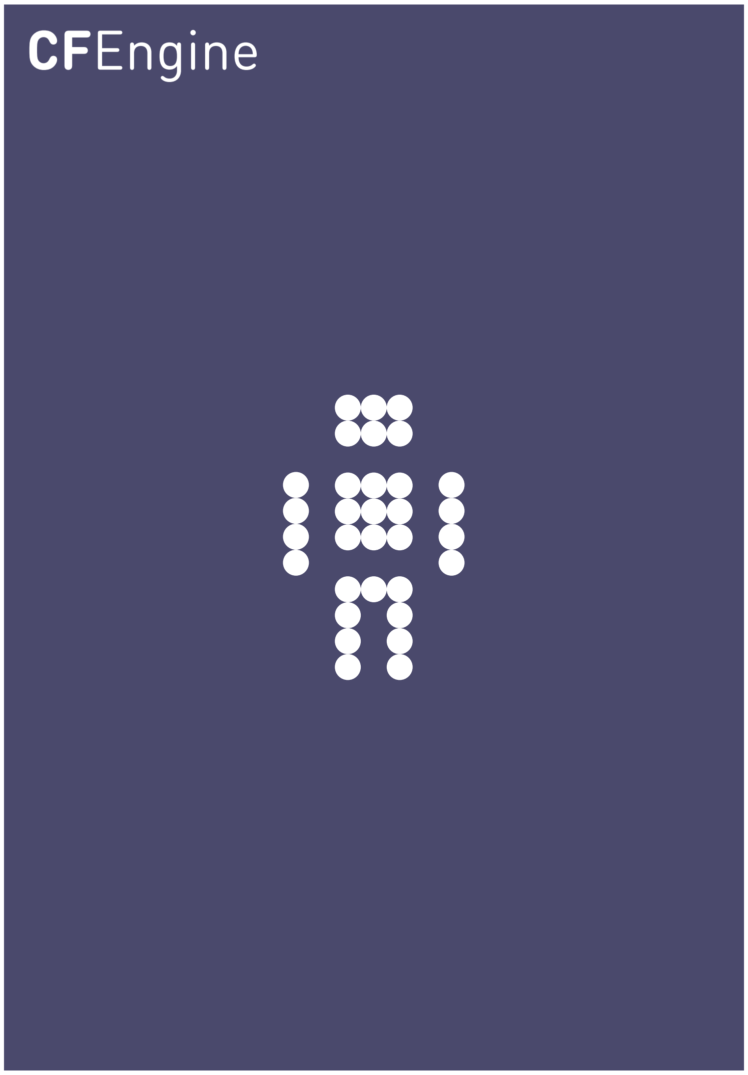


Scale and Scalability (draft)

A CFEngine Special Topics Handbook

CFEngine AS

How large a system can CFEngine manage before special measures are required to make it work? CFEngine is a flexible system with no fixed architecture, that can be adapted to service any number of machines, by adjusting the architecture. This document describes the most common architectures in use today.

Several risk factors are associated with managing huge systems, including loss of control under failures of the human-computer system. Strategies for avoiding these failure modes are discussed.

Scaling gracefully is not just about handling volumes of machines, but also about comprehensibility and manageability to human engineers.

This is an incomplete draft document. Last updated October 2011.

Table of Contents

1 Principles of scalability

1.1 What is scalability?

By scalability we mean the intrinsic capacity of a system to handle growth. Growth in a system can occur in three ways: by the volume of input the system must handle, or in the total size of its infrastructure, and by the complexity of the processes within it.

For a system to be called scalable, growth should proceed unhindered, i.e. the size and volume of processing may expand without significantly affecting the average service level per node.

Although most of us have an intuitive notion of what scalability means, a full understanding of it is a very complex issue, mainly because there are so many factors to take into account. One factor that is often forgotten in considering scalability, is the human ability to *comprehend* the system as it grows. Limitations of comprehension often lead to over-simplification and lowest-common-denominator standardization. This ultimately causes systems to fail due to an information deficit.

1.2 How does CFEngine address scale?

CFEngine is a decentralized (or federated) agent-based system, with no single point of failure. It uses integrated Knowledge Management to present comprehensible views of how infrastructure complies with user intentions.

In this Special Topics Guide, we take a simple approach to gauging the scalability of CFEngine, considering the worst case scalability of the software as a management system for typical environments and network models.

1.3 What does scalability depend on?

CFEngine's scalability is not only a function of the CFEngine software, but also of the environment in which it operates and the choices that are made. Some relevant environmental factors include:

- The capacity of the network.
- The capacity of the server that supplies common information to agents.
- The extent to which parallelism can be employed during updates (Amdahl's law).
- The size and cost of the tasks carried out by the management system (in time and resources).
- The social contract between users and parts of an organization (don't forget that you are really dealing with a human-computer system).

For managers, there are several challenges to scaling that go beyond the infrastructure:

- The ability to express and comprehend *necessary complexity and variation* in policy.
- The ability to process the result (agent efficiency).
- The ability for a significant number of people to understand the result (comprehension).

CFEngine does all processing of configuration policy at the destination node (i.e. on the affected system). There is no centralization of computation. CFEngine Nova adds fault-tolerant multi-node orchestration, in which any node's state can be made available to other nodes on request.

In general, reliance on common or centralized information will limit the inherent scalability of the system. However, through opportunistic use of caching, CFEngine is able to avoid single points of reliance. CFEngine's asynchronous promise model makes the impact of resource sharing less significant.

1.4 A product strategy for scaling

CFEngine's scaling behaviour follows an 'astronomical' hierarchy of scales. Our product range have been chosen to model different issues of scale that occur in systems as they grow from tens to tens of thousands of machines. As indicated above, dealing with scale is not just about machine capacity, but also about knowledge management and comprehension.

An *Enterprise* installation is designed around a single star configuration (like a solar system) in which the hub machine is the star and the managed entities are the planets. Special planets can have their own satellites (customized environments within the single point of control), so this model does not imply complete uniformity.

Future releases will be designed around multiple star configurations and for cases where it is desirable to maintain several points of control or independently managed entities. This is also called a federation of star networks. The reason for choosing this kind of architecture may or may not have to do with service capacity (i.e. for coping with a large number of systems): sometimes knowledge management and responsibilities scale better with federation.

1.5 A user strategy for scaling

This section proposes a simple method for evaluating capacity and scalability at a site. You will need some numbers in order to do some simple calculations.

- Identify the key scales and constraints of your organization.
    - How many distinct environments or requirement sets do you have?
    - How many machines (managed units) are there?
    - What frequency of system state checking is desired (certainty about policy, or control resolution)?

You will be able to use these numbers in the next chapter to work out how far a simple star network configuration (Nova starburst) will go in supporting requirements. An environment that consists of multiple environments, or very large size will have to be handled as a constellation configuration.

1.5.1 Scalable CFEngine architecture

Several architectural principles aid the ability to scale to large size:

- **Patchwork coverage and federated centres**: Most architectures have some kind of central point of change or control, but too much centralization leads to bottlenecks that hinder throughput (see the Special Topics Guide on Federation and Organizational Complexity). Build a federated architecture from the beginning, i.e. a number of hubs or star networks that each covers the requirements for the most local environment. Do not try to make one single model that applies to everything (Grand Unification is an unstable process).
- **Necessary and sufficient complexity**: Do not oversimplify issues to avoid multiple environments. Delegation is cheap and is mainly an issue of trust. Delegation (decentralization) is a key principle of scaling as it avoids concentration of resources and single points of failure.
- **Don't over-constrain systems**: It is wise to avoid configuration rules and requirements that are not necessary, as this can impact systematically on the resources needed to scale. Similarly, don't waste time creating perfect container classes for rules. Rough patches are cheap to maintain – precision targeting costs resource logic.
- **Autonomy – avoid strong dependence**: If systems depend strongly on other systems (i.e. a failure of one leads to a failure of the other), then one creates fragility. Robust, fault tolerant systems avoid interval coupling or dependencies as these can lead to cascade failures.
- **Strive for efficiency**: Management is not free, but one does not generally account for the overhead when designing production systems. In the next chapter, we shall show how to make rough estimates about network requirements for management. As a rule of thumb, a site engineer needs to avoid clogging the network with traffic, and burying systems in CPU or memory intensive work.

1.5.2 Layout guidelines for scalability

The picture of a federated architecture is that of a number of smaller star networks loosely rather than tightly integrated together. This loose coupling avoids rigidity that can cause cascade failure.

This schematic architecture does not answer where these centres will be located however. Will the division into local centres be based on geography, departmental lines, or some other virtual view of the organization?

It does not matter from an architectural point of view what criteria are used for dividing up an organization. The main criteria is the environment itself. If we think in terms of promises for a moment, a strong group culture forms when members of a community make a lot of promises to one another. Organizational entities are therefore clusters of promises. This might or might not coincide with naming of institutional entities.

It is easy to understand the reason for a promise-oriented approach to scalability. Clusters of promises are also clusters where communication is likely to be required and take place. Since scalability is enhanced by limiting the amount and scope of communication, the clusters of promises mark out the areas (sub-networks, if you like) where communication is necessary. This makes for a natural encapsulation of policy issues.

- Model the organization (focusing on business-level issues)
- Give names to the parts
- Identify patterns
- Form a disciplined protocol (best practice)

1.6 Unexpected risks of scaling

- **Division of labour => fragmentation of knowledge**: When problems are management intensive, the need to divide and conquer problems leads to points of failure.
- **Turnover of staff**: Too much specialization of roles means that individuals attain mission critical positions that lead to single points of failure for operations. When key employees leave the organization, this damages operations due to loss of expertise. The cost of this lies in retraining, or even redesign. All roles should have redundancy both for quality assurance and failover, and strong Knowledge Management is required to avoid this scenario.
- **Morale**: When engineers feel powerless, they are demoralized and feel unimportant. This can lead to both disgruntlement and power-hogging. This is a failure of the social contract.
- **Scale reduces certainty**: The larger a system, the less detail human decision-makers can know about the whole.

2 Scalable policy strategy

2.1 Policy guidelines for comprehension

Part of the challenge of scale is comprehension. If every computer is identical, this is no problem, but when there is real growth in complexity scale can lead to a case of information and comprehension overload. Checking that growth of complexity requires some user discipline.

2.1.1 Strategy for scaling policy

What if you have written down 250,000 promises to keep? How can you manage that? Here are some tips:

- **Don't have 250,000 promises**. If your conception of the problem is this complicated, then CFEngine is not your problem. Management will fail by its own overheads unless it can slim down problems to *necessary and sufficient* complexity. Over-constrained systems usually fail eventually because they become overwhelmed by the problem of unintentionally conflicting requirements. Keep it simple.

  If you are migrating from CFEngine 2, then many of your separate promises can be turned into a single promise by using lists and other patterns. This will drastically improve the modelling capabilities and reduce the complexity of the policy.
- **Divide up your management problem and delegate** to autonomous entities that can manage their own pieces. It does not matter how you divide up your organization (geographically, logically, by department, by application team, etc) the important thing is to spread the load of responsibility. **Who proposes the promises to be kept.**

  The criterion for dividing up the organization is to make the entities as autonomous as possible, i.e. with as few dependencies or communication requirements as possible. The configurations for each autonomous entity should be kept and managed locally by those entities, and not mixed together. Centralization is the enemy of scaling.

When writing the CFEngine bundles of promises, you will need to use a 'meta-model' to organize the bundles. This is part of a Knowledge Management (comprehension) strategy. Your best friend here is someone with didactic or pedagogical skills, as he/she will maintain the visibility of high level goals throughout the technical challenges. Your worst enemy is a technician with his head in the machine, who will drag everyone down to the machine components, where goals and challenges are incomprehensible.

- **Start by modelling your business challenges**, not technical solutions. For example, name bundles by service

  ```
  bundle agent service_email
  ```

  not by configuration file:

  ```
  bundle agent etc_conf_postfix
  ```

  Even a non-expert should be able to see what these bundles are for, even if they don't understand their detailed content.
- **Focus on the broad strokes**. Don't micro-manage details that are not necessary.
- **Seek stability and predictability** of your system before turning your attention to other details. If you don't have predictability, you have nothing. This will keep you away from a focus on unhealthy technical detail.

- **Keep promise descriptions at a high level** as far as possible, and use the Copernicus Knowledge Map to locate low level resources and the promises they make.
- **Use** *patterns* **to model similar configuration issues** as a single promise iterated over the pattern, not as many individual promises. This allows you to compress a large number of issues into a small amount of text. The ability to comprehend patterns is also central to human understanding, as it shows the principles or allows us to *'see the general in the particular, or the eternal in the transitory'*.

2.1.2 Tactics for scaling policy

In writing policy, the thing most often forgotten by technicians is explaining *why* decisions have been made.

- **Give meaningful (high level, or even goal oriented) names** to collections of promise attributes. e.g. instead to writing `perms => m("0600")`, try writing something that allows readers to understand the *reason* for the intention: `perms =>write standard_permissions`.
- **Use comments to explain** why the promise is the way it is. e.g. write

  ```
  comment => "This file needs to be writable by the web server
              else application XYZ breaks"
  ```

  instead of

  ```
  comment => "Set permissions on the temp directory"
  ```
- **Encourage good practice in the policy authors**. Check whether objects have made other promises elsewhere, and try to document the connections. If a policy have dependencies, encode these (see 'Best practice for writing promises' in the Reference manual).
- When solving problems, think about how to model the data. For example, one way to model groups is to try to classify the names in lists

  ```
  "group" slist => {
                   classify("a.domain.com"),
                   classify("b.domain.com"),
                   .....4000x...
                   };
  ```

  This is neat, and easy to read but it requires CFEngine to process a list linearly, which becomes increasingly inefficient and can take several seconds if a list contains thousands of hosts. Some improvement can be obtained by converting the domain strings manually:

  ```
  "group" slist => {
                   "a_domain_com",
                   "b_domain_com",
                   .....4000x...
                   };
  ```

  However, the linear scaling is still present.

  In this case, it is inefficient to process the list in memory, because we only need one out of thousands of the entries, thus it makes sense to prune the list in advance.

  To handle this, we can create a flat file of data in the format `"hostname:group"` using the builtin function `getfields()` to read one line from the file.

```
    vars:
     "match_name" int => getfields("a.domain.com:.*","/my/file",":","group_data");

    classes:
     "$(group_data[2])" expression => isgreaterthan("$(match_name)","0");;
```

This assumes, of course, that each host is in one and only one class context. The saving in processing is large, however, as it can be carried out directly in the input buffer. Only one out of thousands of lines thus needs to be processes. Although the scaling is still linear in search, the allocation and read processes are heavily optimized by block device reading and much lower overhead.

A module could also be used to the same effect to define the appropriate class context.

2.1.3 Caching classes that are expensive to compute

As of version 3.4.0 of the CFEngine core, persistent classes can be used to construct a simple time-saving caching of classes that depend on very large amounts of data. This feature can be used to avoid recomputing expensive classes calculations on each invocation. If a class discovered is essentially constant or only slowly varying (like a hostname or alias from a non-standard naming facility)

For example, to create a conditional inclusion of costly class definitions, put them into a separate bundle in a file '`classes.cf`'.

```
# promises.cf

body common control
{
cached_classes::
  bundlesequence => { "test" };

!cached_classes::
  bundlesequence => {  "setclasses", "test" };

!cached_classes::
  inputs => { "classes.cf" };
}


bundle agent test
{
reports:

  !my_cached_class::
   "no cached class";

  my_cached_class::
    "cached class defined";
}
```

Then create ‘classes.cf’

```
# classes.cf

bundle common setclasses
{
classes:

  "cached_classes"            # timer flag
         expression => "any",
        persistence => "480";

  "my_cached_class"
                or => { ...long list or heavy function... } ,
       persistence => "480";

}
```

2.2 Performance impact of promises

In a large system it is natural to expect a large number of promises. This makes the location of a specific promise difficult. The Copernicus Knowledge Map is a key strategy for locating promises.

- Using a very large number of bundles will have a performance impact, as a local environment has to be established for each bundle.
- Using a large number of different input files can have a performance impact during file updating, as each file requires a bi-directional verification involving network traffic. The frequency of policy updates can be limited
  - Test systems – can update relatively often since there are fewer test machines, and their resources are less important than production machines.
  - Production – fewer changes need to be made since most delta changes have been aggregated through the testing phase.
- There is no performance impact involved in having multiple promise-type sections, e.g. `files:` in a bundle. However, splitting up type-sections makes human readability harder.

2.3 Avoiding network traffic

Connecting to a CFEngine server process is one of the most time consuming activities in centralized updating. Every bidirectional query that has to be made adds latency and processing time the limits scalability. The connection time and data transfer size during checking for updates can be minimized by making use of the ‘`cf_promises_validated`’ cache file on the server. This file summarizes whether it is necessary to search for file updates (a search that can take a significant number of seconds per client). Since most checks do not result in a required update, this cache file can save a large amount of network traffic.

```
     files:
```

```
 "$(inputs_dir)/cf_promises_validated"
     comment => "Check whether new policy update to reduce the distributed load",
      handle => "check_valid_update",
   copy_from => u_dcp("$(master_location)/cf_promises_validated","$(sys.policy_hub)"),
      action => u_immediate,
     classes => u_if_repaired("validated_updates_ready");

am_policy_hub|validated_updates_ready::

 "$(inputs_dir)"
     comment => "Copy policy updates from master source on policy server if a new val-
idation was acquired",
      handle => "update_files_inputs_dir",
   copy_from => u_rcp("$(master_location)","$(sys.policy_hub)"),
depth_search => u_recurse("inf"),
file_select  => u_input_files,
  depends_on => { "grant_access_policy", "check_valid_update" },
      action => u_immediate,
     classes => u_if_repaired("update_report");
```

2.4 Defining classes

Modelling environments with classes is a powerful strategy for knowledge management, and is therefore encouraged.

2.5 Using the Copernicus Knowledge Map

The Copernicus Knowledge Map is an integral part of the commercial CFEngine products. It is also a feature that is developing rapidly as part of the CFEngine commitment to research and development. It forms a browsable 'mental model' of relationships between promises, goals and documents that describe them (including the manuals and other documentation sources). The map provides you with an overview of how parts of your policy relate to one another, and to other high level parts of your environment.

- Shows where and when promises are relevant.
- Gives contextual meaning to promises and other issues by showing you their impact on both practical and abstract issues.
- Offers commentary in a browsable and user friendly form.
- How promises relate to business goals.
- Dependencies between promises.
- Overview of promise bundles and their contents.
- Browsable view of promises and their relationships.
- Browsable view of body parts in expanded form.
- Examples of code usage.

Use the knowledge map to:

- Find conflicts of policy.
- Avoid repetition.
- Understand relevance and impact.

2.6 System Tuning and Hub Optimization

CFEngine uses MongoDB as its repository of information for each Nova/Enterprise hub. The performance of a hub depends on the combination of hardware and software. The CFEngine hub falls under the category or role of database server, and this requires fairly specific optimizations. For example, NUMA architecture processing is known to lead to severe processing bottlenecks on database servers, and so NUMA kernel modules should be switched off. Below are some of the optimizations that should be looked into for the CFEngine hubs.

2.6.1 Tuning the linux kernel for thousands of hosts

Although CFEngine communicates with the Mongo database over a local socket, it still uses TCP as its connection protocol and is therefore subject to kernel optimizations.

Every write and read connection to the Mongo database makes a kernel TCP connection. With the extreme density of connections, this is somewhat like a high volume webserver. The standard 'play safe' kernel settings are too conservative for this kind of performance. The main bottleneck eventually becomes the FIN_WAIT timeout, which leaves file descriptors occupied and non-recyclable for too long. We recommend reducing this waiting time to free up descriptors faster:

```
echo "1024 61000" > /proc/sys/net/ipv4/ip_local_port_range
echo "5" > /proc/sys/net/ipv4/tcp_fin_timeout
```

This will clear old connections faster, allowing new ones be created and old threads to terminate faster.

2.6.2 Tuning the MongoDB for thousands of hosts

Some points to consider when scaling the MongoDB:

- Indexing is a very important factor for scaling any database. CFEngine Nova automatically checks and the creates indices needed for scale as part of the schedule of `cf-hub`. The incices are checked every six hours.

  However, if the database schema changes, which may happen during upgrades of CFEngine Nova, the indices may have been changed as well. In large-scale environment we may not have time to wait up to six hours for the indices to get repaired, so this can be done manually by running the following command on the hub.

  ```
  /var/cfengine/bin/cf-hub --index
  ```

  Please make sure `cf-hub` is not running in the background while indices are being created, as this may slow the system down considerably under high load.
- For installations of one or two thousand hosts per hub, one approaches the throughput limitations of x86 hardware and the Mongo database. One can expect to see MongoDB writes dominating the system resources. MongoDB does extensive caching'in RAM, so maximizing installed RAM is an important strategy – however disk access priorities will also play a role.

  To make sure that server connection performance does not suffer as a result of aggressive database writing, we can lower the priority of the MongoDB process

  ```
  ionice -c2 -n0 /var/cfengine/bin/mongod
  ```

  or, using the PID

  ```
  ionice -c2 -n0 -p PID
  ```

- If the MongoDB database is located on the same disk as the last-seen database (as is default under '`/var/cfengine`', there will be contention between last-seen and mongodb updates, which can slow down both processes unnecessarily. Ideally, these should be separate disks with independent queues, even independent controllers. RAID configurations can also slow down performance, so only hardware RAID should be considered.
- Non-Uniform Memory Access (NUMA) hardware works poorly with all databases. If your server uses such hardware, you should try to switch off this feature to improve write stability. NUMA tries to bind memory to specific cores for speed, but in doing so it can prevent free allocation of memory and lead to paging and swapping, thereafter thrashing as the system grinds to a halt. Processes become CPU bound as threads attempt to work around the memory manegement constraints, and performance worses by a factor of ten or more.

  Some users report that using this approach is sufficient:

  ```
  numactl --interleave=all /var/cfengine/bin/mongod # (other args)
  echo 0 > /proc/sys/vm/zone_reclaim_mode
  ```

  However, in our experience, only removing the kernel modules supporting NUMA and rebooting the kernel will solve the NUMA contention issue.

  Setting:

  ```
  numa=off
  ```

  in the kernel boot parameters, e.g. in '`grub.conf`':

  ```
  # grub.conf generated by anaconda
  #
  # Note that you do not have to rerun grub after making changes to this file
  # NOTICE:  You do not have a /boot partition.  This means that
  #          all kernel and initrd paths are relative to /, eg.
  #          root (hd0,0)
  #          kernel /boot/vmlinuz-version ro root=/dev/sda1
  #          initrd /boot/initrd-version.img
  #boot=/dev/sda
  default=0
  timeout=5
  splashimage=(hd0,0)/boot/grub/splash.xpm.gz
  hiddenmenu
  title MY Linux Server (2.6.32-100.26.2.el5uek)
    root (hd0,0)
    kernel /boot/vmlinuz-2.6.32-100.26.2.el5 ro root=/dev/sda1 rhgb quiet numa=off
    initrd /boot/initrd-2.6.32-100.26.2.el5.img
  ```

- Finally, `splaytime` on the client nodes will help to spread out the incoming connections over the update interval of the hub. The hub defaults to 5 minute updates, so a complete update of policy should be spread over less than or equal to the same scheduled cycle time. Alternatively the cycle for policy updates can be extended and the splaying can be even longer.

  ```
  body executor control
  {
  splaytime => "4";
  ```

```
    }
```

3 Internal and external scalability of the software

Scalability is about the internal and external architecture of the system, and the way that information flows around the highways and bottlenecks within it. As a distributed system, some of those are internal to the software, and some lie in the way it is used.

CFEngine allows users to build any external architecture, so the only intrinsic limitations are those internal to the software itself. Poor decisions about external architecture generally lead to greater problems that the internal limitations.

A number of techniques are used internally to bring efficiency and hence scalability in relation to policy.

Hashing and indexing
: are used in many ways to coordinate system without the need for communication. By basing decisions on the predefined and publicly available information about a node (like its name and address), we avoid having to pass messages between nodes.

Classifying of patterns
: is used to model patterns of similarity and difference in the intentions for a system. By describing policy for different categories of system parts, rather than for each individual part by name, we can reduce the amount of information we need to specify.

Persistent locking of execution
: is used to place limits on the frequency and concurrency of operations undertaken by CFEngine. Using the `ifelapsed` timers on each promise, one can say how often work-intensive checks are made.

Lazy evaluation
: is a common approach to efficiency, but it is difficult to achieve in a dynamic environment. Necessary complexity makes for necessary work. CFEngine uses best-effort lazy evaluation to reduce processing, but errs on the side of correctness, reliability and security.

The greatest challenge for scalability is to reduce functional requirements to a description or model that uses patterns (i.e. general rules) to compress only what needs to be said about the policy into a comprehensible form. It is about identifying a *necessary and sufficient* level of complexity without over-simplification, and it is about not having to specify things that don't need to be specified. This is Knowledge Management.

3.1 Best case approach to external scalability

In the most scalable approach to management, each agent works 100% autonomously as a standalone system, requiring no communication with its peers, or with a central agency. Its policy is therefore constant and fixed.

If this model suits your organization, the operation of CFEngine is completely independent of the number of machines, so it exhibits *perfect scaling* with respect to the size of the infrastructure. However, this model is too simple for most sites, and its value lies in documenting the extreme end of the scalability scale, as an ideal to work towards when striving for efficiency.

3.2 Failover and redundancy

As a self-healing system, CFEngine poses a low risk to loss of data. Most management data that are lost in an incident will recover automatically.

Failover due to unavailability.
: The loss of a hub is not a critical failure in a CFEngine managed network. Client machines continue to work with the last known version of policy until a hub returns to 'online' status. During updates, hubs can come under load. To avoid this, splaying options should be used primarily[1]. CFEngine's failover servers for file-copying can be set up to offer immediate redundancy.

Shared storage for '`/var/cfengine/masterfiles`'.
: The only place where shared storage makes sense is between the main policy server (hub) and any failover servers. In that case only this one directory can be shared.

Backup of '`/var/cfengine`'.
: It is not strictly necessary to have a backup of the CFEngine work directory, but keeping a backup of this for the hub can save time when restoring a broken hub, since the records of known clients are stored in this workspace, and the current status of reports is also stored.

: On clients, this workspace can be considered disposable, but many users save public private key-pairs to preserve the 'identity' of hosts that have merely disk failures, etc.

Multiple reporting hubs.
: Although it is technically possible to have multiple reporting hosts, this is recommended against. Multiple hubs will only increase the overhead on all parts of the system for little return.

Our general recommendation is to introduce as few network relationships as possible (make every system as autonomous as possible). Shared storage is more of a liability than an asset in general networks, as it increases the number of possible failure modes. Backup of the hub workspace is desirable, but not a show-stopper.

3.3 A simple worst case approach to scalability: the star network

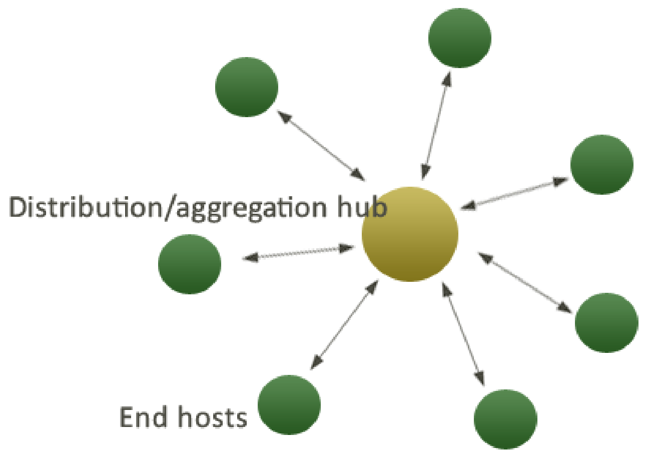
CFEngine 3 Nova is designed around the concept of a simple star network, i.e. a number of 'client' machines bound together by a central hub. This is a commonly used architecture that is easy to understand. The hub architecture introduces a bottleneck, so we expect this to

[1] We strongly recommend users to abandon the idea that it is possible to have 'instant' or 'immediate' updates. There is always some delay. It is more pragmatic to manage that delay by making it predictable than to leave it to chance.

have a limited scalability as long as we cannot increase the power and the speed of the hub without limit.

To understand the external scalability of CFEngine management, we shall estimate how large a CFEngine system can grow using simple centralized management, in this star network pattern. This will force us to confront low level performance characteristics, where we try to extract the most from limited resources.

In a star network, all agents connect to a central hub to obtain possibly frequent updates. The challenge is to optimize this process[2]. This makes the central hub into a bottleneck. It becomes the weakest link in the chain of information, i.e. the star network has limitations that are nothing to do with CFEngine itself. The counterpoint is that the star network is very easy to comprehend, and thus it usually forms the starting point for most management frameworks.



Centralized management with a hub.

In the Community edition of CFEngine, updates travel only in one direction, from server to client. CFEngine Nova, adds to this the collection of data for reporting and analysis (CFDB), from client to hub. Thus the expected load on the server is the sum of both processes.

We would like to answer two equivalent questions. Given a central hub with certain limitations, how many clients can it support? Conversely, given a number of hosts to support with a single hub, what capacity is required in system infrastructure to support it at a certain level?

There are actually two independent issues here. There is the scaling of the policy updates (served by the hub) and the scaling of the reporting updates (collected by the hub). Both of these processes compete for resources.

[2] The estimates here are based on CFEngine core versions 3.1.2 and higher.

3.3.1 Typical CFEngine scales

We start by setting some fundamental scales.

- The interval at which policy and reporting updates are checked ($\Delta t = 5$ mins or $300s$).
- The expected worst size of an update ($U \leq 5$ MB per host).
- The amount of RAM available on the server ($M = 1$ GB)..
- The resident size of a thread or server process ($s \leq 12$ MB).
- The maximum available network capacity for management processes ($C \geq 1$ MB/s).

We choose deliberately a low value for network capacity. Even if more capacity is available, one does not typically expect to use it all for management overhead.

3.3.2 Scaling of updates

To a first approximation, the process scaling relationships are linear, as long as we stay far away from the region of resource contention at which a server will typically perform very quickly. Scalability is thus about having safe margins for worst case behaviour.

The number of threads t supported by a server satisfies:

$$st \ \leq \ M$$

and yields a value for t, given fixed values for the hub's RAM and software build size. For a server with 1GB of memory, we have $t \leq 1024/12 \sim 80$ threads. This will consume $U \leq 1MB$ of data per agent for a community host, and $U \leq 5MB$ for Nova to account for reporting.

The expected time-to-update τ_{round} satisfies:

$$U_{\text{min}}/C \ \leq \ \tau_{\text{round}} \leq \ U_{\text{max}}/C$$

The actual values will normally by significantly less than this worst case, but he must plan for the possibility of update storms. The network capacity $C = 1MBs^{-1}$ suggests an expected time-to-update $\tau_{\text{round}} \leq 2s$ (there and back) with a server-processing overhead. We shall take a conservative value of $\tau_{\text{round}} \leq 10s$ for the maximum round trip request time.

Let's consider the policy/reporting update process, as this dominates the scaling behaviour of the software in terms of number of machines. Assuming that we can arrange for agents to distribute their updates over a single time interval Δt, with maximum possible entropy[3], then we should be able to achieve a maximum number of scheduling slots σ:

$$\sigma \ = \ \Delta\, t/\tau_{\text{round}}$$

Since the round-trip time can be higher or lower, depending on the size of the update, we can propose some approximate limits. $\sigma_{\text{max}} = 300/2$, $\sigma_{\text{min}} = 300/10$ per interval Δt.

The total number of updates that can complete in the Δt interval is thus, at fixed t:

[3] Maximum entropy means the most even or flat distribution over the time interval, in this case. See, for instance, Mark Burgess, *Analytical Network and System Administration*.

$$N \;=\; \sigma\, t \;=\; \frac{\Delta\; t}{\tau_{\text{ round}}} t$$

If there are now t threads, then the total number of updates available must lie between the upper and lower bounds determined by σ:

$$2400 \;\leq\; N \;\leq\; 12,000.$$

So we can gauge a reasonable lower bound of 2000 machines from a single hub, with extremely conservative estimates of the environment. Note that when exceeding several thousands connections over a short time interval, other limitations of the operating system will normally start to play a role, e.g. the maximum number of allowed file descriptors. Further, comprehending a system of more than a few thousand machines is a challenge unless the system is extremely uniform.

In making each of these estimates, we are assuming that the hub machine will be working almost to capacity. Increasing its resources will naturally lead to improvements in performance.

3.3.3 Limitations on the CFDB hub database

A 'star network' architecture has a single concentration of processing, centred around registration of incoming and outgoing hosts on the hub. Read/write activity is focused on two databases:

- The last-seen database that records the current IP addresses of known clients to the hub, accessed every time a host connects or is connected to the hub.
- The document database where reports are stored (using MongoDB), accessed each time an update is stored, or when a user makes a query.

The scalablity of the hub depends on how efficiently reads and writes can be parallelized to these databases. Even with aggressive cachine, databases are disk-intensive and since disk access is typically the slowest or weakest link in the data flow chain, disk accesses will throttle the scalability of the hub the most.

CFEngine tries to support efficient parallelization by using multiple threads to serve and collect data, however access to a shared resource must always be serialized, so ultimately serial access to the disk will be the limiting factor. The CFEngine hub supports Linux systems. Simple SATA disks, USB and Firewire have very limited performance. To achieve the maximum scaling limit of a few thousand hosts per hub, users can invest in faster interfaces and disk speeds, or even solid state disk devices.

Because of the serialization, a heavily loaded MongoDB will eat up most of the resources on the hub, dominating the performance. Some performance tuning option strategies are discussed below. Clearly maximizing the amount of RAM on the hub is a way to improve the performance.

The time estimates for host updating used above include the latency of this database, but it is possible that there might be additional delays once the amount of data passes a certain limits. We currently have no data to support this assumption and await customer experiences either way.

Lookup times for data in the reports database increase with the number of host-keys, so the time required to generate certain reports (especially when searching through logs) must increase as the number of hosts increases.

3.3.4 Update storms

When changes are made, many hosts will start downloading updated files. This can have a sudden impact on the network, as a lot of unexpected traffic is suddenly concentrated over a short interval of time. The contention from these multiple downloads can therefore make each download longer than it otherwise might have been, and this in turn makes the problem worse. The situation is analogous to disk thrashing.

Once thrashing has started, it can cause greatly reduced performance and hosts might pass their ‘blue horizon threshold’, appearing to disappear from the CFEngine Mission Portal. This does not mean the hosts are dead or even ‘out of control’. It only means that updates are taking too long according the tuning parameters. The default update horizon is.

3.4 Optimizations affecting scalability

3.4.1 The role of caching and indexing

To assist the speed of policy processing by `cf-agent`, choose class names evenly throughout the alphabet to make searching easier. CFEngine arranges for indexing and cachine to be performed automatically.

3.4.2 Deterministic queue

3.4.3 Push versus pull

3.5 Redundancy and load balancing in the Nova hub

CFEngine does not *need* a ‘high availability’ architecture to be an effective management system. The software was designed to work with very low availability, and the designers highly recommend avoiding the introduction of dependencies that require such availability. In normal operation, CFEngine will be able to continue to repair systems without any contact with the outside world, until actual policy changes are made.

It is nonetheless possible to balance load and account for failures by multiplying the number of hubs/policy servers in a Nova star network. This only makes sense for the policy servers and reporting hubs, which are in principle single points of failure.

- Availability of policy in case of policy server failure.
- Availability of reports in case of reporting hub failure.

Because CFEngine is a pull-based technology, the two processes look like this:

- Satellite ‘client’ nodes pull policy from a policy server.
- Reporting hubs pull reports from all satellite clients.

In both cases, the processes are ‘self-healing’. If a client is unable to connect to a policy server, it will continue to work with its old policy until the policy server becomes available again. If a report hub fails and reports are no longer accessible, no data are lost because the actual data are sourced from the satellite nodes. When the hub is replaced or restored, it will update and continue as before.

3.5.1 Balancing multiple policy servers

A simple way to balance client hosts across multiple servers is to split them into classes using the `select_class` function[4]. Nothing else is needed to spread computers evenly into a set of named buckets.

```
classes:

  "selection" select_class => { "bucket1", "bucket2" };
```

This selects a particular class for each host from the list. A given host will always map to the same class, thus allowing the rule for policy updates to include copying from a personal 'bucket' server.

```
body copy_from xyz
{
...

bucket1::
     servers     => { "server-abc-1.xyz.com", "failover.xyz.com"  };
bucket2::
     servers     => { "server-abc-2.xyz.com", "failover.xyz.com"  };

...
}
```

Note this method is not suitable for bootstrapping hosts in a hub configuration, since that requires a one to one relationship documented in the '`/var/cfengine/policy_server.dat`' resource.

3.5.2 Redundant reporting hubs and recovery

A CFEngine reporting hub is a host that aggregates reports from all managed hosts in a Nova/Enterprise cluster. The purpose of aggregating reports is to have all the information in one place for searching, calibrating and comparing. The loss of a hub is an inconvenience rather than a problem, as the function of a hub is to make access to information about the system convenient and to enhance the knowledge of users.

- The loss of a hub will not impact the correctness of host configurations.
- A hub will never come under heavy query load, so there is no need for load balancing.
- Running multiple hubs on the same set of hosts will only lead to network contention and increased overhead.
- If a hub serves so many computers that updating the reports becomes a burden, then you have probably already passed the point at which it makes sense to divide up your management into multiple hubs.

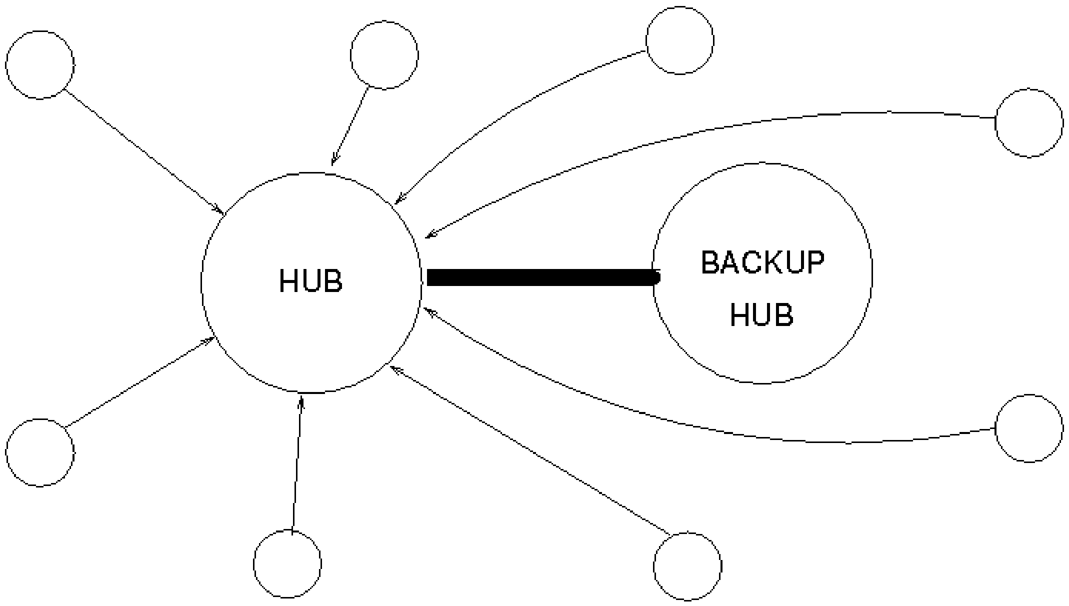
Dividing hosts into multiple hubs should be a decision based on the absence of the 'need to know' for all hosts. By splitting the aggregation up into multiple hubs, one jettisons the function of the hub which is to aggregate information. The should normally be done in conjunction with a decision about knowledge boundaries in your organization.

[4] This feature was added in core 3.1.5.

It makes sense to back up some information from a reporting hub. Hub information is considered to be a mixture of two kinds of data:

- Cached information for data-mining, whose original source is the client from which it was collected. Such information may be collected again if lost, and thus there is no need to back it up. It can, of course, be backed up for convenience.
- Original comments and hand-entered data about systems that are entered by users of the Mission Portal. These data should be backed up as they cannot be recovered from any other source.

In addition to these, the hub makes use of the 'last seen' database of known hosts in order to know where to collect data from. For rapid recovery after a data-loss catastrophe, it is recommended to back up this database also.



Some data can be backed up from a hub for convenience.

The procedure for backing up the ephemeral data is:

- Mongo collections can be dumped to an intermediary format, though this is impractical for large installations.
- For embedded databases, it is sufficient to copy the binary object:

  ```
  cp /var/cfengine/cf_lastseen.db  /backup
  ```