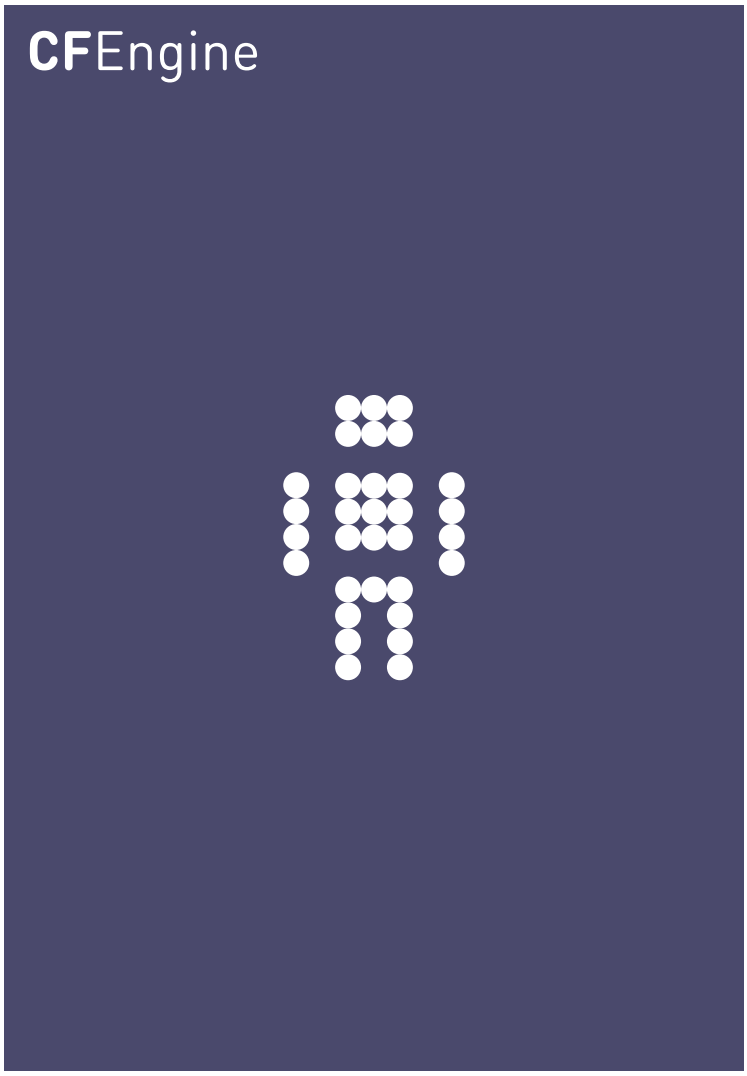

CFEngine for Mission Critical Operations (draft)

A CFEngine Special Topics Handbook

CFEngine AS

As IT services dominate the operations of an increasing number of industries, many companies now view all online services as mission critical. Today, mission critical no longer just means the protection of human lives, but the protection of crucial operational assets.

By making a fully redundant architecture for information updates, it is possible to make reliable promises about availability of status information. Users get reliable and predictable insight, with fault recovery times of only a few minutes in case of failure – something that cannot easily be matched by push-based centralization.

This guide explains how to set up redundant hub availability for a single Nova star network.

Table of Contents

What are Mission Critical Operations?

Mission Critical operation refers to the use and management of systems where the availability and correctness of a system has to be ensured at all times. A mission is said to be critical when any noticable failure in the system would cause a signifiant loss to some stakeholder.

Risk for mission critical systems deals with issues like monetary losses (e.g. in time critical trading applications) or, in the worst case, even the loss of human life (transport systems).

What makes a system robust in a mission critical setting depends on a number of factors. This Special Topics Guide discusses the role of CFEngine in a mission critical environment.

Factors affecting Risk

Risk is about predictability. The ability to predict the behaviour of a system depends both on the complexity of the system itself and the environment around it, since interaction with the environment is what usually provokes failures (the environment is the most unpredictable element of any system, since it is the part over which we have little control).

The cost of predicting and avoiding failures can be prohibitive in a purely manual regime but automated systems can do a lot to reduce costs. CFEngine can play a key role here in reducing the cost of maintaining system state, even in a rapidly changing environment.

- Planning for eventualities.
- Verifying system correctness with sufficient frequency.

The key observation for dealing with mission criticality is that systems are dynamical entities. Most software systems only manage the static setup of hosts. CFEngine manages both the static resources and the run-time state.

Model-based planning for stability

The key to handling mission criticality is to build a model of your critical scenario that is based on a prediction of behaviour. In science and engineering, this is something one does all the time (e.g. wind-tunnel studies), but in Computer Engineering, the methods of modelling are still quite undeveloped. In the nuclear power industry and space programmes, for instance, it is common to use formalized fault-analysis to avoid and secure against error.

The purpose of a model is to describe expectations. If a model is sufficiently well conceived, it should be possible to identify key causal factors in the mission that bring about critical behaviour.

CFEngine's methodology is based on the idea of promises: a promise being something that aims to alter our expectations of outcome in a postive way.

In CFEngine, you make promises about the factors that underpin the stability of your system, and CFEngine's task is to work on your behalf to keep those promises. Promises cannot be guaranteed 'kept' at all times, especially in time-critical situations (such a guarantee would require infinite resources to maintain), but a known schedule of verification and repair allows us to bring a level of predictability to a system, within certain tolerances. This is a best-effort engineering definition of predictability.

Examples of promises that you might want to include in a foundation for a mission critical system are things that bring trusted stability, e.g.

- Check that key processes and applications are running.
- Automated garbage collection that prevents a system from choking on its own biproducts.
- Scan for rootkits (security breaches) every few hours.

The economic aspect of mission criticality is key: the loss of a key application or subsystem for even a minute could result in loss of significant revenues in an online company, or the loss of flight systems for a few seconds could result in a plane losing control and crashing.

Key terminology for Mission Critical Systems

Mean Time Before Failure (MTBF)
: The average measured time between faults occurring on a system. Although this is a well established measurement in the theory of faults and errors, estimating this quantity is not without its challenges.

Mean Time To Repair (MTTR)
: The average time it takes to repair a system after a failure has occurred. The type or meaning of repairs is not specified.

Sampling frequency
: The rate at which we interact with the system in order to measure or repair it. According to Nyquist's theorem, we have to sample a system twice as fast as the rate at which we expect to detect an important change.

Single point of failure
: Any point in the design of a system that would lead to complete failure if destroyed. There might be several 'single points of failure' in a system. Single refers to the fact that it only takes the failure of one of these to cause the total breakdown of the system. For example, the axel, or a tyre on a car would be examples of single points of failure for the 'driving system'.

Strategy for Mission Critical Operations

There are many aspects to thinking about complete reliability of systems.

> The main goal of any system is to seek predictability. Having clear and accurate expectations of a system helps to steer it in a low-risk direction.

Usually these fall into a mixture of two categories:

Redundancy
: Elimination of 'single points of failure' when failure strikes.

Avoidance
: Proactive maintainence to keep the system in a zone of low risk.

Certainty of knowledge
: Knowing accurately what is going in a system can enable correct decisions to be made more quickly when something unexpected happens.

It is impossible to discuss a comprehensive list of points for ensuring reliability, but a few general principles come to mind:

- Maximize Mean Time Before Failure
- Minimize Mean Time To Repair
- Maximize the relevance of information from the system to mission goals.
- Separate procedures for handling change into those for intended change (planned changes to the mission) and unintended change (changes that should not happen in an ideal world), falling into two cases: expected (for which we have written policy to repair) and unexpected (incidents that are handled manually).
- Certainty about information returned by the system, with multiple confirmation.
- Graceful failure modes: failover servers, backups, automatic elasticity (e.g. cloud technology)
- Peak load handling. (Also called Long Tail events.)
- Design for self-correction (negative feedback controllers). This includes, low-impact of management overhead on the mission system to avoid cascade failure.

How CFEngine contributes to reducing mission risks

- Automated monitoring and repair according to a policy model.
- Providing up to date knowledge about the system
- Automatic restoration of compliance with policy, with MTTR 2.5 minutes by default.
- Accuracy of knowledge: all data include running estimates of the certainty of the data.
- Automatic updates of statistics about the system, with continuous updating for accurate and up to date information with context
- Independence of infrastructure dependencies (network/cmdb) CFengine will continue to work even if the network communications are impaired.

High availability access to the Mission Portal

CFEngine is designed to be a system that is resilient to failure. That, in fact, is the opposite of a high availability system, where failures are not supposed to occur at all.

The Mission Portal has a role to play in Mission Criticality, as it is a single source of information, collected, categorized and calibrated for system engineers. Being a single source website, it is can also be regarded as a single point of failure from the point of view of a mission critical application.

The information in the mission portal is largely status information about systems. The content of the Mission Portal database is not in any way deterministic for the configured state of your IT system – it is only a report of actual state, not a template for intended state. If the Mission Portal is 'down' or unavailable, it does not in any way imply that the actual distributed system is down or that there is any fault.

Setting up redundant monitoring hubs

If information and insight into your IT system are indeed Mission Critical for you, it is possible to create a high availability access to the mission information in the portal. In general, we recommend a small amount of professional services to help set up such a system, as there are several details that need to be taken into account.

The CFEngine star-network 'hub' is the report aggregator for the CFEngine commercial edition (Nova/Enterprise). CFEngine commercial editions support multiple hubs for redundancy during reporting. By making a cluster of three (or more) hubs, you can ensure that reports will always be available and up to date, at the time-resolution promised by CFEngine.

To set up redundant hubs, you will need three physical computers, or at least three virtual machines on different physical computers. The idea is to use the underlying technology of the MongoDB database to provide a replicated data store. If a single database server goes down a secondary replica can take over the role.The commercial editions of CFEngine interface with this database through `cf-hub`, and this process can be made aware of the underlying replica technology in the database. The architecture is intended to be as simple as possible for the CFEngine user to employ.

- Install each of the three systems with the Nova extension package for policy hubs.
- The MongoDB backend needs to be set up specially before standard bootstrapping of nodes in an high availability managed network. Alternatively, if you have already bootstrapped hosts, you can manually establish hub redundancy with a little database infrastructure work and some additional CFEngine configuration.

  To do this, configure the Mongo database to set up a minimal replica set. This underlying mechanism for automatic failover. For example, a configuration like the following should be typed into the mongo client on one of the hub machines. Text like the following example can be pasted directly into the mongo shell.

```
host$ mongo

db.runCommand({"replSetInitiate" : {
"_id" : "CFEngineNova",
"members" : [
{
"_id" : 1,
"host" : "10.10.10.1:27017"
},
{
"_id" : 2,
"host" : "10.10.10.2:27017"
},
{
```

```
"_id" : 3,
"host" : "10.10.10.3:27017"
}
]}})
```

To bootstrap the Mongo DB replication you should follow the procedure from the *MongoDB Definitive Guide, O'Reilly*. An example is shown below, assuming the three IP addresses for the hubs have IP addresses 10.10.10.1, 10.10.10.2, 10.10.10.2, we would arrange for the following CFEngine pseudo-code to be executed before bootstrapping any of the hubs:

```
commands:

  10_10_10_1::

     /var/cfengine/bin/mongod --fork       \
          --logpath /var/log/mongod.log    \
          --dbpath $(sys.workdir)/state    \
          --replSet CFEngineNova/10.10.10.2:27017

  10._10_10_2|10_10_10_3::

     /var/cfengine/bin/mongod --fork       \
          --logpath /var/log/mongod.log    \
          --dbpath $(sys.workdir)/state    \
          --replSet CFEngineNova/10.10.10.1:27017
```

Although we write the above in CFEngine pseudo-code, these steps need to be carried out before boostrapping hosts, as the Mongo services need to be initialized before anything can be written to the database, and the bootstrapping of the license initialization writes information to be stored in Mongo. During a single hub installation, these steps can be automated, but bootstrapping the replication prevents a fully automated installation.

- Copy the public and private key from the licensed hub, along with the '`license.dat`' file to the secondary hubs. All hubs will share the same public-private key pair and license file.

- Start the `cf-hub` on each of the three machines.

Next, we'll run through the operation and failure modes of the symmetric hubs. The arrangement of the hubs is shown schematically in the figure below.

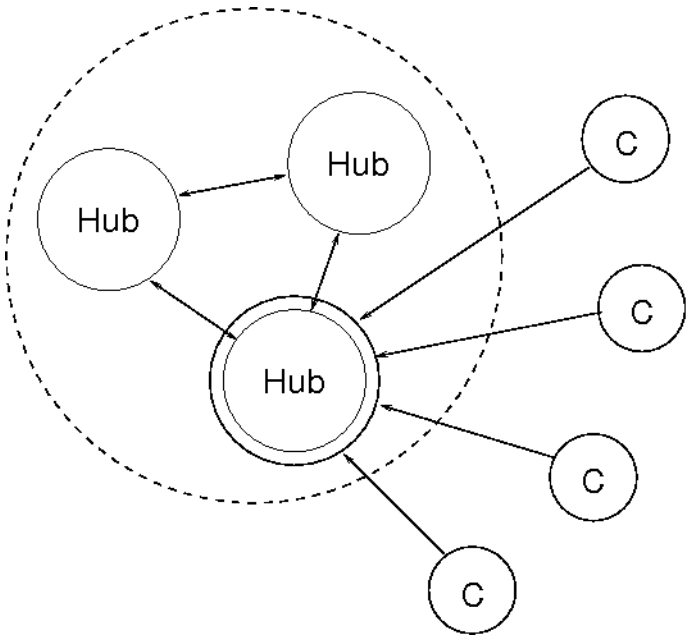

Fig 1. Only one master hub at a time collects data in a symmetric cluster.

Redundant hub architecture

To set up redundancy, you create three hubs, each running a Mongo database server and a `cf-hub` process will play the role of a virtual cluster. All three hosts make promises to one another to coordinate their data. The Mongo replica sub-system promises its own coordination independently. We essentially make three completely symmetrical hub hosts, with different IP addresses.

1. Set up three completely symmetrical hosts, with identical public-private key pairs. That is, generate a key pair only for one of the hubs and then copy those keys to '`/var/cfengine/ppkeys/localhost.pub`' and '`/var/cfengine/ppkeys/localhost.priv`' on the other two hosts. This must be done manually to bootstrap the hub redundancy.
2. The underlying Mongo database infrastructure binds together these hosts into a small cluster called a replica set.

   A voting mechanism selects a 'hub_master' from this set, which is going to be the active hub.
3. Each host is configured to copy the public keys from all the others, so that they all converge on the same set of keys. The following snippet shows the main principles involved in a replica setup.

```
vars:

  "hub_hosts" slist => { "hub1", "hub2", "hub3" };

files:

 am_policy_hub::
```

```
  "$(sys.workdir)/ppkeys"
      comment => "All hubs converge knowledge of client keys",
    copy_from => secure_cp("$(sys.workdir)/ppkeys","$(hub_hosts)"),
depth_search => recurse("inf");
```

4. For optimization we set up a policy that rewrites the IP address in 'policy_server.dat', to point existing clients away from a hub that is no longer responding, to the current master or primary. Clients will initially pick up these changes by failing over, as in the previous point.

```
files:

 am_hub_master::

   "$(sys.workdir)/policy_server.dat"
       comment => "Point clients to the current hub master",
     edit_line => append_if_no_line("$(sys.hub_master)")
  edit_defaults => empty;
```

5. To make the redundany hubs double as redundant policy servers, we make sure that the copying of policy in 'update.cf' uses the replicas as failover servers for policy updates. This means that changes to policy should always be copied to '/var/cfengine/masterfiles' on all three hub hosts.

```
body copy_from update_copy
{
...
servers => { $(sys.policy_server), "hub3", "hub2", "hub1" };
...
}
```

The policy server variable will point to one of these hubs. If a hub host, doubling in its role as policy server, fails for some reason, the clients will all fail over to the next hub, and updates will continue. By the time this happens, we can expect the main policy will have been adjusted by the edit in the previous point, and clients will be pointed to a new primary. It does not matter that hosts appear twice in the list; we could, for instance place hub1 last in the list if we assume that hub1 is the initial primary, so as to minimize the wait due to a double-failover.

With this configuration, all the hubs promise to synchronize their keys, and share 'last seen' client host data with each other in order to symmetrize. However, only one of the hubs actually collected reports from the clients. This is the host that is elected by the MongoDB replica-set vote.

With the approach taken above we can be sure that data are being collected redundantly without duplication of network overhead.

Note that the standard Nova configuration files contains example configurations for the replica set configuration. Integrating all the settings can require extensive modifications, as mentioned above.

Features of hub/policy server redundancy

The Nova starburst hub configuration supports the following properties:

- Redundant availability of policy changes, with automatic failover.
- Redundant responsibility for report collection from managed hosts, with automatic reassignment of hub master.
- Redundant availability for the Mission Portal console, with about 15 second changeover.
- Fault tolerance of all parts of CFEngine to complete network failure.

Variables and classes for hubs

CFEngine Nova/Enterprise supports special classes to help write policy for managing the hub infrastructure.

```
am_policy_hub
am_hub_master
```

The variable '`$(sys.hubmaster)`' is also available *on hosts that are hubs*, and points to that host currently voted into the role of hub master.

> Setting up redundant hubs might be best done with some professional service assistance. Although it is quite simple, it changes the bootstrapping procedure in the beginning of a Nova/Enterprise deployment.

How do I make a change in mission-critical infrastructure?

To make changes in a mission critical environment, you publish a fully tested, risk-assessed policy to '`/var/cfengine/masterfiles`' on all three of the hubs in the cluster. The policy on all hubs should be the synchronized. It might be worth setting up a promise to verify that the contents of these directories are the same between all hubs, since unsychronized policies could cause serious issues.

```
vars:

   "hubs" slist => { "hub1", "hub2", "hub3" };

files:

  "$(sys.workdir)/masterfiles"

    copy_from => secure_cp("$(sys.workdir)/masterfiles","$(hubs)"),
       action => warn;
```

All changes to a mission critical system should be classified according to the level of risk to the mission. Changes fall into different categories.

- Regular routine maintenance to the system (preplanned).
- Course or goal corrections to policy (replanned).
- Unforeseen repairs (unplanned).

In each case, we recommend that changes be made using CFEngine's hands-free automation. Humans should never be the instruments of change. By using CFEngine, you can get a documented and predictable handle on change, using technology designed for stability with agility.

CFEngine has the ability to change entire systems of thousands of hosts within a timeframe of 5 to 10 minutes.

Separating Policy Changes from *ad-hoc* Changes

In mission critical environments, there are often struct processes for approving change. Unintelligently applied, such rules can do more harm than good – i.e. when the process for approving changes causes greater risk to the survival of the mission than not acting at all does. It is important to separate changes into categories that allow the minimization of risk. If we take the categories in the previous section:

Regular routine maintenance to the system (preplanned).
: Changes here have alredy gone through risk assessment, and root cause has been deemed understood or irrelevant. These changes should be automated and implemented in the minimum time. e.g. restarting a web server that crashed or was stopped accidentally.

Course or goal corrections to policy (replanned).
: This is a change in the mission plan and requires significant impact analysis for each change. Although many businesses are concerned about liabilities, the real aim of this analysis is to mitigate loss.

Unforeseen repairs (unplanned).
: These changes are usually discovered and repaired manually. Once some kind of root cause analysis is performed to the required level, there should be an analysis of how to automate the prevention of this kind of change in the future.

This is a partially finished Special Topics Guide, in which additional material can be expected at a later date. It is made available in its current form for your convenience.

Appendix MongoDB access from the command line.

Using more than one hub at a time: If you want to be able to perform queries on any one of the redundant hosts, not only the master, then it's necessary to set a flag on each of the running mongo servers (master and slave):

```
host$ mongo

rs.slaveOk();
```

This ensures that updates are synchonrized for querying. e.g. In this example we see the first query fail, and the second succeed after setting the flag:

```
CFEngineNova:SECONDARY> use cfreport
switched to db cfreport

CFEngineNova:SECONDARY> db.notebook.find();
error: { "$err" : "not master and slaveok=false", "code" : 13435 }

CFEngineNova:SECONDARY> db.notebook.find();
CFEngineNova:SECONDARY> rs.slaveOk();
not master and slaveok=false
CFEngineNova:SECONDARY> db.notebook.find();
{ "_id" : ObjectId("4e5cd788d5d6b92c00000000"),
"_kh" : "SHA=9e9ad21d192fa...1635",
"_rD" : "0 : 10.10.160.115", "_t" : 1, "n" : [
    {
        "u" : "admin",
        "m" : "This machine is a web server",
        "d" : 319944880
    }
] }
```

Appendix Shutting down Mongo with replication

Anytime you shutdown mongo, it will automatically failover. You can~t specifically tell which node to become primary, but you can use replFreeze and replSetStepDown to alter which is eligible to become primary (small difference).

Basically we would freeze a node from becoming primary, then tell the primary to step down, which leaves only one node (see http://www.mongodb.org/display/DOCS/Forcing+a+Member+to+be+Primary for a good explanation).

However, failover is the easy part. The other half to the action is that all client will receive a new policy_server.dat based on the new primary. This will create quite a flux in the system for a bit as it reconfigure. This will take 10-20 minutes for everyone to stabilize. I would reserve a full failover for when the primary will be down for an extended period of time.

Since the outages you have are fairly quick, I would just shutdown mongo/cfengine (or freeze) the secondaries to prevent a failover. Shutting them down would prevent all client from getting new files, but that shouldn~t be an issue.

If you freeze the secondaries, and leave CFE runnning, the clients will update from the secondaries (they will get a license error).

To freeze the secondaries:

On any node:

```
# mongo -host 'IP OF THE SECONDARY' # or login to the node and run plain mongo.
> rs.status()                       # this will show the status of the replicaSet
> rs.freeze(seconds)                # how many seconds before the host can become primary.
> exit
```

Set the freeze for several hours (for four hours - 'rs.freeze(14400)'). Then shutdown cfengine on the primary as normal (/etc/init.d/cfengine3 stop). When you are done with the change, bring up cfengine (cf-agent -f failsafe.cf), then set the freeze time for 0 seconds to unfreeze it ('rs.freeze(0)').