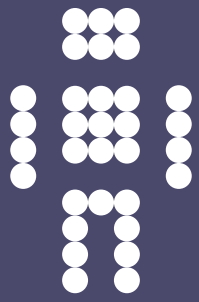
CFEngine



Monitoring and Reporting

A CFEngine Special Topics Handbook

CFEngine AS

A significant capability of CFEngine Nova over previous versions of CFEngine is the existence of automated system reporting. CFEngine collects history, state and change data about computers and ties them together.

The CFEngine strategy is to replace conventional CMDBs with a more scalable and flexible approach to information mining over the coming years. Commercial versions of CFEngine are designed to bring state of the art methods to the problem of information management for IT operations.

Users of CFEngine's Community Edition can use in-built logging and reporting functions to simulate some aspects of these reports, by applying simple principles with work and ingenuity.

Table of Contents

What are monitoring and reporting?

Monitoring is the sampling of system variables at regular intervals in order to present an overview of actual changes taking place over time. Monitoring data are often presented as extensive views of moving-line time series. Monitoring has the ability to detect anomalous behaviour by comparing past and present.

The term *reporting* is usually taken to mean the creation of short summaries of specific system properties suitable for management. System reports describe both promises about the system, such as compliance, discovered changes and faults.

The challenge of both these activities is to compare *intended* or *promised*, behaviour with the *actual* observed behaviour of the system.

Should monitoring and configuration be separate?

The traditional view of IT operations is that configuration, monitoring and reporting are three different things that should not be joined. Traditionally, all three have been independent centralized processes. This view has emerged historically, but it has a major problem. Humans are needed to glue these parts back together.

Monitoring as an independent activity is inherently non-scalable. When numbers of hosts grow beyond a few thousands, centralized monitoring schemes fail to manage the information. Tying configuration (and therefore repair) to monitoring at the host level is essential for the effective management of large and distributed data facilities. CFEngine foresaw this need in 1998, with its Computer Immunology initiative, and continues to develop this strategy.

CFEngine's approach is to focus on scalability. The commercial editions of CFEngine provide what meaningful information they can in a manner that can be scaled to tens of thousands of machines.

Reporting in CFEngine

If you have regular reporting needs, we recommend using our commercially supported version of CFEngine (CFEngine Nova or above), as you will save considerable time and resources in programming, and you will have access to the latest developments through the software subscription.

No promises made in CFEngine imply automatic aggregation of data to a central location. In commercial CFEngine versions, e.g. CFEngine Nova, an optimized aggregation of standardized reports is provided, but the ultimate decision to aggregate must be yours.

Monitoring and reporting capabilities in CFEngine depend on the software version include:

- **Community Edition:** Basic output to file or logs may be customized on a per-promise basis. Users can design their own log and report formats, but data processing and extraction from CFEngine's embedded databases must be scripted by the user.
- **Nova:** In addition to community features, Nova provides automated extraction of data from CFEngine's self-learning agents, and the generation of a standard set of reports

in text, HTML or XML formats. Nova summarizes distributed data and provides simple compression and aggregation of these summaries. Finally summaries are tied into a knowledge map or semantic index for browsing by IT operations. Command line tools in `cf-report` are also available for Nova users to browse network-wide data.

- **Constellation:** In addition to Nova features, Constellation performs additional data extraction from the collected reports. It analyses correlations and provides reverse look up of system attributes based on searchable expressions. At this level, CFEngine exceeds other industry CMDB solutions in both reporting and configuration.

Standard reports in CFEngine

The following list of reports are only available in full in commercial editions of CFEngine. Some sample reports are provided in the Community Edition.

Available patches report
: Patches already installed on system if available.

Classes report
: User defined classes observed on the system – inventory data.

Compliance report
: Total summary of host compliance, all promises aggregated over time.

File_changes report
: Latest observed changes to system files with time discovered.

File_diffs report
: Latest observed differences to system files, in a simple diff format.

Hashes report
: File hash values measured (change detection).

Installed patches report
: Patches not yet installed, but published by vendor if available.

Installed software report
: Software already installed on system if available.

Lastseen report
: Time and frequency of communications with peers, host reliability.

Micro-audit report
: Generated by CFEngine self-auditing. This report is not aggregated.

Monitor summary report
: Pseudo-real-time measurement of time series data.

Performance report
: Time cost of verifying system promises.

Promise report
: Per-promise average compliance report over time.

Promises not kept report
: Promises that were recently un-kept.

Promises repaired report
: Promises that were recently kept by repairing system state.

Setuid report
: Known setuid programs found on system.

Variables report
: Current variable values expanded on different hosts.

CFEngine output levels

CFEngine's default behaviour is to report to the console (known as standard output). It's default behaviour is to report nothing except errors that are judged to be of a critical nature.

By using CFEngine with the inform flag:

```
# cf-agent -I
# cf-agent --inform
```

you can alter the default to report on action items (actual changes) and warnings.

By using CFEngine with the verbose flag:

```
# cf-agent -v
# cf-agent --verbose
```

you can alter the default to report all of its thought-processes. You should not interpret a message that only appears in CFEngine's verbose mode as an actual error, only as information that might be relevant to decisions being made by the agent.

Creating custom reports – all versions

CFEngine allows you to use `reports` promises to make reports of your own. A simple example of this is shown below.

```
body common control
{
bundlesequence => { "test" };
}

#

bundle agent test
{
reports:

  cfengine_3::

   "$(sys.date),This is a report"
     report_to_file => "/tmp/test_log";
}
```

We can apply this idea to make more useful custom reports. In this example, the agent tests for certain software package and creates a simple HTML file of existing software.

```
body common control
{
bundlesequence => { "test" };
}

#

bundle agent test
{
vars:

 "software" slist => { "gpg", "zip", "rsync" };

classes:

 "no_report"        expression => fileexists("/tmp/report.html");
 "have_$(software)" expression => fileexists("/usr/bin/$(software)");

reports:

  no_report::

      "
      <html>
      Name of this host is: $(sys.host)<br>
      Type of this host is: $(sys.os)<br>
      "

         report_to_file => "/tmp/report.html";

      #

      "
      Host has software $(software)<br>
      "

        ifvarclass     => "have_$(software)",
        report_to_file => "/tmp/report.html";

      #

      "
      </html>
      "
```

```
        report_to_file => "/tmp/report.html";

}
```

The outcome of this promise is a file called '`/tmp/report.html`' containing output like this:

```
<html>
Name of this host is: atlas<br>
Type of this host is: linux<br>

Host has software gpg<br>

Host has software zip<br>

Host has software rsync<br>

</html>
```

The mechanism shown above, can clearly be used to create a wide variety of report formats, but it requires a lot of coding and maintenance by the user.

> CFEngine Nova simplifies this kind of report generation by enabling and updating many out-of-the-box reports directly from the `cf-report` agent.

Including data in reports

CFEngine generates information internally that you might want to use in reports. For example, the agent `cf-agent` interfaces with the local light-weight monitoring agent `cf-monitord` so that system state can be reported simply:

```
body common control

{
bundlesequence  => { "report" };
}

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

bundle agent report

{
reports:

  linux::

   "/etc/passwd except $(const.n)"

     showstate => { "otherprocs", "rootprocs" };
```

```
}
```

A corollary to this is that you can get CFEngine to report system anomalies.

```
reports:

 rootprocs_high_dev2::

   "RootProc anomaly high 2 dev on $(mon.host) at approx $(mon.env_time)
    measured value $(mon.value_rootprocs)
    average $(mon.average_rootprocs) pm $(mon.stddev_rootprocs)"

      showstate => { "rootprocs" };

 entropy_www_in_high&anomaly_hosts.www_in_high_anomaly::

   "High entropy incoming www anomaly on $(mon.host) at $(mon.env_time)
    measured value $(mon.value_www_in)
    average $(mon.average_www_in) pm $(mon.stddev_www_in)"

      showstate => { "incoming.www" };
```

This produces standard output of the form:

```
R: State of otherprocs peaked at Tue Dec  1 12:12:21 2009

R: The peak measured state was q = 98:
R: Frequency: [kjournald]      |**      (2/98)
R: Frequency: [pdflush]        |**      (2/98)
R: Frequency: /var/cfengine/bin/cf-execd|**     (2/98)
R: Frequency: COMMAND          |*       (1/98)
R: Frequency: init [5]         |*       (1/98)
R: Frequency: [kthreadd]       |*       (1/98)
R: Frequency: [migration/0]    |*       (1/98)
R: Frequency: [ksoftirqd/0]    |*       (1/98)
R: Frequency: [events/0]       |*       (1/98)
R: Frequency: [khelper]        |*       (1/98)
R: Frequency: [kintegrityd/0]  |*       (1/98)
```

Finally, you can quote lines from files in your data for convenience.

```
body common control

{
bundlesequence  => { "report" };
}
```

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

bundle agent report

{
reports:

  linux::

   "/etc/passwd except $(const.n)"

     printfile => pr("/etc/passwd","5");

}

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

body printfile pr(file,lines)

{
file_to_print => "$(file)";
number_of_lines => "$(lines)";
}
```

This produces output of the form

```
R: /etc/passwd except
R: at:x:25:25:Batch jobs daemon:/var/spool/atjobs:/bin/bash
R: avahi:x:103:105:User for Avahi:/var/run/avahi-daemon:/bin/false
R: beagleindex:x:104:106:User for Beagle indexing:/var/cache/beagle:/bin/bash
R: bin:x:1:1:bin:/bin:/bin/bash
R: daemon:x:2:2:Daemon:/sbin:/bin/bash
```

Creating custom logs

Logs can be attached to any promise. In this example, an executed shell command logs a message to the standard output. CFEngine recognizes the `stdout` filename for Standard Output, in the Unix/C standard manner.

```
bundle agent test
{
commands:

  "/tmp/myjob",

     action => logme("executor");
```

```
}

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

body action logme(x)
{
log_repaired => "stdout";
logstring => " -> Started the $(x) (success)";
}
```

In this next example, a file creation promise logs different outcomes (success or failure) to different log files.

```
body common control
{
bundlesequence => { "test" };
}

bundle agent test
{
vars:

  "software" slist => { "/root/xyz", "/tmp/xyz" };

files:

  "$(software)"

    create => "true",
     action => logme("$(software)");

}

#

body action logme(x)
{
log_kept => "/tmp/private_keptlog.log";
log_failed => "/tmp/private_faillog.log";
log_repaired => "/tmp/private_replog.log";
log_string => "$(sys.date) $(x) promise status";
}
```

This generates three different logs with outputs in of the form:

```
atlas$ more /tmp/private_keptlog.log
Sun Dec  6 11:58:16 2009 /tmp/xyz promise status
Sun Dec  6 11:58:43 2009 /tmp/xyz promise status
```

Redirecting output to logs

CFEngine interfaces with the system logging tools in different ways. Syslog is the default log for Unix-like systems, while the event logger is the default on Windows. You may choose to copy a fixed level of CFEngine's standard screen messaging to the system logger on a per-promise basis.

```
body common control
{
bundlesequence => { "one" };
}


bundle agent one
{
files:

  "/tmp/xyz"

       create => "true",
       action => log;
}

body action log
{
log_level => "inform";
}
```

Change auditing - the all seeing eye

Total auditing of a system is a surprisingly difficult thing to do, and it is extremely resource intensive. The followers of an audit trail are often paranoid by nature and are seldom satisfied with the level of detail they find. However, the times we really need an audit are rare, but the cost is ever present. The price of certainty is high.

Spend a moment considering this: if you want to describe every change of state that happens on a computer, then you need to remember old state and compare it to new state. Then you have to record the differences. So you need more than the entire size of your computer's normal resources to do this. Your storage efficiency will always be less than 50% and your processing efficiency will be less than 50% on every audited item. Is this worth the effort? Perhaps your resources would be better spent keeping targeted backups and simply rebuilding contaminated systems.

Switch on auditing like this:

```
body agent control
{
auditing  => "true";
}
```

If you decide to go for full auditing, CFEngine will not collect and centralize the reports as they will be too large for this to be a scalable operation. Still, you can view them in a web browser on the local host, or copy them manually to a suitable location.

Cheaper options - tripwires

Doing a change detection scan is a convergent process, but it can still detect changes and present the data in a compressed format that is often more convenient than auditing. The result is less precise, but there is a trade-off between precision and cost.

To make a change tripwire, you use a '`files`' promise, something like this:

```
body common control
{
bundlesequence  => { "testbundle"  };
}
#

bundle agent testbundle

{
files:

  "/home/mark/tmp" -> "me"
       changes      => scan_files,
       depth_search => recurse("inf");
}

# library code ...

body changes scan_files
{
report_changes => "all";
update_hashes  => "true";
}

body depth_search recurse(d)
{
depth        => "$(d)";
}
```

In CFEngine Nova, reports of the following form are generated when these promises are kept by the agent:

```
Change detected          File change
Sat Dec 5 18:27:44 2009  group for /tmp/testfile changed 100 -> 0
Sat Dec 5 18:27:44 2009  /tmp/testfile
Sat Dec 5 18:20:45 2009  /tmp/testfile
```

These reports are generated automatically in CFEngine Nova, and are integrated into the web browsable knowledge map. Community edition users have to extract the data and create these themselves.

Commercial edition measurements promises

In commercial versions of CFEngine, you can extract data from the system in more 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.

```
bundle monitor watch
{
measurements:

   "/tmp/file"

         handle => "line_counter",
    stream_type => "file",
      data_type => "counter",
    match_value => scanlines("MYLINE.*"),
   history_type => "log";

}

#

body match_value scanlines(x)
{
select_line_matching => "^$(x)$";
}
```

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

Hub Reporting

In the commercial editions of CFEngine much more extensive and searchable reporting is available.

Mission Portal access to the hub

The preferred approach to querying information on a hub is to use the web interface in the Mission Portal. This gives the greatest flexibility in both search and presentation of data. Given the extensiveness of the Mission Portal user interface, the details are covered in a separate document.

Command line access to the hub

Users with login access to the hub can also use the command line tool `cf-report` to extract a limited view of the data. Currently supported reports include:

`compliance`
: The percentage total compliance log for all hosts.

`dead-clients`
: Shows a list of client hosts that have not made incoming requests within the standard time horizon (default 15 minutes).

`file_changes`
: The change log

`file_diffs`
: The change details for text files.

`last-seen`
: Show the last time hosts connnected to the hub

`promises` Compliance by promise, labelled by promise-handle.

`setuid` The list of setuid/setgid root files detected on the system.

`software` The installed software base of the system.

`summary` A summary of how many hosts are compliant within a given set of search parameters.

`vars` The values of variables set on hosts.

Some special command line options are supported in the commercial versions.

'`--query-hub`'
: or '`-q`' Query hub database interactively. This option is the entry point for querying the hub data with `cf-report`, and must always be specified.

'`--show name`'
: Select the name of the report from the above list.

'`--promise-handle`'
: or '`-p regex`'. For promise compliance report, this defines a regular expression to search for promises of a specific name. Specify a promise-handle to look up

'`--hostkey`'
: or '`-k hashkey`'. Specify a particular host to query for data, using the unique host-key.

‘`--class-regex`’
or ‘`-c regex`’ - Specify a class regular expression to search for

‘`--filter`’
or ‘`-F regex`’ - Specify a name regular expression for filtering results

Example command hub searches

If only a host-key is specified, CFEngine returns with the last known location and identity of the host. (Note that, in the following examples, the SHA keys are reduced for readability).

```
host# cf-report -q --hostkey  SHA=bd6dfcc2...
 -> Hostname: hub.test.cfengine.com
 -> Recent IP Addresses: 10.0.0.29
```

To dump all values from all hosts:

```
cf-report -q --show promises
cf-report --query-hub --show promises
```

You can select a single host for a particular report:

```
cf-report -q --hostkey SHA=c40fb732c6e5... --show vars
```

Or you can select a CFEngine class of hosts that will be selected to report

```
cf-report -q --show summary --class-regex linux
cf-report -q --show summary --class-regex SuSE
cf-report -q --show summary --class-regex NewYork
```

Here are some examples using filters to 'grep' out certain items:

```
cf-report -q --hostkey SHA=c40fb732c6...  --show vars --filter date

cf-report -q --filter "mail.*" --hostkey  SHA=bd6dfccb1a... --show setuid

cf-report -q  --show promises -p knowledge_files_db_stamp
```