A Tiny Overview of Cfengine: Convergent Maintenance Agent

Mark Burgess

Oslo University College, Norway

Abstract. Cfengine is a widely used software tool with an on-going research project, looking at distributed system administration. System administration deals with the setup, configuration and maintenance of computing devices in a network, a task where it is natural to apply methods of automation. Since its inception in 1993, the cfengine tool-set has been adopted by a broad range of users from small businesses to huge organizations[1]. It is currently running on close to a million nodes around the world.

1 Introduction

Cfengine falls into a class of approaches to system administration which is called policy-based configuration management [2]. Instead of providing detailed imperative programs for software agents to follow, policy based management is about painting the broad strokes, or placing limits on the behaviour of self-adapting agents. A cfengine agent has expert knowledge and a set of tools to configure and repair systems according to a declared policy.

Cfengine's task is to configure the files and processes running on networked computers, e.g. Unix or Windows workstations.

- *Policy* (P) is a description of intended host configuration. It comprises a partially ordered list of operations or tasks for an agent to check.
- *Operators* ($\hat{O}$) or primitive skills/*actions* are the commands that carry out maintenance checks and repairs. They are the basic sentences of a cfengine program. They describe *what* is to be constrained.
- *Classes* are a way of slicing up and mapping out the complex environment into discrete ('digital') regions that can then be referred to by a symbol or name. They are formally constraints on the degrees of freedom available in the system parameter space. They are an integral part of specifying rules. They describe *where* something is to be constrained.
- A cfengine *state* is a fuzzy region within the total system parameter space. It is defined formally with symbols *classes* that define the environment in which a policy rule lives and by the specificity of the policy rules themselves with respect to the internal characteristics of the operators (e.g. file permissions, process characteristics). States have the form: `(address,constraint) = (class,values)`

Burgess M. (2005).
A Tiny Overview of Cfengine: Convergent Maintenance Agent.
In *Proceedings of the 1st International Workshop on Multi-Agent Robotic Systems*, pages 183-188
DOI: 10.5220/0001159201830188

Rather than assuming that transitions between states of its model occur only at the instigation of an operator, or at at the behest of a protocol, cfengine imagines that changes of state occur unpredictably at any time, as part of the environment to be discovered[3]. Cfengine holds to a set of principles, referred to as the *immunity model* [4], for seeking correctness of configuration. These embody the following features:

- Centralized policy-based specification, using an operating system independent language.
- Distributed agent-based action; each host agent is responsible for its own maintenance.
- Convergent semantics encourage every transaction to bring the system closer to an 'ideal' average-state, like a ball rolling into a potential well.
- Once the system has converged, action by the agent desists.

A 'healthy state' is defined by reference to a local policy. When a system complies with policy, it is healthy; when it deviates, it is sick. Cfengine makes this process of 'maintenance' into an error-correction channel for messages belonging to a fuzzy alphabet [5], where error-correction is meant in the sense of Shannon [6].

The main components of cfengine are:

- A central repository of policy files, which is accessible to every host in a domain.
- A declarative policy interpreter (cfengine is not an imperative language but has many features akin to Prolog [7]).
- An active agent which executes intermittently on each host in a domain.
- A secure server which can assist with peer-level file sharing, and remote invocation, if desired.
- A passive information-gathering agent which runs on each host, assisting in the classification of host state over persistent times.

2 Classes, Environment and States

Setting configuration policy for distributed software and hardware is a broad challenge, which must be addressed both at the detailed level, and at the more abstract enterprise level. Cfengine is deployed throughout an environment and *classifies* its view of the world into overlapping sets. Those tasks which overlap with a particular agent's world view are performed by the agent.

A class based decision structure is possible because each host knows its own name, the type of operating system it is running and can determine whether it belongs to certain groups or not. Each host which runs a cfengine agent therefore builds up a list of its own attributes (called the classes to which the host belongs). Some examples include:

1. The identity of a machine, including hostname, address, network.
2. The operating system and architecture of the host.
3. An abstract user-defined group to which the host belongs.
4. The result of any proposition about the system.
5. A time or date.

6. Logical combinations of any of the above.

The environment is large and complex and we cannot describe it in precise terms, so cfengine classifies it into coarse abstract properties that are suitable for management purposes. The classifiers form a patchwork covering of the environment.

Given that the agent, running on a host, can determine the class attributes for that environment, it can now pick out what guidelines it needs from a globally specified policy, since each policy task is also labelled with the classes to which it applies. This policy predicates the agent's application of skills according to broad criteria, encompassing distributed collaborations.

A command or action is only executed if a given host is in the same class as the policy action in the configuration program. There is no need for other formal decision structures, it is enough to label each statement with classes. For example:

```
   linux:: linux-actions
solaris:: solaris-actions
```

More complex combinations can perform an arbitrary covering of a distributed system [8], e.g.

```
   AllServers.Hr22.!exception_host::
actions
```

where `AllServers` is an abstract group, and `exception_host` is a host which is to be excluded from the rest. Classes thus form any number of overlapping sets, which cover the coordinate space of the distributed system (h, c, t), for different hosts h, with software components c, over time t. Classes sometimes become active in response to situations which conflict with policy.

The inherent unknowability of the host environment means that cfengine does not operate with any single notion of state; it has effectively several template definitions. Administrators do not use the same mental model to describe network arrival processes as they do the permissions of files, even though the essential nature of maintenance is the same.

A state is defined by policy. The specification of a policy rule is like the specification of a coordinate system (a scale of measurement) that is used to examine the compiance of the system. The full policy is a patchwork of such rules, some of which overlap. A cfengine state does not appear as a digital string, but rather as a set 'language' classes[9], often represented in the form of a number of regular expressions, that place bounds on

- Characterizations of the configuration of operating system objects (cfagent digital comparisons of fuzzy sets).
- Numerical counts of environmental observations (cfenvd counts or values with real-valued averages).
- The frequency of execution of closed actions (cfagent locking).

3 Policy and Convergence

The view of policy taken in ref. [3] is that of a series of instructions that summarizes the *expected* behaviour. The precise behaviour is not enforcable, so there is no sense in trying to specify it at each computational timestep.

This is where the split between system and environment has a fundamental conceptual bearing on our description of it. There are two kinds of normality that pertain to:

- Properties that we feel confident in deciding for ourselves (permissions of files, processes etc). These are decided and enforced. Deviations from these 'digital' specifications can be repaired or warned about directly by Shannon-like error correction.
- Properties that are controlled by the environment and must be learned (number of users logged in, the level of web requests). These have fluctuating values but might develop stable averages over time. These cannot normally be 'corrected' but they can be regulated over time (again this agrees with the maintenance theorem's view of average specification over time).

Cfengine deals with these two different realms differently: the former by direct language specification and the latter by machine learning and by classifying (digitizing) the arrival process.

The Shannon communication model of the noisy channel has been used to provide a simple picture of the maintenance process [5]. Maintenance is the implementation of corrective actions, i.e. the analgoue of error correction in the Shannon picture. Maintenance appears more complex than Shannon error correction, however. What makes the analogy valid is that Shannon's conclusions are independent of a theory of observation and measurement. For alphabetic strings, the task of observation and correction is trivial.

To view policy as digital, one uses the computer science idea of a language [9]. One creates a one-to-one mapping between the basic operations of cfengine and a discrete symbol alphabet. e.g.

```
A -> ''file mode=0644''
B -> ''file mode=0645''
C -> ''process email running''
```

Since policy is finite, in practice, this is denumerable. In operator language, the above action might be written:

$$\hat{O}_{\text{file}}(\text{name}, \text{mode}, \text{owner}) \tag{1}$$

The transmission medium in this process is time itself. We regard the system as being propagated from its current location to exactly the same place, over time. In other words, the time development of the system is just the transmission of the system into the future over no distance.

Cfengine introduced the notion of 'convergence' into system administration. This was orginially only implicit in the early work, but was named explicitly in the Computer Immunology essay in [10] and was immediately taken up by Couch et al [7] and formed the basis of the configuration management workshops. This concept was quickly understood to be important.

A key part of avoiding uncontrolled behaviour are cfengine's transaction locks [11]. These were designed to ensure three things:

- Consistency of the outcome of atomic operations, i.e. avoid contention due to concurrent execution of multiple agents.

– To limit the frequency with which operations could be repeated.
– To ensure that operations would not be able to hang indefinitely.

Behind these, is the assumption that new cfengine agents will be spawned frequently to check for maintenance operations.

Cfengine uses the idea of *convergence* to an ideal state. This means that, no matter how many times cfengine is run, its state will only get closer to the ideal configuration. This is a stronger condition than *idempotence* as in Couch's interpretation [12, 13]. Since idempotence requires only $\hat{O}^2 = \hat{O}$, while convergence is relative to a specific policy state q_0 [14]:

$$\begin{aligned} \hat{O}q &= q_0 \\ \hat{O}q_0 &= q_0. \end{aligned} \tag{2}$$

The point of convergence over multiple runs is that multiple orthogonal, convergent operations will always lead to the correct configuration, no matter which part of the configuration is incorrect, or in what order things occur. Complex operations might not complete within a single scheduled iteration, if external factors intervene in an untimely manner; but they will always converge eventually. This is proven in ref.[4].

If two operations are *orthogonal*, it means that they can be applied independently of order, without affecting the final state of the system. The construction of a consistent policy compliant configuration has been subject to intense debate[4, 15, 13].

A little-discussed but relevant part of the ordering problem is the matter of cfengine's adaptive transaction locking [11]. The transaction locks allow cfengine processes to 'flow through' one another and avoid going into infinite regression and also prevents agents from repeating themselves too often, or getting stuck on a problem. If an agent gets stuck, another one will destroy it and take over.

4 Anomalies

In cfengine, an extra daemon (cfenvd) is used to collect statistical data about the recent history of each host (approximately the past two months), and classify it in a way that can be utilized by the cfengine agent. The agent learns. Data are gradually aged so that older values become less important [16]. The daemon automatically adapts to the changing conditions, but has a built-in inertia which prevents anomalous signals from being given too much credence. Persistent changes will gradually change the 'normal state' of the host over an interval of a few weeks. Unlike some systems, cfengine's training period never ends. The challenge of future anomaly detection is the find a stochastic anomaly language for a reactive agent policy.

References

1. Burgess, M.: Evaluation of cfengine's immunity model of system maintenance. Proceedings of the 2nd international system administration and networking conference (SANE2000) (2000)

2. Sloman, M., Moffet, J.: Policy hierarchies for distributed systems management. Journal of Network and System Management **11** (1993) 1404
3. Burgess, M.: On the theory of system administration. Science of Computer Programming **49** (2003) 1
4. Burgess, M.: Cfengine's immunity model of evolving configuration management. Science of Computer Programming **51** (2004) 197
5. Burgess, M.: System administration as communication over a noisy channel. Proceedings of the 3nd international system administration and networking conference (SANE2002) (2002) 36
6. Shannon, C., Weaver, W.: The mathematical theory of communication. University of Illinois Press, Urbana (1949)
7. Couch, A., Gilfix, M.: It's elementary, dear watson: Applying logic programming to convergent system management processes. Proceedings of the Thirteenth Systems Administration Conference (LISA XIII) (USENIX Association: Berkeley, CA) (1999) 123
8. Comer, D., Peterson, L.: Understanding naming in distributed systems. Distributed Computing **3** (1989) 51
9. Lewis, H., Papadimitriou, C.: Elements of the Theory of Computation, Second edition. Prentice Hall, New York (1997)
10. Burgess, M.: Computer immunology. Proceedings of the Twelth Systems Administration Conference (LISA XII) (USENIX Association: Berkeley, CA) (1998) 283
11. Burgess, M., Skipitaris, D.: Adaptive locks for frequently scheduled tasks with unpredictable runtimes. Proceedings of the Eleventh Systems Administration Conference (LISA XI) (USENIX Association: Berkeley, CA) (1997) 113
12. Couch, A., Sun, Y.: On the algebraic structure of convergence. Submitted to DSOM 2003 (2003)
13. Couch, A., Sun, Y.: On observed reproducibility in network configuration management. Science of Computer Programming **(to appear)** (1994)
14. Burgess, M.: Analytical Network and System Administration — Managing Human-Computer Systems. J. Wiley & Sons, Chichester (2004)
15. Traugott, S.: Why order matters: Turing equivalence in automated systems administration. Proceedings of the Sixteenth Systems Administration Conference (LISA XVI) (USENIX Association: Berkeley, CA) (2002) 99
16. Burgess, M.: Two dimensional time-series for anomaly detection and regulation in adaptive systems. IFIP/IEEE 13th International Workshop on Distributed Systems: Operations and Management (DSOM 2002) (2002) 169