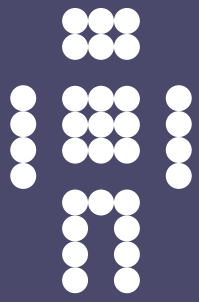
CFEngine



Distributed Scheduling and Workflows

A CFEngine Special Topics Handbook

CFEngine AS

Distributed scheduling is about tying together jobs to create a workflow across multiple machines. It introduces a level of fragility into system automation. Using CFEngine promises, we can create self-healing workflows, but we recommend minimizing dependencies. This document shows how to build workflows using CFEngine primitives.

Table of Contents

What is distributed scheduling?

Scheduling refers to the execution of non-interactive processes or tasks (usually called 'jobs') at designated times and places around a network of computers (see the Special Topics Guide on Scheduling). Distributed Scheduling refers to the chaining of different jobs into a coordinated workflow that spans several computers. For example, you schedule a processing job on `machine1` and `machine2`, and when these are finished you need to schedule a job on `machine3`. This is distributed scheduling.

Coordinating dispatch

Dispatch is the term used for starting actually the execution of a job that has been scheduled. There are two ways to achieve distributed job scheduling:

- Centralized dispatch of jobs.
- Peer to peer signalling with local dispatch of jobs.

There are pros and cons to centralization. Centralization makes consistency easy to determine, but it creates bottlenecks in processing and allows one machine to see all information. Decentralization provides an automatic and natural load-balancing of job dispatch, and it allows machines to reveal information on a 'need to know' basis.

CFEngine is a naturally decentralized system, and only policy definition is usually centralized, but you can set up practically any architecture you like, in a secure fashion.

Job scheduling and periodic maintenance

You promise to execute tasks or keep promises at distributed places and times:

- You tell CFEngine *what* and *how* with the details of a promise.
- You tell CFEngine *where* and *when* promises should be kept, using *classes*.

CFEngine is designed principally to maintain desired state on a continuous basis. There are three cases for job scheduling:

- Unique jobs run once and only once.
- Standard jobs run sporadically on demand.
- Standard jobs run on a regular schedule.

This list transfers to workflow processes too. If one job needs to follow after another (because it depends on it for something), we can ask if this workflow is a standard and regular occurrence, or a one-off phenomenon.

One-off workflows

In CFEngine, you code a one-off workflow by specifying the space-time coordinates of the event that starts it. For example, if you want a job to be run a 16:45 on Monday 24th January 2012, you would make a class corresponding to this time, and place the promise of a job (or jobs) in this class. Let's look at some examples of this, in which host1 executes a command called '`my_job`', and host2 follows up with a bundle of promises afterwards.

The simplest case is to schedule the exact times.

```
bundle agent workflow_one
{
methods:

  Host2.Day24.January.Year2012.Hr16.Min50_55::

    "any" usebundle => do_my_job_bundle;

commands:

  Host1.Day24.January.Year2012.Hr16.Min45_50::

    "/usr/local/bin/my_job";

}
```

Host1 runs its task at 16:45, and Host2 excutes its part in the workflow five minutes later. The advantage of this approach is that no direct communication is required between Host1 and Host2. The disadvantage is that you, as the orchestrator, have to guess how long the jobs will take. Moreover Host2 doesn't know for certain whether host1 succeeded in carrying out its job, so it might be a fruitless act.

We can change this by signalling between the processes. Whether not you consider this an improvement or not depends on what you value highest: avoidance of communication or certainty of outcome. In this version, we increase the certainty of control by asking the predecessor or upstream host for confirmation of success if the job was carried out.

```
bundle agent workflow_one
{
classes:

  Host2::

     "succeeded"     expression => remoteclassesmatching
                                   (
                                   "did.*",    # get classes matching
                                   "Host1",    # from this server
                                   "no",       # encrypt comms?
                                   "hostX"     # prefix
                                   );
methods:

  Host2.hostX_did_my_job

    "any" usebundle => do_my_job_bundle;

commands:

  Host1.Day24.January.Year2012.Hr16.Min45_50::
```

```
    "/usr/local/bin/my_job",
        classes => state_repaired("did_my_job");
}
```

In this example, the methods promise runs on Host2 and the commands promise runs one Host1 as before. Now, host 1 sets a signal class '`did_my_job`' when it carries out the job, and Host2 collects it by contacting the `cf-serverd` on Host1. Assuming that Host1 has agreed to let Host2 know this information, by granting access to it, Host2 can inherit this class, with a prefix of its own choosing. Thus is transforms the class '`did_my_job`' on Host1 into '`hostX_did_my_job`' on Host2.

The advantage of this method is that the second job will only be started if the first completed, and we don't have to know how long the job took. The disadvantage of this is that we have to exchange some network information, and this has a small network cost, and requires some extra configuration on the server side to grant access to this context information:

```
bundle server access_rules
{
access:

  "did_my_job"

  resource_type => "context",
        admit   => { "Host2" };
}
```

Regular workflows

To make a job happen at a specific time, we used a very specific time classifier '`Day24.January.Year2012.Hr16.Min45_50`'. If we now want to make this workflow into a regular occurrence, repeating at some interval we have two options:

- We repeat this at the same time each week, day, hour, etc.
- We don't care about the precise time, we only care about the interval between executions.

The checking of promises in CFEngine is controlled by *classes* and by *ifelapsed locks*, which may be used for these two cases respectively. If nothing else is specified, CFEngine runs every 5 minutes and reconsiders the state of all its active promises. To be specific about the time, we just alter which promises are active at different times. Classes (as used already) allow us to anchor a promise to a particular region of time and space. Locks, on the other hand, allow us to say that a promise will only be rechecked if a certain time has elapsed since the last time.

So, to make a promise repeat, we simply have to be less specific about the time. Let us make the promise on Host1 apply every day between 16:00:00 (4 pm) and 16:59:59, and add an ifelapsed lock saying that we do not want to consider rechecking more often tha once every 100 minutes (more than 1 hour). Now we have a workflow process that starts at 16:00 hours each day and runs only once each day.

```
bundle agent workflow_one
{
classes:
```

```
  Host2::

     "succeeded"     expression => remoteclassesmatching(
                                                         "did.*",
                                                         "Host1",
                                                         "no",
                                                         "hostX"
                                                         );
methods:

  Host2.hostX_did_my_job

    "any" usebundle => do_my_job_bundle;

commands:

  Host1.Hr16::

    "/usr/local/bin/my_job",
         action => if_elapsed("100"),
        classes => state_repaired("did_my_job");
```

Fancy distributed encapsulation

We could try to be fancy about distributed scheduling, packaging it into a reusable structure. This may or may not be a good idea, depending on your aesthetics. The following example, from the community unit tests, shows how we might proceed.

```
body common control

{
bundlesequence  => { job_chain("Hr16.Min10_15") };
}

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

bundle common g
{
vars:

 # Define the name of the signal passed between hosts

 "signal"   string => "pack_a_name";
}
```

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

bundle agent job_chain(time)
{
vars:

    # Define the names of the two parties

    "client"   string => "downstream.exampe.org";
    "server"   string => "upstream.example.org";

classes:

    # derive some classes from the names defined in variables

    "client_primed" expression => classmatch(canonify("$(client)")),
                    ifvarclass => "$(time)";

    "server_primed" expression => classmatch(canonify("$(server)")),
                    ifvarclass => "$(time)";

 client_primed::

     "succeeded"     expression => remoteclassesmatching(
                                                   "$(g.signal)",
                                                   "$(server)",
                                                   "yes",
                                                   "hostX"
                                                   );
methods:

 client_primed::

    "downstream" usebundle => do_job("Starting local follow-up job"),
                    action => if_elapsed("5"),
                ifvarclass => "hostX_$(g.signal)";

 server_primed::

    "upstream"   usebundle => do_job("Starting remote job"),
                    action => if_elapsed("5"),
                   classes => state_repaired("$(g.signal)");

reports:

  !succeeded::
```

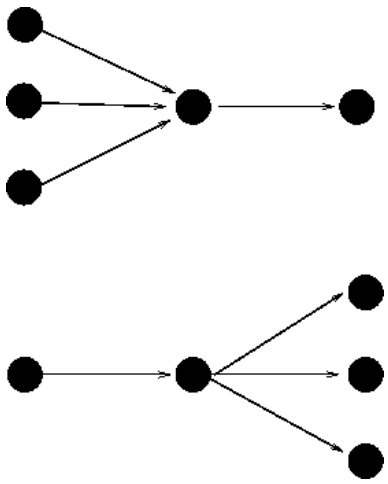
```
    "Server communication failed",

                ifvarclass => "$(time)";
}

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

bundle agent do_job(job)
{
commands:

   # do whatever...

   "/bin/echo $(job)";
}

#############################################################
# Server config
#############################################################

body server control
{
allowconnects         => { "127.0.0.1" , "::1" };
allowallconnects      => { "127.0.0.1" , "::1" };
trustkeysfrom         => { "127.0.0.1" , "::1" };
allowusers            => { "mark" };
}

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

bundle server access_rules()
{
access:

  "$(g.signal)"

  resource_type => "context",
        admit   => { "127.0.0.1" };
}
```

More links in the chain

In the examples above, we only had two hosts cooperating about jobs. In general, it is not a good idea to link together many different hosts unless there is a good reason for doing so. In HPC or Grid environments, where distributed jobs are more common and results are combined

from many sub-tasks, one typically uses some more specialized middleware to accomplish this kind of cooperation. Such software makes compromises of its own, but is generally better suited to the specialized task for which it was written than a tool like CFEngine (whose main design criteria are to be secure and generic).

Nevertheless, there are some tricks left in CFEngine for distributed scheduling if we want to trigger a number of follow-ups from a single job, or aggregate a number of jobs to drive a single follow-up (see figure).



Aggregation of multiple jobs

When aggregating jobs, we must combine their exit status using AND or OR. The most common case it that we require all the prerequisites in place in order to generate the final result, i.e. trigger the followup only if all of the prerequisites succeeded.

```
bundle agent workflow_one
{
vars:

  "n" slist => { "2", "3", "4" };

classes:

     "succeeded$(n)"     expression => remoteclassesmatching(
                                                "did.*",
                                                "Host$(n)",
                                                "no",
                                                "hostX"
                                                ),
                         ifvarclass => "Host$(n)";
methods:

  Host2.Host3.Host4.hostX_did_my_job
```

```
    "any" usebundle => do_my_job_bundle;

commands:

  Host1.Hr16::

    "/usr/local/bin/my_job",
         action => if_elapsed("100"),
        classes => state_repaired("did_my_job");
```

This example shows an all-or-nothing result. The follow-up job will only be executed if all three jobs finish within the same 5 minute time-frame. There is no error handling or recovery except to schedule the whole thing again.

Triggering from one or more predecessors, i.e. combining with OR, looks similar, we just have to change the class expression:

```
...

methods:

  (Host2|Host3|Host4).hostX_did_my_job

    "any" usebundle => do_my_job_bundle;

...
```

Triggering multiple follow-ups

The converse scenario is to trigger a number of jobs from a single pre-requisite. This is simply a case of listing the jobs under the trigger classes.

```
bundle agent workflow_one
{
classes:

  Host2::

     "succeeded"     expression => remoteclassesmatching(
                                                         "did.*",
                                                         "Host1",
                                                         "no",
                                                         "hostX"
                                                         );
methods:

  Host2.hostX_did_my_job

    "any" usebundle => do_my_job_bundle1;
    "any" usebundle => do_my_job_bundle2;
    "any" usebundle => do_my_job_bundle3;
```

```
commands:

  Host1.Hr16::

    "/usr/local/bin/my_job",
         action => if_elapsed("100"),
        classes => state_repaired("did_my_job");
```

Self-healing workflows

To apply CFEngine's self-healing concepts to workflow scheduling, we can imagine the concept of a convergent workflow, i.e. one that, if we repeat everything a sufficient number of times, will eventually lead to the result. The outcome of the chained sequence of jobs must have an outcome that is repeatably achievable and which will eventually be achieved if we try a sufficient number of times. Using CFEngine this is a natural outcome – however, most system designers do not think in terms of repeatable sustainable outcomes and fault-tolerance.

Beware however, one-off jobs *cannot* be made convergent, because they only have a single chance to succeed. It is a question of business process design whether you design workflows to be sustainable and repeatable, or whether you trust the outcome of a single shot process. Using the persistent classes in CFEngine together with the if-elapsed locks to send signals between hosts, it is simple and automatic to make convergent self-healing workflows.

Long workflow chains

Long workflow chains are those which involve more than one trigger. These can be created by repeating the pattern above several times. Note however, that each link in the chain introduces a new level of uncertainty and potential failure. In general, we would not recommend creating workflows with long chains.

Summary of Distributed Scheduling

Distributed scheduling is about tying together jobs to create a workflow across multiple machines. It introduces a level of fragility into system automation. Using CFEngine promises, we can create self-healing workflows, but we recommend minimizing dependencies. This document shows how to build workflows using CFEngine primitives.