with M/Monit, the event queue provides a safe event store for M/Monit in the case of temporary problems.

The event queue is persistent across Monit restarts and provided that the back-end filesystem is persistent, 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:

```
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.

Example:

```
  set eventqueue basedir /var/monit slots 5000
```

If you are running more then one Monit instance on the same machine, you **must** use separated event queue directories.

SERVICE METHODS

Each service can have associated *start*, *stop* and *restart* methods which Monit can use to execute action on the service.

Syntax:

```
<START | STOP | RESTART> [PROGRAM] = "program"
       [[AS] UID <number | string>]
       [[AS] GID <number | string>]
```

```
[[WITH] TIMEOUT <number> SECOND(S)]
```

If the `program` is a shell script it must begin with `#!` and the remainder of the first line must specify an interpreter for the program. e.g. `#!/bin/sh`

The `program` must also be executable (for example mode 0755).

It's possible to write scripts directly into the *program* this way:

```
stop = "/bin/bash -c 'kill -s SIGTERM `cat /var/run/process.pid`'"
```

By default the program is executed as the user under which Monit is running. If Monit is running as root, you may optionally specify the *UID* and *GID* the executed program should switch to.

Example:

```
check process mmonit with pidfile /usr/local/mmonit/mmonit/logs/mmonit.pid
  start program = "/usr/local/mmonit/bin/mmonit" as uid "mmonit" and gid "mmonit"
  stop program = "/usr/local/mmonit/bin/mmonit stop" as uid "mmonit" and gid "mmonit"
```

In the case of a process check, Monit will wait up to 30 seconds for the start/stop action to finish before giving up and report an error. You can override this timeout using the *TIMEOUT* option.

Example:

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

SERVICE POLL TIME

Services are checked in regular intervals by the `set daemon n` statement. Checks are performed in the same order as they are written in the `.monitrc` file, except if dependencies are setup between services, where pre-requisite services are tested first.

It is possible to modify a service check schedule by using the `every` statement.

There are three variants:

1. custom interval based on a poll cycle multiple

```
EVERY [number] CYCLES
```

2. check is schedule based on a cron-style string

```
EVERY [cron]
```

3. do-not-check schedule based on a cron-style string

```
NOT EVERY [cron]
```

A cron-style string, consist of 5 fields separated with white-space. All fields are required:

```
Name:        | Allowed values:            | Special characters:
-------------------------------------------------------------
Minutes      | 0-59                       | * - ,
Hours        | 0-23                       | * - ,
Day of month | 1-31                       | * - ,
Month        | 1-12 (1=jan, 12=dec)       | * - ,
Day of week  | 0-6 (0=sunday, 6=saturday) | * - ,
```

The special characters:

```
Character:   | Description:
-------------------------------------------------------------
* (asterisk) | The asterisk indicates that the expression will
             | match for all values of the field; e.g., using
             | an asterisk in the 4th field (month) would
             | indicate every month.
- (hyphen)   | Hyphens are used to define ranges. For example,
             | 8-9 in the hour field indicate between 8AM and
             | 9AM. Note that range is from start time until and
             | including end time. That is, from 8AM and until
             | 10AM unless minutes are set. Another example,
             | 1-5 in the weekday field, specify from monday to
             | friday (including friday).
, (comma)    | Comma are used to specify a sequence. For example
             | 17,18 in the day field indicate the 17th and 18th
             | day of the month. A sequence can also include
             | ranges. For example, using 1-5,0 in the weekday
             | field indicate monday to friday and sunday.
```

Example 1: Check once per two cycles

```
check process nginx with pidfile /var/run/nginx.pid
      every 2 cycles
```

Example 2: Check every workday 8AM-7PM

```
check program checkOracleDatabase
       with path /var/monit/programs/checkoracle.pl
      every "* 8-19 * * 1-5"
```

Example 3: Do not run the check in the backup window on Sunday 0AM-3AM

```
check process mysqld with pidfile /var/run/mysqld.pid
      not every "* 0-3 * * 0"
```

Limitations:

The current scheduler is poll cycle based. When a service check is constraint with the *every cron* statement, Monit will check whether the current time match the cron-string pattern. If it does, the check is performed, otherwise it is skipped. The cron specification thus does not guarantee when exactly the test will run, this depends on the default poll time and the length of the check cycle. In other words, we cannot guarantee that Monit

will run on a specific time. Therefor we **strongly** recommend to use an asterix in the minute field or at minimum a range, e..g. 0-15. **Never** use a specific minute as Monit may not run on that minute.

We will address this limitation in a future release and convert the scheduler from serial polling into a parallel non-blocking scheduler where checks are guaranteed to run on time and with seconds resolution.

SERVICE GROUPS

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
```

A service can be added to multiple groups by using more than one group statement:

```
group www
group filesystem
```

SERVICE MONITORING MODE

Monit supports three monitoring modes per service: *active*, *passive* and *manual*.

Syntax:

```
MODE <ACTIVE | PASSIVE | MANUAL>
```

In *active* mode (the default), Monit will pro-actively monitor a service and in case of problems raise alerts and/or restart the service.

In *passive* mode, Monit will passively monitor a service and will raise alerts, but will **not** try to fix a problem by executing *start*, *stop* or *restart*.

In *manual* mode, Monit will enter *active* mode *only* if a service was started via Monit:

```
monit start <servicename>
```

Use "monit stop <servicename>" to stop the service and take it out of Monit's control. The manual mode can be used to build a simple cluster with active/passive HA-services.

A service's monitoring state is persistent across Monit restart.

If you use Monit in a HA-cluster you should place the Monit state file in a temporary filesystem so if the machine which runs HA-services should crash and the stand-by machine take over its services, the HA-services won't be started after the crashed node will boot again:

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

SERVICE RESTART LIMIT

Monit provides a restart limit mechanism for situations where a service simply refuses to start or respond over a longer period.

The restart limit mechanism is based on number of 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>
```

The *action* value is either one of common actions or TIMEOUT (for backward compatibility, equals to UNMONITOR 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 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 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 check service entry name used in your `.monitrc` file, 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.

If the service is started, all services which this service depends on will be started before starting this service. if start of some service failed, the service with prerequisites will NOT be started and the, but will remember that it should start and will retry next cycle.

If a service is restarted, it will first stop any active services that depend on it and after it is started, start all depending services that were active before the restart again.

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 with pidfile "/var/run/httpd.pid"
(2)    depends on httpd
(3)    ...
(4)
(5) check file httpd with path /usr/bin/httpd
(6)    if failed checksum then stop
```

The first entry is the process entry for apache. The second line sets up a dependency between this entry and the service entry named httpd in line 5. A dependency 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 6 then an stop action is executed and apache binary is not checked anymore. But since the apache process entry depends on the httpd entry this entry will also execute the stop action. In short, if the checksum test for the httpd binary file should fail, both the check file httpd and the check process apache entry are stopped.

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 filesystem 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 services 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

Monit will start *a*

If *b* does not run

Monit will first stop *a* then start *b* and finally start *a* if *b* is up again.

If *c* does not run

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

If *d* does not run

Monit 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.

SERVICE TESTS

LIMITS

You can configure and set various limits to tweak buffer sizes and timeouts used by Monit. In most situations the default values are fine. If needed, below are the limits you can currently modify in Monit.

Syntax:

```
SET LIMIT {
  PROGRAMOUTPUT:     <number> <unit>,
  SENDEXPECTBUFFER:  <number> <unit>,
  FILECONTENTBUFFER: <number> <unit>,
  HTTPCONTENTBUFFER: <number> <unit>,
  NETWORKTIMEOUT:    <number> <timeunit>
```

```
}
```

Where: *unit* is "B" (byte), "kB" (kilobyte) or "MB" (megabyte) *timeunit* is "MS" (millisecond) or "S" (second)

Options legend:

Option	Description	Default
programOutput	limit for check program output (truncated after)	512 B
sendExpectBuffer	limit for send/expect protocol test	256 B
fileContentBuffer	limit for file content test (line)	512 B
httpContentBuffer	limit for HTTP content test (response body)	1 MB
networkTimeout	timeout for network I/O	5 sec

GENERAL SYNTAX

Monit offers several if-tests you can use in a 'check' statement to test various aspects of a service.

You can test both for a predefined value or for a range and take actions if the value changes.

General syntax for testing a specific value or range:

```
IF <test> THEN <action> [ELSE IF SUCCEEDED THEN <action>]
```

The action is evaluated each time the <TEST> condition is true. Success action is optional and executed only when the state changes from failure to success. If success action is not set, Monit will send a recovery alert by default.

General syntax for a value change test:

```
IF CHANGED <test> THEN <action>
```

The action is executed each time the value changes. Monit will remember the new value and will trigger event if the value change again.

ACTION

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.
- **RESTART** restarts the service **and** send an alert. Restart is performed by calling the service's registered restart method or by first calling the stop method followed by the start method if restart is not set.
- **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.

- **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 requires 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. The program is by default executed only once, on the state change. You can enable program repetition when the error persists for given number of cycles. For instance:

```
if failed <test> then exec "/usr/local/bin/sms.sh"
     as uid nobody and gid nobody
     repeat every 5 cycles
```

  Remember, if Monit is run by root, then all programs executed by Monit will be started with superuser privileges unless the uid and gid extension is used.

- **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 the web interface or from the console.

FAULT TOLERANCE

By default an action is executed if it matches and the corresponding service is set in an error state. However, you can require a test to fail more than once before the error event is triggered and the service state is changed to failed. This is useful to avoid getting alerts on spurious errors, which can happen, especially with network tests.

Syntax:

```
FOR <X> CYCLES ...
```

or:

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

The condition can be used both for failure and success action.

The first, simpler and recommended format requires `X` consecutive events before switching the state:

```
if failed
   port 80
   for 3 cycles
then alert
```

The second format is more advanced and allows to tolerate intermittent issues, but still catch excessive problems, where the service is flapping between error and success states frequently.

For example if every second cycle fails (1-0-1-0-1-0-...), then "for 2 cycles" condition will never match, despite the service having problems. The following statement will catch

such a state:

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

Example which sets multiple error levels and actions:

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

Note: the maximum value for cycles is 64.

EXISTENCE TESTING

This test is implicite and always active for service checks of type, *process*, *file*, *directory* and *fifo*. If not defined, it defaults to a restart action.

You can override the default action with the following statement:

```
IF [DOES] NOT EXIST THEN <action>
```

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

Example:

```
check file with path /cifs/mydata
      if does not exist then exec "/usr/bin/mount_cifs.sh"
```

RESOURCE TESTING

Monit can examine how much system resources a service is 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 utilise systems which are idle and to spare system under high load.

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

```
IF <resource> <operator> <value> THEN <action>
```

resource is a choice of "CPU", "TOTAL CPU", "CPU([user|system|wait])", "MEMORY", "SWAP", "THREADS", "CHILDREN", "TOTAL MEMORY", "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 kernel space

and I/O. The user/system/wait modifier is optional, if not used, the total system cpu usage is tested.

SWAP is the swap usage of the system in either percent (of the systems total) or as an amount (Byte, kB, MB, GB).

Process only resource tests:

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

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

THREADS is the number of processes' threads.

CHILDREN is the number of child processes of the process.

TOTAL MEMORY 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. For CPU, TOTAL CPU, MEMORY and TOTAL MEMORY 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" 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
```

FILE CHECKSUM TESTING

The checksum statement may only be used in a file service entry and can be used to check the file's MD5 or SHA1 checksum.

Check specific checksum:

```
IF FAILED [MD5|SHA1] CHECKSUM [EXPECT checksum] THEN action
```

Check any file changes:

```
IF CHANGED [MD5|SHA1] CHECKSUM THEN action
```

The choice of MD5 or SHA1 is optional. MD5 features a 128 bits checksum (32 bytes hex encoded string) and SHA1 a 160 bits checksum (40 bytes hex encoded string). If this option is omitted, Monit will try to guess the method from the EXPECT string or use MD5 as the default checksum.

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

```
if failed
   checksum expect 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.

Reloading a server if its configuration file was changed:

```
check file apache_conf with path /etc/apache/httpd.conf
    if changed checksum then exec "/usr/bin/apachectl graceful"
```

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

TIMESTAMP TESTING

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

Specific timestamp syntax:

```
IF TIMESTAMP [[operator] value [unit]] THEN action
```

Timestamp changed syntax:

```
IF CHANGED TIMESTAMP 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(S)", "MINUTE(S)", "HOUR(S)" or "DAY(S)".

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

For example to reload apache if the configuration file timestamp changed:

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

For example testing directory for file addition or removal:

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

FILE SIZE TESTING

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

Testing specific size or range:

```
IF SIZE [[operator] value [unit]] THEN action
```

Testing size changes:

```
IF CHANGED SIZE 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" or "UNMONITOR".

For example to send an alert if the file is too large:

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

FILE CONTENT TESTING

The content statement can be used to incrementally test the content of a text file by using regular expressions.

Syntax:

```
IF CONTENT <operator> <regex|path> THEN action
```

operator is either a "=" for match or "!=" for no-match.

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" or "UNMONITOR".

On startup the read position is set to the end of the file and Monit continues to scan to the end of the file on each cycle.

If the file size should decrease or inode changed, the read position is set to the start of the file.

Only lines ending with a newline character are inspected.

By default only the first 511 characters of a line are inspected. You can increase the limit using the set limits statement.

```
IGNORE CONTENT <operator> <regex|path>
```

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

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

For example:

```
check file syslog with path /var/log/syslog
      ignore content = "^monit"
      if content = "^mrcoffee" 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 filesystem flags such as if the filesystem become read-only (on disk error) or mount flags were changed (such as nosuid).

The syntax for the fsflags statement is:

```
IF CHANGED FSFLAGS THEN action
```

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

Example:

```
check filesystem rootfs with path /
      if changed fsflags then exec "/my/script"
```

SPACE TESTING

Monit can test a filesystem or a disk for space usage. This test may only be used in the context of a filesystem service type.

Filesystems usually have some space reserved for the root user (ca. 1-5%), so non-superusers cannot write to a nearly full filesystem. If you set a limit for the filesystem which is used by non-root users you might want to consider these reserved blocks when setting the limit. You can use Monit itself to view the reserved blocks percentage by using the CLI status command or the HTTP interface for the given filesystem.

Syntax:

```
IF SPACE operator value unit THEN action
```

or:

```
IF SPACE FREE operator value unit 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" or "UNMONITOR".

Example:

```
check filesystem rootfs with path /
      if space usage > 90% then alert
```

INODE TESTING

Monit can test filesystem inode usage. This test may only be used in the context of a filesystem service type.

Syntax:

```
IF INODE(S) operator value [unit] THEN action
```

or:

```
IF INODE(S) FREE operator value [unit] 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" or "UNMONITOR".

Example:

```
check filesystem rootfs with path /
      if inode usage > 90% then alert
```

PERMISSION TESTING

Monit can test the permissions of file objects. This test may only be used in the context of a file, fifo, directory or filesystem service types.

Syntax for testing specific permissions:

```
IF FAILED PERM(ISSION) octalnumber THEN action
```

Syntax for testing any permission change:

```
IF CHANGED PERM(ISSION) THEN action
```

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

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

Example:

```
check file shadow with path /etc/shadow
      if failed permission 0640 then alert
```

UID TESTING

Monit can monitor the owner user id (uid) of a file, fifo, directory or owner and effective user of a process.

Syntax:

```
IF FAILED [E]UID user THEN action
```

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

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

Example:

```
check file shadow with path /etc/shadow
      if failed uid root then alert
```

GID TESTING

Monit can monitor the owner group id (gid) of a file, fifo, directory or process.

Syntax:

```
IF FAILED GID group THEN action
```

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

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

Example:

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

PID TESTING

Monit can test the process' PID. This test is implicit and Monit will send an alert in case the PID changed outside of Monit's control.

Syntax:

```
IF CHANGED PID THEN action
```

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

This test is useful to detect possible process restarts which has occurred in the timeframe between two Monit testing cycles.

For example if someone changes sshd configuration and did sshd restart outside of Monit's control you will be notified that the process was replaced by a new instance:

```
check process sshd with pidfile /var/run/sshd.pid
      if changed pid then alert
```

PPID TESTING

Monit can test the process' parent PID (PPID) for changes. This test is implicit and Monit will send alert in the case that the PPID changed outside of Monit control.

The syntax for the ppid statement is:

```
IF CHANGED PPID THEN action
```

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

Example:

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

PROCESS UPTIME TESTING

The uptime statement may only be used in a process service type context.

Syntax:

```
IF UPTIME [[operator] value [unit]] 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 uptime 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" or "UNMONITOR".

Example of restarting the process every three days:

```
check process myapp with pidfile /var/run/myapp.pid
   start program = "/etc/init.d/myapp start"
   stop program = "/etc/init.d/myapp stop"
   if uptime > 3 days then restart
```

PROGRAM STATUS TESTING

You can check the exit status of a program or a script. This test may only be used within a check program service entry in the Monit control file.

Syntax for testing specific exit value:

```
IF STATUS operator value THEN action
```

Syntax for testing any exit value change:

```
IF CHANGED STATUS 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).

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

Example:

```
check program myscript with path /usr/local/bin/myscript.sh
      if status != 0 then alert
```

Sample script for the above example (/usr/local/bin/myscript.sh):

```
#!/bin/bash
echo test
exit $?
```

You can also send parameters with the program:

```
check program list-files with path "/bin/ls -lrt /tmp/"
      if status != 0 then alert
```

Arguments to the program or script is a sequence of whitespace separated strings. In the above example the strings '-lrt' and '/tmp/' are arguments to the program '/bin/ls'. If arguments are used, it is recommended to use quotes " to enclose the string, otherwise, if no arguments are used, quotes are not needed.

Notes: If the program is a script, the interpreter is required in the first line. The program or script must also be executable.

If Monit is run as the super user, you can optionally run the program as a different user and/or group. In this example we run the *ls* program as user www and as group staff:

```
check program ls with path "/bin/ls /tmp" as uid "www"
         and gid "staff"
      if status != 0 then alert
```

Monit will execute the program periodically and if the exit status of the program does not match the expected result, Monit can perform an action. In the example above, Monit will raise an alert if the exit value is different from 0. By convention, 0 means the program exited normally.

Program checks are asynchronous. Meaning that Monit will not wait for the program to exit, but instead, Monit will start the program in the background and immediately continue checking the next service entry in *monitrc*. At the next cycle, Monit will check if the program has finished and if so, collect the program's exit status. If the status indicate a failure, Monit will raise an alert message containing the program's error (stderr) output, if any. If the program has not exited after the first cycle, Monit will wait another cycle and so on. If the program is still running after 5 minutes, Monit will kill it and generate a program timeout event. It is possible to override the default timeout (see the syntax below).

The asynchronous nature of the program check allows for non-blocking behaviour in the current Monit design, but it comes with a side-effect: when the program has finished executing and is waiting for Monit to collect the result, it becomes a so-called "zombie" process. A zombie process does not consume any system resources (only the PID remains in use) and it is under Monit's control and the zombie process is removed from the system as soon as Monit collects the exit status. This means that every "check program" will be associated with either a running process or a temporary zombie. This unwanted zombie side-effect will be removed in a later release of Monit.

Multiple status tests can be used, for example:

```
check program hwtest with path /usr/local/bin/hwtest.sh
      with timeout 500 seconds
      if status = 1 then alert
      if status = 3 for 5 cycles then exec "/usr/local/bin/emergency.sh"
```

NETWORK LINK STATUS TEST

You can check the network link state. This test may only be used within a check network service entry in the Monit control file.

Syntax:

```
IF FAILED LINK THEN action
```

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

The test will fail if the link/interface is down or link errors were detected.

Example:

```
check network eth0 with interface eth0
      if failed link then alert
```

In case a link failed you can add a start and stop program to automatically restart the interface which might help. (Substitute with the relevant network commands for your system)

```
check network eth0 with interface eth0
      start program = '/sbin/ipup eth0'
      stop program = '/sbin/ipdown eth0'
      if failed link then restart
```

NETWORK LINK CAPACITY TEST

You can check the network link mode capacity for changes. This test may only be used within a check network service entry in the Monit control file.

Syntax:

```
IF CHANGED LINK [CAPACITY] THEN action
```

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

The test will match if the link mode has changed (e.g. maximum speed dropped) or if the duplex mode has changed.

NOTE: not all interface types allow for capacity monitoring. Pseudo interfaces such as loopback device or VMWare interfaces does not have a speed attribute.

Example:

```
check network eth0 with interface eth0
      if changed link capacity then alert
```

NETWORK SATURATION TEST

You can check the network link saturation. Monit then computes the link utilisation based on the current transfer rate vs. link capacity. This test may only be used within a check network service entry in the Monit control file.

Syntax:

```
IF SATURATION operator value% 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).

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

NOTE: this test depends on the availability of the speed attribute and not all interface types have this attribute. See the LINK SPEED test description.

Example:

```
check network eth0 with interface eth0
      if saturation > 90% then alert
```

NETWORK BANDWIDTH TEST

You can check a network link upload and download bandwidth usage, current transfer rate and total data transferred in the last 24 hours. This test may only be used within a check network service entry in the Monit control file.

Warning: this test requires monit poll time to be at maximum 30 seconds.

Current upload bandwidth rate test syntax:

```
IF UPLOAD operator value unit THEN action
```

Current download bandwidth rate test syntax:

```
IF DOWNLOAD operator value unit THEN action
```

Total upload test syntax:

```
IF TOTAL UPLOAD operator value unit IN LAST number time-unit THEN action
```

Total download test syntax:

```
IF TOTAL DOWNLOAD operator value unit IN LAST number time-unit 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".

time-unit is a choice of "MINUTE(S)", "HOUR(S)", "DAY". NOTE: Monit maintains a rolling count of total uploaded and downloaded bytes for the last 24 hours only. The value of time-unit can therefor not specify a range wider than one day.

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

Examples:

```
check network eth0 with interface eth0
      if upload > 500 kB/s then alert
      if total download > 1 GB in last 2 hours then alert
      if total download > 10 GB in last day then alert
```

NETWORK PACKETS TEST

You can check the network link upload and download packets count, current transfer rate and total data transferred in last 24 hours. This test may only be used within a check network service entry in the Monit control file.

Warning: this test requires monit poll time to be at maximum 30 seconds.

Current upload bandwidth rate test syntax:

```
IF UPLOAD operator value PACKETS/S THEN action
```

Current download bandwidth rate test syntax:

```
IF DOWNLOAD operator value PACKETS/S THEN action
```

Total upload test syntax:

```
IF TOTAL UPLOAD operator value PACKETS IN LAST number time-unit THEN action
```

Total download test syntax:

```
IF TOTAL DOWNLOAD operator value PACKETS IN LAST number time-unit 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).

time-unit is a choice of "MINUTE(S)", "HOUR(S)", "DAY". NOTE: Monit keeps total upload/download statistics only for the last 24 hours. The time-unit value cannot therefor span more than one day.

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

Examples:

```
check network eth0 with interface eth0
      if upload > 1000 packets/s then alert
      if total upload > 900000 packets in last hour then alert
```

NETWORK PING TEST

Monit can perform a network ping test by sending ICMP echo request datagram packets to a host and wait for the reply. This test can only be used within a check host statement. Monit must also run as the root user in order to be able to perform the ping test (because the ping test must use raw sockets which usually only the super user is allowed to).

Syntax:

```
IF FAILED PING[4|6]
   [COUNT number]
   [SIZE number]
   [TIMEOUT number SECONDS]
   [ADDRESS string]
THEN action
```

If a DNS host name was used in the *check host* statement and the host name resolve to several addresses (either IPv4 or IPv6), Monit will ping the first available address and continue with the next address until one connection succeed or until there are no more addresses left to try. You can force Monit to only ping IPv4 or IPv6 addresses by using the PING4 or the PING6 keyword instead of PING.

The **COUNT** parameter specifies how many consecutive ping requests will be sent to the host in one cycle at maximum. The default value is 3.

The **SIZE** parameter specifies the ping request data size. Default is 64 bytes.

If no reply arrive within **TIMEOUT** seconds, Monit reports an error. If at least one reply was received, the ping test is considered a success.

The **ADDRESS** parameter specifies source IP address.

Monit will, by default, send up to *three* ping request packets in one cycle to prevent false alarm (i.e. up to 66% packet loss is tolerated). You can set the **COUNT** option to a value between 1 and 20 to send more or fewer packets. 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).

Note that many ISPs have started to filter out ping or ICMP packets now, in which case there will be no reply from the host.

If a ping test is used in a check host entry, this test is run first and if the test should fail, we assume that the connection to the host is down and Monit will *not* continue with any subsequent port tests.

Example:

```
check host mmonit.com with address mmonit.com
      if failed ping then alert  # IPv4 or IPv6

check host mmonit.com with address 62.109.39.247
      if failed ping then alert # Address is IPv4 so IPv4 is preferred
```

or test that the system is explicit accessible via IPv4 and IPv6:

```
check host mmonit.com with address mmonit.com
      if failed ping4 then alert  # IPv4 only
      if failed ping6 then alert  # IPv6 only
```

or with all parameters; Send five 128 byte pings to mmonit.com and wait for up to 10 seconds for a reply

```
  check host mmonit.com with address mmonit.com
       if failed ping count 5 size 128 with timeout 10 seconds then alert
```

CONNECTION TESTING

Monit can perform connection testing via network ports or via Unix sockets. A connection test may only be used within a process or host service type context.

If a service listens on one or more sockets, Monit can connect to the port (using 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 execute a specified action.

TCP/UDP port test syntax:

```
IF FAILED
   [host]
   <port>
```

```
    [localaddress]
    [ipversion]
    [type]
    [ssl [with options {...}]]
    [certificate checksum]
    [certificate valid days]
    [protocol | <send|expect>, ...]
    [timeout]
    [retry]
 THEN action
```

Unix socket test syntax:

```
 IF FAILED
    <unixsocket>
    [type]
    [protocol | <send|expect>, ...]
    [timeout]
    [retry]
 THEN action
```

Examples:

```
 if failed port 80 then alert

 if failed port 53 type udp protocol dns then alert

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

Options:

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 is used inside a remote host entry then the entry's remote host is assumed.

port: PORT number. The port number to connect to

unixsocket: UNIXSOCKET path. Specifies the path to a Unix socket (local machine only).

localaddress: ADDRESS string. The source IP address to use.

ipversion: IPV4 | IPV6 . Optionally specify the IP version Monit should use when trying to connect to the port. If not used, Monit will try to connect to the first available address (IPv4 or IPv6). If multiple addresses are available and connection to one address failed, Monit will try the next address and so on until a connection succeed or until there are no more addresses left to try.

type: TYPE [TCP | UDP]. Optionally specify the socket type Monit should use when trying to connect to the port. The different socket types are: TCP or UDP, where TCP is a regular stream based socket, UDP, a datagram socket. The default socket type is TCP.

ssl: [SSL | TLS] [with options {...}]. Set SSL/TLS options and override global/default SSL options. You can set the SSL/TLS version to use, whether to verify certificates, trust self-

signed certificates or set the SSL client certificates database-file for client certificate authentication.

certificate checksum: CERTIFICATE CHECKSUM [MD5|SHA1] hash. Verify the SSL server certificate by checking its checksum. You can use either MD5 or SHA1 checksum (if you don't specify the type, Monit will determine the digest based on the hash length). You can use the *openssl* command line tool to get the checksum value for your certificate, which you can then use in Monit's control file:

```
openssl x509 -fingerprint -sha1 -in server.crt | head -1 | cut -f2 -d'='
```

Example:

```
if failed
    port 443
    protocol https
    and certificate checksum = "1ED948A6F4258ACAB964227EF4EB19FCC453B0F8"
then alert
```

certificate expire days: CERTIFICATE VALID for number DAYS. Send an alert if the certificate will expire in the given number of days. This test is pretty useful to get a notification when it is time to renew your SSL certificate.

Example:

```
if failed
    port 443
    protocol https
    and certificate valid > 30 days
then alert
```

protocol: PROTO(COL) protocol. 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 HTTPS IMAP IMAPS CLAMAV LDAP2 LDAP3 LMTP MEMCACHE MONGODB MYSQL NNTP NTP3 PGSQL POP POPS POSTFIX-POLICY RADIUS RDATE REDIS RSYNC SIEVE SIP SMTP SMTPS SSH TNS WEBSOCKET*

If the target server's protocol is not found in this list, simply do not specify the protocol and Monit will use a default connection test.

timeout: [WITH] TIMEOUT number 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.

retry: RETRY number. Optionally specifies the number of consecutive retries within the same testing cycle in the case that the connection failed. The default is fail on first error.

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

Specific protocol test options

GENERIC (SEND/EXPECT)

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.

Syntax:

```
[<SEND|EXPECT> "string"]+
```

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 expects such characters to terminate a string (most text based protocols used over Internet do).

Monit will by default read up to 255 bytes from the server and use this string when comparing the EXPECT string. You can override the default value using the set limits statement.

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. For example, to test a Quake 3 server:

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

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.

Since both regex and string compare operates on a zero terminated string, you cannot test for '\0' in an EXPECT buffer since this character marks the end of the buffer. However, we escape '\0' in the expect buffer as "\0" which you can test for. That is, '\' followed by the ascii value for 0. For instance, here is how to test for an expect string that starts with zero followed by any number of characters.

```
expect "^[\\]0.*"
```

Here is a simple SMTP protocol example:

```
if failed
   port 25 and
   expect "^220.*"
   send   "HELO localhost.localdomain\r\n"
   expect "^250.*"
   send   "QUIT\r\n"
then alert
```

SEND/EXPECT can be used with any socket type, such as TCP sockets, UNIX sockets and UDP sockets.

HTTP

Syntax:

```
PROTO(COL) HTTP
```

```
    [REQUEST "string"]
    [STATUS operator number]
    [CHECKSUM checksum]
    [HTTP HEADERS list of headers]
    [CONTENT < "=" | "!=" > STRING]
```

REQUEST option can set an URL string specifying a document on the HTTP server. If the request statement isn't specified, the default "/" page will be requested.

For example:

```
if failed
   port 80
   protocol http
   request "/data/show?a=b&c=d"
then restart
```

STATUS option can be used to explicitly test the HTTP status code returned by the HTTP server. If not used, the HTTP protocol test will fail if the status code returned is greater than or equal to 400. You can override this behaviour by using the *status* qualifier.

For example to test that a page does **not** exist (the HTTP server should return 404 in this case):

```
 if failed
    port 80
    protocol http
    request "/non/existent.php"
    status = 404
 then alert
```

CHECKSUM You can test the checksum of documents returned by a HTTP server. Either MD5 or SHA1 hash can be used. 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). There are no limitation on the document size, but keep in mind that Monit will use time to download the document over the network to compute the checksum.

Example:

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

HTTP HEADERS can be used to send a list of any HTTP headers with a HTTP protocol test. For instance, the host header. If the host header is not set, Monit will use the hostname or IP-address of the host as specified in the check host statement. Specifying a host header is useful if you want to connect to and test a name-based virtual host. The syntax for setting HTTP headers is

```
  http headers [name:value, name:value,..]
```

where each name:value pair is separated with ','. If you need to use ':' in the value string, for instance to set port number for a host header, you must enclose the value in quotes. For example,

```
  http headers [Host: "mmonit.com:443"]
```

In a check host context, using this statement might look like

```
  check host mmonit.com with address mmonit.com
    if failed
       port 80 protocol http
       with http headers [Host: mmonit.com, Cache-Control: no-cache,
         Cookie: csrftoken=nj1bI3CnMCaiNv4beqo8ZaCfAQQvpgLH]
       and request /monit/ with content = "Monit [0-9.]+"
    then alert
```

Setting HTTP headers is associated with the HTTP protocol test and must come before *request* as in the example above.

CONTENT option sets the pattern which is expected in the data returned by the server. If the pattern doesn't match, event is triggered.

By default at maximum 1MB is inspected. You can increase the limit using the set limits statement.

For example:

```
  if failed
     port 80
     protocol http
     content = "foobar [0-9.]+"
  then alert
```

APACHE-STATUS

The *APACHE-STATUS* test allows to check server performance by examination of 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.

Syntax:

```
 PROTOCOL APACHE-STATUS [PATH <path>] [<property> <operator> <number>]+
```

path is an optional path to apache status ("/server-status" by default)

property is acronym for child status:

```
 (1) logging (loglimit)
 (2) closing connections (closelimit)
 (3) performing DNS lookups (dnslimit)
 (4) in keepalive with a client (keepalivelimit)
 (5) replying to a client (replylimit)
```

```
(6) receiving a request (requestlimit)
(7) initialising (startlimit)
(8) waiting for incoming connections (waitlimit)
(9) gracefully closing down (gracefullimit)
(10) performing cleanup procedures (cleanuplimit)
```

operator is one of "<", "=", ">".

number is percentile numeric limit.

Each of these limits can be compared against a value relative to the total number of active Apache child processes.

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

Example:

```
if failed port 80 protocol apache-status
       loglimit > 10% or
       dnslimit > 50% or
       waitlimit < 20%
then alert
```

SIP

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

Syntax:

```
PROTOCOL SIP [TARGET valid@uri] [MAXFORWARD n]
```

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)

For example:

```
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
```

RADIUS

Syntax:

```
PROTOCOL RADIUS [SECRET string]
```

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

For 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 pingpong
      then alert
```

MYSQL

Syntax:

```
PROTOCOL MYSQL [USERNAME string PASSWORD string]
```

USERNAME MySQL username (maximum 16 characters).

PASSWORD MySQL password (special characters can be used, but for non-alphanumerics the password has to be quoted).

The credentials are **optional**. If no credentials are set, Monit will perform the test using anonymous login. That may cause authentication error to be logged in your MySQL log for higher MySQL log levels.

If credentials are set, Monit will login and perform MySQL ping. Monit doesn't require any database privileges, it needs just database user, we recommend to create standalone user for Monit testing, for example:

```
CREATE USER 'monit'@'host_from_which_monit_performs_testing' IDENTIFIED BY 'mysecretpas
FLUSH PRIVILEGES;
```

For example:

```
check process mysql with pidfile /var/run/mysqld/mysqld.pid
    start program = "/sbin/start mysql"
    stop program = "/sbin/stop mysql"
    if failed
       port 3306
       protocol mysql username "foo" password "bar"
    then alert
```

or with unix-socket and OS X start/stop commands

```
check process mysql with pidfile /var/run/mysqld/mysqld.pid
    start program = "/usr/local/mysql/support-files/mysql.server start"
    stop program = "/usr/local/mysql/support-files/mysql.server stop"
    if failed
       unixsocket /tmp/mysql.sock
       protocol mysql username "foo" password "bar"
```

```
    then alert
```

WEBSOCKET

Syntax:

```
PROTOCOL WEBSOCKET
        [REQUEST string]
        [HOST string]
        [ORIGIN string]
        [VERSION number]
```

HOST you may specify an alternative Host header

REQUEST you may specify an alternative request, default is "/"

ORIGIN you may specify an alternative origin, default is "http://www.mmonit.com"

VERSION you may specify an alternative version, default is "0"

For example:

```
check host websocket.org with address "echo.websocket.org"
      if failed
         port 80 protocol websocket
         host "echo.websocket.org"
         request "/"
         origin 'http://websocket.com'
         version 13
      then alert
```

CONFIGURATION EXAMPLES

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

```
check process nginx with pidfile /var/run/nginx.pid
```

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

```
check process nginx with pidfile /var/run/nginx.pid
      start program = "/etc/init.d/nginx 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 restart is to first execute the stop-program, then wait (up to 30s) for the process to stop and then execute the start-program and wait (30s) for it to start. The length of start or stop wait can be overridden 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 for 2 cycles then restart
      if failed port 443 for 2 cycles 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 can do the uid and gid switch by themselves, 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 can only switch uid and gid for the 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:

```
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
```

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 which 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
```

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 demonstrate a different way to check a web-server 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 and
         send "GET / HTTP/1.1\r\nHost: www.sol.no\r\n\r\n"
         expect "HTTP/[0-9\.]{3} 200.*"
      then alert
```

Here we ping a remote host to check if it is up and if not, send an alert:

```
check host www.tildeslash.com with address www.tildeslash.com
      if failed ping 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 checksum test should fail, monitoring will be disabled to prevent possibly restarting 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 a document's checksum on a remote server. If the checksum was changed we send an alert:

```
check host mmonit.com with address mmonit.com
      if failed
         port 80 protocol http and
         request "/monit/dist/monit-5.7.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 total memory > 1000.0 MB for 5 cycles then alert
 if total memory > 1500.0 MB for 5 cycles then alert
 if total memory > 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

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 total memory > 1000.0 MB for 5 cycles then alert
  if total memory > 1500.0 MB for 5 cycles then alert
  if total memory > 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
```

Some servers are slow starters, like for example Java based Application Servers. 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 every 2 cycles
      start program = "/etc/init.d/dynamo start"
      stop program  = "/etc/init.d/dynamo stop"
      if failed port 8840 then alert
```

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"
      if failed
         port 9001 protocol tns
      then restart
```

```
      group database
```

This resource checks example will send an alert if CPU usage of the Apache's HTTP daemon and its child processes goes beyond 60% for two cycles. Apache is restarted if the CPU usage is over 80% for five cycles or the memory usage is 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 total cpu > 60% for 2 cycles then alert
      if total cpu > 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
     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 for 5 cycles 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 for 3 cycles 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
```

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).

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

/run directory is available).

/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 saves its state to this file and utilises 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 utilised 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,

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 reinitialise 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 log file. 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.'s excellent fetchmail program. Some portions of this man page also receive inspiration from the same authors.

COPYRIGHT

SEE ALSO

GNU text utilities; md5sum(1); sha1sum(1); openssl(1); glob(7); regex(7)

BACK TO MONIT