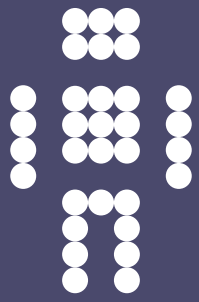
CFEngine



Monitoring with CFEngine

A CFEngine Special Topics Handbook

CFEngine AS

CFEngine fulfils an unusual role as a management system, closing the loop between measurement or monitoring of resources and change management. CFEngine learns the normal behaviour of a system using smart lightweight algorithm, and builds a statistical view of what is normal behaviuour. Policy may then me measured against this normal state to provide *relativistic* reporting of state, and the detection of anomalies.

A significant capability of CFEngine Nova over previous versions of CFEngine, as well as other monitoring software, is the existence of lightweight extensible probes, based on Perl Compatible Regular Expressions. These probes can extract and store data in an efficient and non-intrusive manner. Reports can be integrated into the Nova Knowledge Map and anomalies are automatically detected by CFEngine's self-learning algorithms.

Table of Contents

1 Monitoring introduction

What is monitoring?

The world of IT management is replete with monitoring software. Monitoring is considered to be a major part of management, and it plays a kind of 'feel good' role to engineers even when it often reveals little useful information. Users are often fiercely loyal to certain monitoring solutions, However, most monitoring systems have some key problems:

- Many monitoring systems are so heavy weight that they lead to a system large overhead.
- Scalability of monitoring solutions is often poor, both in terms of system resource consumption and comprehensability of the data collected.

One might argue that these problems can be traced back to the overly ambitious nature of what they try to achieve.

Some common or popular monitoring solutions include:

- HP OpenView
- Nagios
- Munin
- Zenoss
- Ganglia
- collectd

What are the goals of monitoring?

Few montioring systems yield accurate or even very clear results about systems, and yet we feel reassured by moving traces. We monitor systems first and foremost out of a desire for knowledge. If that seems like a trivial statement, you should examine your own motivations carefully – what is it you really want from a monitoring solution? Accurate data, or some basic reassurance?

As scientific instruments, most monitoring software is rather poorly constructed. The devices are rarely calibrated, the results are presented without context, sorted according to arbitrary thresholds, and it is unclear what delay there was between the sampling of the system and the presentation of the data. That makes the data values and the traces almost useless - but not quite. What users really see in monitoring is patterns of change. Monitoring software forms a bridge between actual data about the system and the habits of the human brain.

What does monitoring software do?

Typically, monitoring software samples data from systems through a number of probes and presents the data in some graphical form. A few systems can also perform statistical analysis and even look-ahead forecasting of the data. Much monitoring software is based on the Simple Network Management Protocol (SNMP) which is an active probe regime usually run from a centralized network manager.

The simple fact of the matter is that most monitoring software simple presents a rough visualization of the raw data to users as either a set of alarms (loggable messages) or as a moving time-series, analogous to a hospital vital-signs monitor (EEG or ECG).

If one is cynical, it might be said that some monitoring systems waste users' time by producing moving graphs with a level of detail that is utterly inappropriate. Users then sit transfixed to these moving traces, watching for any insignificant change – and, because there is no context or history to meaure the changes against, every change appears to be interesting.

Monitoring in CFEngine

In CFEngine, there is `cf-monitord`, which runs as a local agent on every computer. This daemon wakes up every couple of minutes and samples data for a number of variables without using the network. The data are then stored in an embedded database on the localhost, using a smart algorithm that prevents the datasize from growing endlessly. CFEngine uses a model of system behaviour based on the findings of research about how computers behave in a network. The model reveals strong weekly patterns in most measurable data, or no pattern whatsoever. This knowledge can be used to compress the data by a large factor and enables `cf-monitord` to carry out a real time statistical analysis of the normal behaviour that is updated over time.

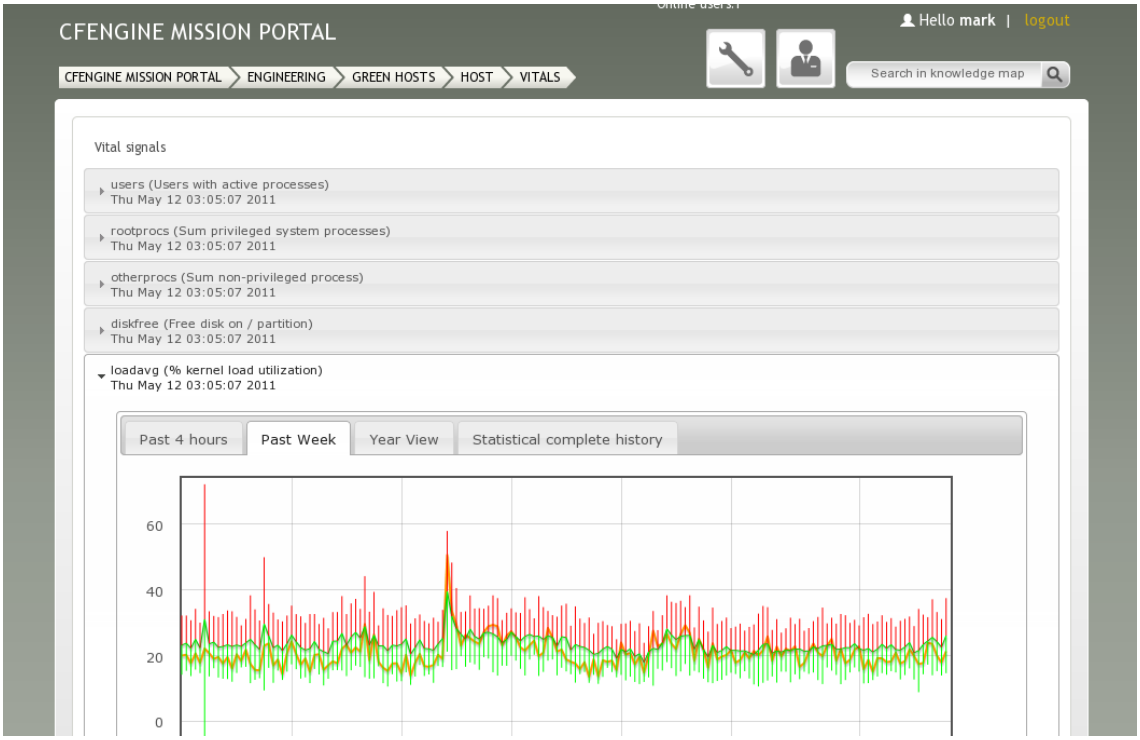
CFEngine was not written to replace other monitoring systems, but to achieve rather concrete goals. In order to achieve these goals, CFEngine does not monitor as often as other systems, and it presents results rather differently.

The goals of monitoring in CFEngine are:

- To not waste users' time with insignificant changes, but provide meaningful updates at a rate that is defensible based on the rate of change of the system.
- To provide meaningful information that is placed in the context of what is normal.
- To reveal trends and patterns at a glance.
- To scale to tens of thousands of hosts without placing a significant burden on the hosts being monitored.
- To be as hands-free in configuration as possible, but allow customization.
- To provide a feedback mechanism for system policy so that systems can respond directly to conditions that are detected.

The information returned by `cf-monitord` comes in a number of forms:

- As visual, plottable graphs.
- As CFEngine classes that are passed to `cf-agent` and may be used to generate alarms or automatic responses.



A graphical rendering of a 100 x load average pattern collected by a host.

Visualization of monitoring in CFEngine

The CFEngine community edition provides limited support for visualization. The `cf-report` command can be used to generate files that can be plotted with other free software. So far this is not well documented, since the process requires special knowledge of some less-well known Open Source tools (see Reference Manual, reporter control promises).

However, in the commercial editions of CFEngine: Nova and Constellation, much effort has been put into making the centralized collection and visualization of these data straightforward and powerful so that all of the learned information about a network may be seen and analysed from a single location.

Any model of fluctuating values is based on the idea that the changing signal has a basic separation of signal and noise. The variability of the signal is generally characterized by a probability distribution which often peaks about the mean value. Some tools and many papers assume that the distribution of fluctuations is Gaussian. This is almost never the case in real computer systems.

CFEngine plots the following values together to provide an interpretive context for the data:

- The last sampled value (‘`value`’). This is the actual ‘current value’. This is the orange line in the figure above.
- The rolling average of the data for each 5 minute interval of the week (‘`av`’). This represents the best estimate of what is normal. This is the green line in the figure above.
- An envelope of on standard deviation above (red vertical bars) and below (green vertical bars) the average to show the envelope of normal ‘variation’ (‘`dev`’).

Note: it is a common misconception that the mean and standard deviation only apply to Gaussian statistical models. This is not true, although it is true that these quantities have a special significance for these distributions. You may think of the rolling mean as a representative average value that represents what is approximately 'normal'. The standard deviation plays the role of an approximate estimate of the uncertainty in the value of the mean. These values should be treated as heuristics, not as absolute truths in any reasonable statistical interpretation of the data.

Standard measured variables

When CFEngine detects an anomaly, it classified the current statistical state of the system into a number of classes.

CFEngine classifies anomalies by whether the currently measured state of the system is higher or lower than the average for the current time of week. The amount of deviation is based on an estimate of the 'standard deviation'. The precise definition of the average and standard deviations is complex, and is discussed in the paper "M. Burgess, Probabilistic anomaly detection in distributed computer networks", (submitted to Science of Computer Programming, and available on the web).

The list of measured attributes is currently fixed to the following:

The first part of the string is from the list:

`users` The number of different users that appear in the process table of the system.

`rootprocs`
The nunmber of current processes started by root/Administrator.

`userprocs`
The number of current processes started by non-privileged users.

`diskfree` The amount of disk free on root file system.

`loadavg` The load average of the system (actually multiplied by 100).

Socket counts of network services distinguish between incoming and outgoing sockets (to a service or from a client).

`netbiosns`
Registers traffic to/from port 137.

`netbiosdgm`
Registers traffic to/from port 138.

`netbiosssn`
Registers traffic to/from port 139.

`irc` Registers traffic to/from port 194.

`CFEngine` Registers traffic to/from port 5308.

`nfsd` Registers traffic to/from port 2049.

`smtp` Registers traffic to/from port 25.

`www` Registers traffic to/from port 80.

`ftp` Registers traffic to/from port 21.

`ssh` Registers traffic to/from port 22.

`wwws` Registers traffic to/from port 443.

If you have tcpdump program installed in a standard location, then the monitor can be confugured to collect data about the network flows to your host.

`icmp` Traffic belonging to the ICMP protocol (ping etc).

`dns` Traffic to port 53, the Domain Name Service (usually a special case of UDP).

`udp` Miscellaneous UDP traffic that is not related to DNS.

`tcpsyn` Registers TCP packets with SYN flag set.

`tcpack` Registers TCP packets with ACK flag set.

`tcpfin` Registers TCP packers with FIN flag set.

`misc` Registers all other packets, not covered above.

Estimate of the level of normality

When `cf-monitord` has accurate knowledge of statistics, it classifies the current state into 3 levels:

`normal` means that the current level is less than one standard deviation above normal.

`dev1` means that the current level is at least one standard deviation about the average.

`dev2` means that the current level is at least two standard deviations about the average.

`anomaly` means that the current level is more than 3 standard deviations above average.

Each of these charaxterizations assumes that there are good data available. The '`cf-monitord`' evaluates its data and decides whether or not the data are too noisy to be really useful. If the data are too noisy but the level *appears* to be more than two standard deviations above aaverage, then the category `microanomaly` is used.

Here are some example classes:

```
userprocs_high_dev2
userprocs_low_dev1
www_in_high_anomaly
smtp_out_high_dev2
```

A complete list of standard metrics Base classes:

```
users
rootprocs
otherprocs
diskfree
loadavg
netbiosns_in
netbiosns_out
netbiosdgm_in
netbiosdgm_out
netbiosssn_in
netbiosssn_out
irc_in
```

```
irc_out
CFEngine_in
CFEngine_out
nfsd_in
nfsd_out
smtp_in
smtp_out
www_in
www_out
ftp_in
ftp_out
ssh_in
ssh_out
wwws_in
wwws_out
icmp_in
icmp_out
udp_in
udp_out
dns_in
dns_out
tcpsyn_in
tcpsyn_out
tcpack_in
tcpack_out
tcpfin_in
tcpfin_out
tcpmisc_in
tcpmisc_out
```

Suffixes:

```
_high_microanomaly
_low_microanomaly

_high_dev1
_low_dev1

_high_dev2
_low_dev2

_high_anomaly
_low_anomaly

_high_ldt
_low_ldt
```

Variables

The `cf-monitord` sets variables which cache the values that were valid at the time of the anomaly's occurrance. These are of the same form as above.

```
value_rootprocs
average_rootprocs
stddev_rootprocs

value_nsfd_in
average_nfsd_in
stddev_nfsd_in
```

The Leap Detection Test buffer is called

```
ldtbuf_users
ldtbuf_otherprocs
```

etc.

Entropy

For network related data, CFEngine evaluates the entropy in the currently measured sample of measurements, with respect to the different IP addresses of the sources. You can use these to predicate the appearance of an anomaly, e.g.

```
entropy_www_in_high
entropy_smtp_in_low
```

For example, if you only want to know when a huge amount of SMTP traffic arrives from a single IP source, you would label your anomaly response:

```
entropy_smtp_in_low.smtp_in_high_anomaly::
```

since the entropy is low when the majority of traffic comes from only a small number of IP addresses (e.g. one). The entropy is maximal when activity comes equally from several different sources.

Persistent classes for alert conditions

Another application for alerts is to pass signals from one invocation of the CFEngine agent to another by persistent, shared memory. For example, suppose a short-lived anomaly event triggers a class that relates to a security alert. The event class might be too short-lived to be followed up by cfagent in full. One could thus set a long term class that would trigger up several follow-up checks. A persistent class could also be used to exclude an operation for an interval of time.

Persistent class memory can be added through a system alert functions to give timer behaviour. For example, consider setting a class that acts like a non-resettable timer. It is defined for exactly 10 minutes before expiring.

```
body classes example
     {
     persist_time => "10";
     }

body classes example
     {
     timer_policy => "reset";
     }
```

2 Monitoring customization

What are measurements?

Measurement promises perform sampling of system variables, and scanning of files and probes, at regular controllable intervals in order to present an efficient overview of actual changes taking place over time.

`measurements` promises

In CFEngine Nova and above, you can extract data from the system in sophisticated ways from files or pipes, using Perl Compatible Regular Expressions to match text. The `cf-monitord` agent is responsible for processing measurement promises.

In this example, we count lines matching a pattern in a file. You might want to scan a log for instances of a particular message and trace this number over time.

Scanning log files for patterns

You will have to scan the log file for each separate summary you want to keep, so you win a lot of efficiency by lumping together mulitple patterns in a longer regular expressions.

Be careful however about the trade-off. Disk access is certainly the most expensive computing resource, but a smart filesystem might do good caching.

Regular expression processing, on the other hand, is CPU expensive, so if you have very long or complex patterns to match, you will begin to eat up CPU time too.

At the end of the day, you should probably do some tests to find a good balance. One goal of CFEngine is to minimally impact your system performance, but it is possible to write promises that have the opposite effect. Check your work!

```
bundle monitor watch
{
measurements:

   "/home/mark/tmp/file"

         handle => "line_counter",
    stream_type => "file",
      data_type => "counter",
    match_value => scan_log("MYLINE.*"),
   history_type => "log",
         action => sample_rate("0");

}
```

```
#############################################################

body match_value scan_log(line)
{
select_line_matching => "$(line)";
track_growing_file => "true";
}

body action sample_rate(x)
{
ifelapsed => "$(x)";
expireafter => "10";
}
```

Scanning syslog for FTP statistics

There are many things that you can set CFEngine at monitoring. For example, CFEngine can automtically collect information about the number of socket-level connections made to the ftp server, but you might want more detailed statistics. For example, you might want to track the volume of data sent and received, or the number of failed logins. Here are a collection of monitoring promises for doing just that.

Note that the ftp logs are maintained by syslog, so it is necessary to match only those lines which correspond to the appropriate service. We also assume that the specific messages are sent to '`/var/log/messages`', while your configuration may specify otherwise. Likewise, your operating systems's version of ftp may issue messages with a slightly different format than ours

```
bundle monitor watch_ftp
{
vars:
   "dir" slist => { "get", "put" };

measurements:

   "/var/log/messages"

        handle => "ftp_bytes_${dir}",
   stream_type => "file",
     data_type => "int",
   match_value => extract_log(".*ftpd\[.*", ".*${dir} .* = (\d+) bytes.*"),
  history_type => "log",
        action => sample_rate("0");

   "/var/log/messages"

        handle => "ftp_failed_login",
```

```
   stream_type => "file",
     data_type => "counter",
   match_value => scan_log(".*ftpd\[.*", ".*FTP LOGIN FAILED.*"),
  history_type => "log",
        action => sample_rate("0");

   "/var/log/messages"

        handle => "ftp_failed_anonymous_login",
   stream_type => "file",
     data_type => "counter",
   match_value => scan_log(".*ftpd\[.*", ".*ANONYMOUS FTP LOGIN REFUSED.*"),
  history_type => "log",
        action => sample_rate("0");

}

#########################################################

body match_value scan_log(line)
{
select_line_matching => "$(line)";
track_growing_file => "true";
}

body match_value extract_log(line, extract)
{
select_line_matching => "$(line)";
extraction_regex => "$(extract)";
track_growing_file => "true";
}

body action sample_rate(x)
{
ifelapsed => "$(x)";
expireafter => "10";
}
```

Scanning DNS logs for query statistics

Another thing you might want to do is monitor the types of queries that your DNS server is being given. One possible reason for this is to test for unusual behavior. For example, suddenly seeing a surge in 'MX' requests might indicate that your system is being targeted by spammers (or that one of your users is sending spam). If you are thinking of converting to IPv6, you might want to compare the number of 'A' requests to 'AAAA' and 'A6' requests to see how effective your IPv6 implementation is.

Because DNS logs are directly maintained by '`bind`' or '`named`' (and do not go through syslog), the parsing can be simpler. However, you *do* need to configure DNS to log query requests to the appropriate log file. In our case, we use '`/var/log/named/queries`'.

```
bundle monitor watch_dns
{
vars:
    "query_type" slist => { "A", "AAAA", "A6", "CNAME", "MX", "NS",
                            "PTR", "SOA", "TXT", "SRV", "ANY" };
measurements:
    "/var/log/named/queries"
        handle => "DNS_$(query_type)_counter",
        stream_type => "file",
        data_type => "counter",
        match_value => scan_log(".* IN $(query_type).*"),
        history_type => "log",
        action => sample_rate("0");
}

#############################################################

body match_value scan_log(line)
{
select_line_matching => "$(line)";
track_growing_file => "true";
}

body action sample_rate(x)
{
ifelapsed => "$(x)";
expireafter => "10";
}
```

Scanning syslog for email statistics

Email is another syslog-based facility that you may want to use CFEngine to monitor. There are a number of volumetric data that are of interest. For example, the number of messages sent and received, the number of messages that have been deferred (a large number might indicate networking problems or spam bounces), and the number of spam messages that have been detected and removed by the assorted spam filters.

The samples below assume that there is a separate logfile for email (called '`/var/log/maillog`') and that a few of the standard sendmail rulesets have been enabled (see '`http://www.sendmail.org/~ca/email/relayingdenied.html`' for details). As with any syslog-generated file, you need to check for the appropriate service, and in this case we are lumping local messages (sent through '`sm-mta`') and remote messages (sent through '`sendmail`') into a single count. Your mileage may of course vary.

If you use one or more sendmail "milters", each of these will also output their own syslog messages, and you may choose to track the volume of rejections on a per-filter basis.

```
bundle monitor watch_email
{
vars:
    "sendmail" string => ".*(sendmail|sm-mta)\[.*";

    "action" slist => { "Sent", "Deferred" };

measurements:

   "/var/log/maillog"

         handle => "spam_rejected",
    stream_type => "file",
      data_type => "counter",
                   # This matches 3 kinds of rulesets: check_mail,
                   # check_rcpt, and check_relay
    match_value => scan_log("$(sendmail)ruleset=check_(mail|rcpt|relay).*"),
   history_type => "log",
         action => sample_rate("0");

   "/var/log/maillog"

         handle => canonify("mail_$(action)",
    stream_type => "file",
      data_type => "counter",
    match_value => scan_log("$(sendmail)stat=$(action) .*"),
   history_type => "log",
         action => sample_rate("0");

}

#############################################################

body match_value scan_log(line)
{
select_line_matching => "$(line)";
track_growing_file => "true";
}

body action sample_rate(x)
{
ifelapsed => "$(x)";
expireafter => "10";
```

```
}
```

Scanning syslog for email milter failures

Milters are relatively new in sendmail, and some have problems. You can also use monitoring to detect certain types of failure modes. For example, if a milter is running (that is, there is a process present) but it does not respond correctly, sendmail will log an entry like this in syslog (where 'xyzzy' is the name of the milter in question):

```
Milter (xyzzy): to error state
```

A small number of these messages is no big deal, since sometimes the milter has temporary problems or simply encounters an email message that it finds confounding. But a larger value of these messages usually indicates that the milter is in a broken state, and should be restarted.

You can use '`cf-monitord`' to check for the number of these kinds of messages, and use the soft classes that it creates to change how '`cf-agent`' operates. For example, here we will restart any milter which is showing a high number of failure-mode messages:

```
bundle monitor watch_milter
{
vars:
   "milter" slist => { "dcc", "bogom", "greylist" };

measurements:

   "/var/log/maillog"

         handle => "${milter}_errors",
    stream_type => "file",
      data_type => "counter",
    match_value => scan_log(".*Milter (${milter}): to error state"),
   history_type => "log",
         action => sample_rate("0");
}

bundle agent fix_milter
{
vars:
    "m[dcc]" string      => "/var/dcc/libexec/start-dccm";
    "m[bogom]" string    => "/usr/local/etc/rc.d/milter-bogom.sh restart";
    "m[greylist]" string => "/usr/local/etc/rc.d/milter-greylist restart";

commands:
    "$(m[$(watch_milter.milter)])"
        ifvarclass => "$(watch_milter.milter)_high";
}
```

Scanning syslog for breakin attempts

A lot of script-kiddies will probe your site for vulnerabilities, using dictionaries of account/password combinations, looking for unguarded accounts or accouts with default passwords. Most of these scans are harmless, because a well-maintained site will not use the default passwords that these hackers seek to exploit.

However, knowing that you are being scanned is a good thing, and CFEngine can help you find that out. Because 'sshd' logs it's message through 'syslog', we again need to filter lines based on the service name. On our system, authorization messages are routed to '/var/log/auth.log', and we would monitor it like this:

```
bundle monitor watch_breakin_attempts
{
measurements:
    "/var/log/auth.log"
         # This is likely what you'll see when a script kiddie probes
         # your system

         handle => "ssh_username_probe",
    stream_type => "file",
      data_type => "counter",
    match_value => scan_log(".*sshd\[.*Invalid user.*"),
   history_type => "log",
         action => sample_rate("0");

    "/var/log/auth.log"
         # As scary as this looks, it may just be because someone's DNS
         # records are misconfigured - but you should double check!

         handle => "ssh_reverse_map_problem",
    stream_type => "file",
      data_type => "counter",
    match_value => scan_log(".*sshd\[.*POSSIBLE BREAK-IN ATTEMPT!.*"),
   history_type => "log",
         action => sample_rate("0");

    "/var/log/auth.log"
         # Someone is trying to log in to an account that is locked
         # out in the sshd config file

         handle => "ssh_denygroups",
    stream_type => "file",
      data_type => "counter",
    match_value => scan_log(".*sshd\[.*group is listed in DenyGroups.*"),
   history_type => "log",
         action => sample_rate("0");
```

```
    "/var/log/auth.log"
         # This is more a configuration error in /etc/passwd than a
         # breakin attempt...

         handle => "ssh_no_shell",
    stream_type => "file",
      data_type => "counter",
    match_value => scan_log(".*sshd\[.*because shell \S+ does not exist.*"),
   history_type => "log",
         action => sample_rate("0");

    "/var/log/auth.log"
         # These errors usually indicate a problem authenticating to your
         # IMAP or POP3 server

         handle => "ssh_pam_error",
    stream_type => "file",
      data_type => "counter",
    match_value => scan_log(".*sshd\[.*error: PAM: authentication error.*"),
   history_type => "log",
         action => sample_rate("0");

    "/var/log/auth.log"
         # These errors usually indicate that you haven't rebuilt your
         # database after changing /etc/login.conf - maybe you should
         # include a rule to do this command: cap_mkdb /etc/login.conf

         handle => "ssh_pam_error",
    stream_type => "file",
      data_type => "counter",
    match_value => scan_log(".*sshd\[.*login_getclass: unknown class.*"),
   history_type => "log",
         action => sample_rate("0");
}
```

See the CFEngine Nova documentation for more possibilities of measurement promises.

Threshold monitoring

```
vars:

 "probes" slist => { "www", "smtp", "ssh" };

classes:

 "$(probes)_threshold" expression => isgreaterthan("$(mon.$(probes))","50");
```

```
reports:

  "Help $(probes)!" ifvarclass => "$(probes)_threshold";
```

Summary Monitoring

There are endless possibilities for monitoring with CFEngine Nova. This document has suggested a few.