

An brief overview of the implementation of principles of system administration in cfengine

(Not for publication)

Mark Burgess

June 17, 2004

Abstract

Cfengine is a distributed agent framework for performing policy-based network and system administration, used on hundreds of thousands of Unix-like and Windows systems. This paper describes cfengine's stochastic approach to policy implementation using distributed agents. It builds on the notion of 'convergent' statements, i.e. those which cause agents to gravitate towards an ideal configuration state, which is implied by policy specification. Convergent semantics ensure consistency under dynamical change, and provide a form of transaction control that is robust to unpredictable change. Cfengine's host classification model is briefly described and the model is compared to related work.

1 Introduction

Cfengine is an on-going research project, looking at distributed system administration. 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 an estimated several hundred thousand nodes around the world. Since its inception, cfengine has developed considerably, although the basic framework and key principles have remained the same. Cfengine is written almost exclusively by the author. Minor patches and support have been provided by others.

Cfengine falls into a class of approaches to system

administration which is called policy-based configuration management. To some extent, policy-based configuration can be seen as a reaction to the inadequacies of control and monitoring software, now typified by many Simple Network Management Protocol (SNMP) implementations. Administrative schemes, employing autonomous agents, are the only administrative solutions which scale to large numbers of hosts[2], because this does not rely on the funnelling of configuration instructions through a serial queue, governed by a human. Today, the approach is used by many large organizations to manage anything from a handful to thousands of systems[1].

Over the last five years, development on cfengine has been slowed in order to perform some parallel research, attempting to unravel core issues, previously unaddressed by the research community. These include

- Scheduling ideas
- Game theoretical strategies
- Dependency resolution
- Anomaly detection

Cfengine is a research tool that has been used to develop proof-of-principle system administration methodology in a practical tool. Although cfengine has become widely used and its users demand a high standard of workmanship, the design has never been driven directly by users' wishes. Rather it has been

built around the research carried out mainly here at Oslo University College.

In this review I have tried to describe the concepts and literature behind cfengine.

1.1 Some key nomenclature

The following phrases are used in discussing cfengine.

- Ad hoc: following no predefined pattern. Presently there is the concept of an ‘ad hoc network’ which is a self-organizing (usually mobile) network of peers.
- Alphabet: a set of independent symbols that belong together.
- Arrival process: The arrival of different kinds of events over time is called an arrival process. For instance, the arrival of network traffic, or the arrival of instructions to the computer. Arrival processes are a concept from stochastic analysis and it is generally assumed that the arrival of events is random.
- Autonomous: a host or peer system is autonomous if its policy does not originate from another host or peer.
- Coarse-graining: this is the process of taking a detailed viewpoint and eliminating detail to form coarser, less specific classes. Coarse grains are like black-box versions of parts of a system.
- Digitization, classification: This is a form of coarse-graining in which an observed information stream is reduced into classes of known extent.
- Information, complexity: this has a precise theoretical meaning in terms of the average entropy of a digital observation.
- Graph: a number of nodes connected together by links or edges.

1.2 Cfengine philosophy

What makes cfengine different from similar approaches to configuration management is that it embraces a stochastic model of system evolution. Rather than assuming that transitions between states of its model occur only at the instigation of an operator, or at the behest of a protocol, cfengine imagines that changes of state occur unpredictably at any time.

The focus of cfengine, and supporting work[3], is this willingness to accept the idea of increasing random entropy of configuration through interaction. Users’ social behaviour[4] is seen as a central and unignorable mixture of signals which tends to disorder the system configuration, over time. Cfengine holds to a set of principles, referred to as the *immunity model*[5], for seeking correctness of configuration. These embody the following features:

- Centralized policy-based specification, using an operating system independent language, which conceals implementation details.
- Distributed agent-based action, in which every host node 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, or more usually, does not even start at all, when convergence was assured on a previous run of the agent.

The last two points are the most important. Most configuration agents either require a human to initiate change or rewrite the same constant configuration many times. In an analogous way to the healing of a body from sickness, cfengine’s configuration approach is to always move the system closer to a ‘healthy’ state[2], or oppose unhealthy change: hence the name ‘immunity model’. This idea shares several features with the security model proposed in refs. [6, 7]. Convergence is described further below.

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[8], where error-correction is meant in the sense of Shannon[9].

In ref. [3] it was shown that a complete specification of policy determines an approximate configuration of a software system only approximately over persistent times. There are fundamental limits to the tolerances one can expect a system to satisfy with respect to policy compliance. A policy does not meaningfully ensure a precise configuration over microscopic intervals, even when the policy is repeatedly enforced, because every system is coupled to an environment of users, whose effect on the system is unpredictable. Fluctuations in the policy configuration can always occur over shorter times. They can be corrected, provided sufficient vigilance is maintained, but such fluctuations must inevitably occur for however brief an interval. Cfengine expects this average ideal state to be maintained only through such constant appraisal and adjustment, rather than assuming that a one-off implementation of policy suffices to place software into a persistent configured state.

1.3 Components of the cfengine framework

The main components of cfengine are (see table 1 and fig. 1):

- 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[10]).
- 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.
- Various supporting tools.

Cfengine employs centralized declaration of policy, with a democratic, distributed implementation. Once boot-strapped with a basic configuration, each agent decides, based on its current policy whether it wishes to import or update its policy specification from a trusted source. Cfengine is explicitly deaf to external commands, except for one generic command to re-check its state, using an existing policy. This command is protected from malicious, repetitive execution (spamming) attempts by an adaptive locking scheme[11].

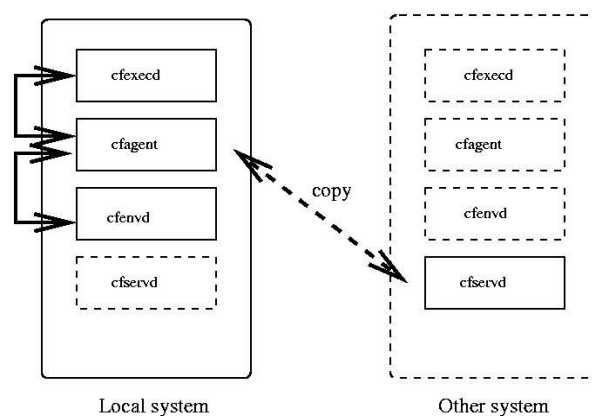


Figure 1: Cfengine components

1.4 Declarative language

The fact that cfengine uses a *declarative language* for describing policy is often a source of confusion to newcomers who are more familiar with imperative scripting languages such as Perl. Cfengine has to be far more practical than theoretical languages like Prolog however and this has presented many challenges to the design and implementation of the language, particularly in the evaluation of iterated tasks.

Component	Cfengine 1.x	Cfengine 2.x
Agent	cfengine	cfagent
Server	cf	cfserverd
Scheduler	cron,cfwrap	cfexecd
Poller	cfrun	cfrun
Key Gen	cfkey	cfkey
Long term state	-	cfenvd
State grapher	-	cfenvgraph

Table 1: Components in cfengine

It should be regarded as a principle that the language of cfengine that it should prevent administrators from coding non-convergent (non-maintainable) policies. By not releasing imperative control to policy writers, cfengine is able evaluate policy in a safe and convergent manner in the widest range of circumstances.

A rule such as this:

control:

```
list = ( host1,host2,host3 )
```

admit:

```
/etc/passwd $(list)
```

is quite intuitive, but the following example is not so clear:

control:

```
list = ( host1,host2,host3 )
perms = ( 0644,0645,0700 )
```

copy:

```
/etc/passwd
dest=/tmp/passwd
server=$(list)
mode=$(perms)
```

Do the different values of “list” correspond to the different values of “perm” or should one take each value

of perm for each value of list etc. There are many ambiguities in declarative syntax when complex behaviour is specified. Cfengine goes to some lengths to make such combinations safe and intuitive, but more work is needed in this area.

1.5 Key ideas in this text

- *Policy* (P) is a description of intended host configuration. It comprises a partially ordered list of operations or actions. Cfengine policy is that part of system policy that can be coded into the host itself.
- *Operators* (\hat{O}) or primitive *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). State definitions often contain wildcards or value-widths (e.g. measured in standard deviations). e.g. $q_1 = (\text{solaris,mode}(/etc/passwd,0644) = \text{true})$

Let us elaborate on these ideas.

1.6 Classes and environment

The distributed nature of today’s software systems mixes local and remote notions freely and in a variety of ways. Setting configuration policy for such an

array of software and hardware is a broad challenge, which must be addressed both at the detailed level, and at the more abstract enterprise level. Cfengine uses the idea of host classification to dissect a distributed environment into overlapping sets.

A class based decision structure is possible because a cfengine agent is run by every host on the network individually. 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). Classes that are meaningful in the context of a particular host 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. A randomly chosen strategy element.
7. The logical combination of any of the above, with AND (`.`), OR (`|`), NOT (`!`) and parentheses.

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. One can think of this as a projection of the environment onto a discrete set of abstract classifiers. The classifiers form a patchwork covering of the environment (see fig 2). This simple viewpoint is only possible if the system is sufficiently linear to admit a stable classification. The periodic projection of system management described in refs.[4, 12] ensures this.

Given that the agent running on a host can determine the class attributes for the host, it can now pick out what it needs from the global policy, since policy criteria are also labelled with the classes to which

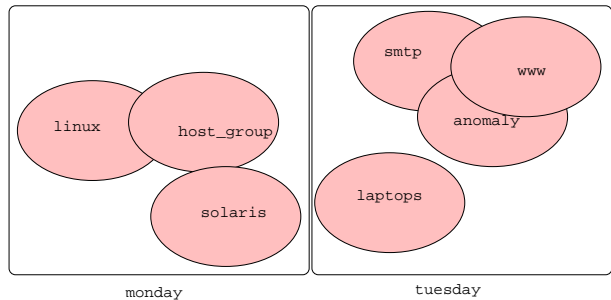


Figure 2: Overlapping classes form a covering of the environment parameter space.

they apply. Figure 3 illustrates how cfengine classifies the environment of a host. 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 formal decision structures, it is enough to label each statement with classes. At the simplest level, one has commands belonging only to a single class, say the operating system type of the hosts:

```
solaris::
    actions

sunos_5_8::
    actions
```

More complex combinations can perform an arbitrary covering of a distributed system[13], e.g.

```
AllBinaryServers.Hr22.OnTheHour.!exception_host::
    actions
```

where `AllBinaryServers` is assumed to be 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,

Class predication allows policy to encompass many-to-one maps and one-to-many maps. Many

GNU Configuration Engine - 2.1.7a1
Free Software Foundation 1994-
Donated by Mark Burgess, Faculty of Engineering,
Oslo University College, 0254 Oslo, Norway

Host name is: nexus
Operating System Type is sunos
Operating System Release is 5.9
Architecture = sun4u
Using internal soft-class solaris for host solaris
The time is now Sun Jun 13 21:58:47 2004

Defined Classes = (128_39_89 128_39_89_10
2001_700_700_3_a00_20ff_fe9b_dd4a 32_bit AllBinaryServers
AllHomeServers CfPubKeys Day13 ElseWhen FTPservers Hr22 Hr22_Q1 June
MailHub Min00 Min00_05 NameServers OnTheHour PasswdServer PrimeServers
Q1 SambaServers Sunday WWWServers Yr2004 anomaly_hosts any cfengine_2
cfengine_2_1 cfengine_2_1_7a1 compiled_on_solaris2_9
entropy_cfengine_out_low entropy_dns_in_low entropy_dns_out_low
entropy_ftp_in_low entropy_ftp_out_low entropy_icmp_in_low
entropy_icmp_out_low entropy_irc_in_low entropy_irc_out_low
entropy_misc_in_low entropy_misc_out_low entropy_netbiosdgm_in_low
entropy_netbiosdgm_out_low entropy_netbiossns_in_low
entropy_netbiossns_out_low entropy_netbiosssn_out_low
entropy_tcpack_in_low entropy_tcpack_out_low entropy_tcpfin_in_low
entropy_tcpfin_out_low entropy_tcpsyn_in_low entropy_tcpsyn_out_low
entropy_udp_in_low entropy_udp_out_low entropy_www_out_low
entropy_wwws_out_low fe80__a00_20ff_fe9b_dd4a ipv4_128 ipv4_128_39
ipv4_128_39_89 ipv4_128_39_89_10 longjob net_iface_hme0 net_iface_lo0
nexus nexus_iu_hio_no peer_group3 securehosts solaris sparc sun4u
sunos_5_9 sunos_sun4u sunos_sun4u_5_9
sunos_sun4u_5_9_Generic_112233_11)

Figure 3: Example of cfengine environmental class determination

hosts can belong to the same class, and therefore policy actions can be common to many hosts. Similarly, each host can be characterized by many different classes or attributes which label its intended state, recognizing the multiple functions of each node in the virtual community, and the distributed nature of software systems.

Additional classes are automatically evaluated based on the state of the host, in relation to earlier times. This is accomplished by the additional *cfenvd* daemon, which learns and continually updates a database of system averages, which characterize “normal” behaviour. The state of the system is examined and compared to the database, and the state is classified in terms of the current level of activity, as compared to an average of equivalent earlier times. e.g.

```
RootProcs_low_dev2
netbiossn_in_low_dev2
smtp_out_high_anomalous
www_in_high_dev3
```

The first of these tells us that the number of root processes is two standard deviations below the average of past behaviour, which might be fortuitous, or might signify a problem, such as a crashed server. The WWW item tells us that the number of incoming connections is three standard deviations above average. The smtp item tells us that outgoing smtp connections are more than three standard deviations above average, perhaps signifying a mail flood. The setting of these classes is transparent to the user, but the additional information is only visible to the privileged owner of the cfengine work-directory, where the data are cached.

2 What is a state?

Due to the fuzzy nature of the policy, and the inherent unknowability of the host environment, cfengine does not operate with any single notion of state; it has effectively several. No single notion of state could suffice, nor would any single notion be adequate from a human point of view to describe what is going on in the system. 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.

As with all things, a state is essentially 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 compliance of the system. The full policy is a patchwork of such rules, some of which overlap.

A cfengine state is seldom a precise digital string, but rather a set of members of a ‘language’ in the computer science meaning[14], often represented in the form of a number of regular expressions, that place bounds on

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

3 A model of system administration

The main theoretical viewpoint underpinning cfengine was outlined in ref. [15] and is fully described in ref. [3]. Although a precise description of the cfengine viewpoint is involved, the idea is rather simple. The author’s textbook *Analytical Network and System Administration*[16] develops many of the notions that underpin this theory systematically and in detail.

The theory behind cfengine attempts to derive a simple ‘unintelligent’ (i.e. non-reasoning) system of fault maintenance for human-computer systems, i.e. systems that are not merely specified by a designer, but that are actually used by an unpredictable assembly of users sitting either locally at a keyboard, or remotely across a network.

Each computer system that is to be managed by cfengine is treated as an autonomous system embed-

ded in an environment (see fig. 4). It is important here to divide the system into two parts: host and environment because these are fundamentally different representations of processes that occur in the human-computer system.

A random or stochastic system can be defined to be one in which the amount of information needed to describe it is much greater than the amount of information that we can observe. We say that observations of such a system are random because we cannot encompass sufficient information to explain everything that happens causally. The environment of a computer is a stochastic system. The behaviour of a host, on the other hand, is governed by a relatively simple computer program, with easily digitizable content. The level of complexity or information[17] in the computer is much less than that of the environment.

This distinction between the impulses that a computer receives from its environment and those that are programmed into it means that we must form a hybrid model for describing the changes that occur in a computer system.

In reference [3], a computer policy is described as a specification of what we would like a human-computer system to do, from the viewpoint of the computer. Implicit in this definition is the projection of a complex human-computer system onto a simplistic digital computer system. We cannot therefore treat the computer system as a rule-based deterministic automaton, because we cannot predict the input from users and network. We have two options: we can isolate the computer from all random input (by making it a closed batch device), or we are forced to deal with the problem of stochastic change within the digital system. Since system administration is about human-computer interaction, we take the latter view.

4 Policy and its interpretation

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

Cfengine does not attempt to provide a complete

description of system policy. It deals with a specific problem: how to configure the alphabetic properties of the system of a single host, given that we cannot fully predict what changes are taking place. This, in turn, can affect the human aspects of policy through access control and other behavioural constraints.

Policy is a really description about what we consider to be normal. A description of normality is a decision about how we define errors. If we cannot equate normality with policy then we have not even a partially predictable system to manage and the concept of ‘management’ would be meaningless.

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.

Learning an environmentally controlled state requires extra processing. First, the environment daemon cfenvd collects data and learns the normal state of the system using machine learning methods, then the state of the system is measured relative to the learned average state.

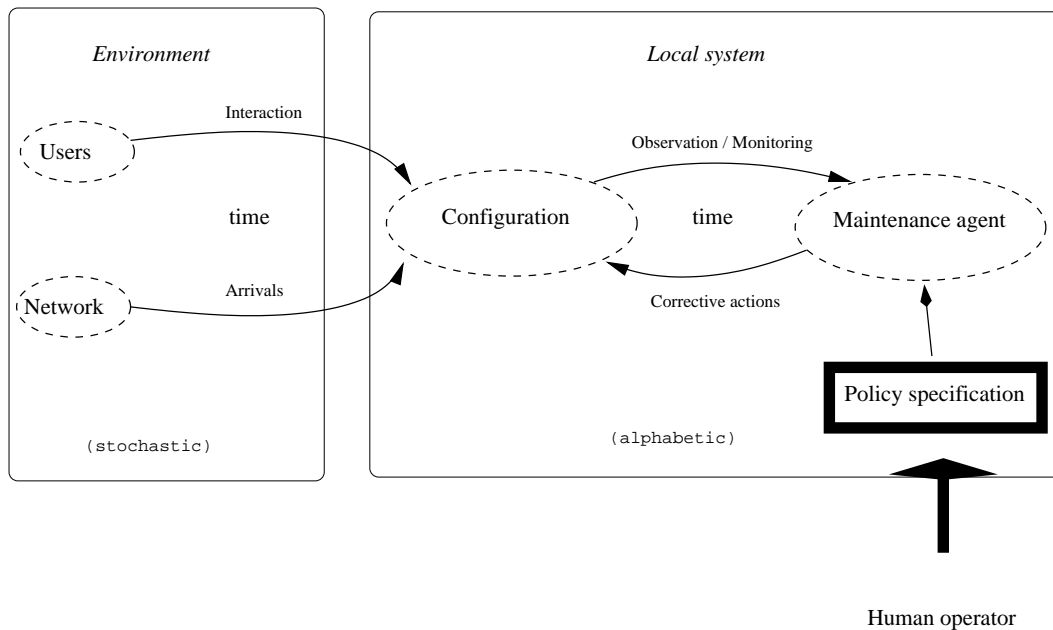


Figure 4: A schematic view of the management model.

5 Feedback and error regulation loops

The Shannon communication model of the noisy channel has been used to provide a simple picture of the maintenance process[8].

The concept of maintenance was introduced to discuss the error correction process[3]. Essentially, maintenance is the implementation of corrective actions, i.e. the analogue of error correction in the Shannon picture. Maintenance is rather more complex than Shannon error correction, however, since it is not immediately clear that there is a simple digital picture of information for a system policy.

What makes the analogy valid is that Shannon's conclusions are independent of a theory of observation and measurement that becomes essential for policy. For simple alphabetic strings, the task of observation and correction is trivial. However, the conclusions apply even for more complicated models of observation (monitoring) and correction because the conclusions do not depend on the nature of these ac-

tions.

A necessary and sufficient characterization of digital policy is provided by the computer science idea of a language[14]. Defining policy in language theoretical terms is a task that remains for future research. All we need to do this is to create 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'

The agent interprets and translates the policy symbols into actions through operations, also in one to one correspondence.

1. Cfagent observes : X
2. Policy says : $X \rightarrow A$
3. Agent says : $A \rightarrow \hat{O}_{\text{file}}(\text{passwd}, 0644, \text{root})$

The alphabet might appear infinite, on causal reflection, in the sense that data objects that parameterize operations tie basic operations to a relativistic system – not to a fixed system, as there must be an infinite variety of possible names and combinations of values. These are indeed potentially infinite across an infinite population of systems, however cfengine is used in a single real system where all policies are actually finite. In other words, although the space of all possible policies is potentially very very large (though never truly infinite due to finite memory etc), only a small fraction of the possibilities is ever realized on a real system and this problem is not a limitation.

There is a larger point here: this is indeed what we mean by management. If we cannot reduce policy to a finite number of assertions and tasks then we have effectively lost control of the system and the idea of management becomes meaningless. We can define manageability as the ratio[16]

$$\mathcal{M} = \frac{\text{Information in environment}}{\text{Information in policy}} \quad (1)$$

It can therefore be assumed that the number of realistic policies and desirable operations is finite and that these can be defined and enumerated into a set of alphabetic operations.

These presumed and observed states of the system feed into the definition of the policy[5, 18]. However, if one defines the operations into classes $\hat{O}_1, \hat{O}_2, \dots$ etc, then these form a strict alphabet of ‘black box’ objects. The fuzziness in the operations can be eliminated by introducing a new symbol for each denumeration of the possible parameters. Here is an example operation, simplified for the purpose of illustration.

files:

```
/etc/passwd mode=0644 owner=root
```

This is actually a special instance of

files:

```
<filename> mode=<permissions> owner=<username>
```

which tells the agent to assert these properties of the named file object. Since there is a finite number of

files and permissions and users, there is no impediment to listing every possible permutation of these and assigning a different alphabetic symbol to them. In operator language, the above action might be written:

$$\hat{O}_{\text{file}}(\text{name}, \text{mode}, \text{owner}) \quad (2)$$

Let us suppose that example above evaluates to the alphabetic symbol ‘A’. When the agent observes these properties of the named object it comes up with a symbol value based upon what it has measured. Suppose now that a user of the system (who is formally part of the environment) accidentally changes the permissions of the password file from mode=0644 to mode=0600. Moreover, we can suppose that this new value evaluates to the alphabetic character ‘X’.

The transmission medium in this process is time itself. We regard the system (as is normal in the physical sciences) 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.

The primitive nature of the basic cfengine operations makes this set of operations fall into the following categories:

- Orthogonal (non-overlapping) operations that are strictly independent alphabetic atoms.
- User defined operations, outside of cfengine’s framework, that cannot be guaranteed to be independent and might therefore ‘overlap’ with other alphabetic symbols.
- Operations entirely outside of cfengine’s jurisdiction.

Since there is nothing we can do about overlapping objects that fall outside of cfengine’s jurisdiction, no attempt is made to solve this problem beyond wrapping these operations in independent transaction locks. This provides a certain level of predictability that improves the possibility of debugging any contradictions that occur due to overlap of symbol operational content.

This point is rather subtle and it is handled in cfengine by encouraging users to use cfengine’s primitive mechanisms exclusively whenever possible. Users who mix their own programs inside cfengine policy could experience contention. The extent to which this causes problems for some users is unknown. It has not proven to be a problem for any system which the author is familiar.

6 Configuration management and convergence

Cfengine introduced the notion of ‘convergence’ into system administration. This was originally only implicit in the early work, but was named explicitly in the Computer Immunology essay in [2] and was immediately taken up by Couch et al[10] and formed the basis of the configuration management workshops. This concept was quickly understood to be important.

A key part of making cfengine practically safe, and avoiding some of the problems of ordering and run-away execution that are implicitly absent from a convergent process, 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.

They form a wrapper that increases the certainty that a reparative action can actually be performed within a reasonable time frame. Behind them is the assumption that the cfengine agent will be run frequently.

One would like to secure the property that changes made to a configuration move towards a definite state, terminate after a small number of iterations, and that the route taken back towards the ideal state is unique and unidirectional. If this were not the case, then contradictions and non-terminating cycles

would result. We require there to be absorbing states, or for operations to behave like semi-groups[19, 16].

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[19, 18].

Note that the point of convergence over multiple runs is not the number of runs required to achieve convergence, but 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.

Property 1 *Regardless of how long it takes, or of the scheduling order, a configuration will be implemented without requiring procedural logic. Operations are thus self-ordering, as long as all of the operations are convergent and orthogonal. This provides a notion of atomicity, and transactional security.*

Cfengine addresses convergence in two ways: by making each successful operation convergent in a single step, and by checking for contrary sequences. If a single step should fail or be undermined, for whatever reason (crash, interruption, changing conditions, loss of connectivity etc), it can be repeated later; this is sufficient to ensure that simple configurations converge. We now consider how this works.

If two operations are *orthogonal*, it means that they can be applied independently of order, without affecting the final state of the system. Using a linear representation of vectors and matrix valued operators, this is equivalent to requiring their commutativity.

7 The debate about ordering

A language is an ordered string of symbols. The basic syntactically correct strings of a language might then be ordered or not, in a greater whole. Is policy an ordered set of such strings? Does it have meta-structure.

The construction of a consistent policy compliant configuration has been subject to intense debate since it was understood that cfengine actions are not generally order dependent[5, 20, 18].

The identification of convergence with order-free configuration led S. Traugott to formulate an alternative philosophy which he referred to as ‘congruence’ at the 2001 cfengine workshop in San Diego. The concept was the opposite of the cfengine view and suggests that extreme ordering is the answer to reliability. Instead of always being able to find a path to the correct configuration from a given configuration, Traugott proposes simply destroying a faulty machine and building it up from scratch, like the differentiation of a biological stem cell.

The argument about the necessity of ordering has been successfully refuted in [5, 18], with certain qualifications. The two approaches can both be made to work, but only the convergent approach can be used for realtime maintenance.

A little discussed but relevant part of the ordering problem is the matter of cfengine’s transaction locking[11]. The transaction locks allow cfengine processes to ‘flow through’ one another and avoid going into infinite regression.

8 Strategies

Game theory has been suggested as a way of optimizing decisions about system policy[3, 16]. Cfengine can be thought of as a gaming agent that plays in a game against system ‘gremlins’, always trying to do their worst. Sometimes these gremlins are motivated users, at other times they might be the forces of chance.

Often a player can benefit by using a variety of tactics, in a *mixed strategy*, so as to either confuse the opponent, or simply seek a compromise between multiple aims. Game theory predicts that there is not always a pure strategy (policy) that is the optimum solution to a problem. Sometimes a random mixture of policies over time can be most effective. In cfengine, the concept of a strategy is like that of a mixed game strategy. A strategy is a set of classes, each of which is used to label a strategy for configu-

ration management. A strategy definition looks like this:

```
strategies:
  { spread_load
    percent_10: "1"
    percent_30: "3"
    percent_60: "6"
  }
```

This declaration defines one and only one of the set of classes `percent_10`, `percent_30`, `percent_60`, on each invocation of the agent `cfagent`. The probability that the class will be defined is determined by the quoted integers. The sum of the integers in a strategy is used to define a total which divides each of the integers, thus leading to a fraction between zero and one, which is the probability with which the classes should be defined. A Monte Carlo (metropolis) algorithm is then used to select one of the classes at random.

The name of the strategy is used only as a convenience; it is not referred to anywhere else, but it does identify the elements as being part of an entity. The identifier can be used to modify the strategy with class-based exceptions. It is possible, for instance, to insert an additional element on a special host,

```
strategies:
  # ...
  SpecialHost::
  { spread_load
    dominant_strategy:: "10"
  }
```

though the addition of this class would cause the percentages to be recalculated, so that the percent names above would cease to be correct (the total would then be $1 + 3 + 6 + 10 = 20$, giving fractions $1/20, 3/20, 6/20, 10/20$, or $5\%, 15\%, 30\%, 50\%$).

Once a strategy element has been selected, at random, the element is then treated as an ordinary class. For example:

```
processes:

Nameservers.percent_10::

    "named" signal=term
        restart "/local/sbin/named -u dns"
```

This rule kills and restarts the *named*, nameserver daemon ten percent of the time. This turns out to be an effective strategy against the daemon crashing, due to an internal error (presumably a memory leak). To randomize the occurrence of a particular action in time, one could try a strategy like the following:

```
tidy:

percent_10.Hr16::

    tidy rules.....

percent_30.Hr20::

    tidy rules.....
```

This would ensure that tidying was performed 10 percent of the time, at 4 p.m., while thirty percent of the time at 8 p.m. (but possibly both). This application of strategies can be used for load spreading, for instance. Randomization of time can be applied to backups, integrity checks, checks on connections, log rotation, and so on.

Other applications could include randomizing links to actual binaries, to thwart intrusion attempts on systems which require high security and have high risk. In the configuration policy were secret, one could also randomize the contents of key variables for security purposes: location of backup host, choice of filesystem to check, location of key programs (e.g. link to alternative versions e.g. passwd or ps, which are often the object of Trojans). This is essentially a temporary diversion based on “security through

obscurity”; however, since it is dynamical and random, rather than fixed, it is possible to win some effect from this game-theoretically. This kind of tactic might be sufficient to delay the efficacy of an intrusion for long enough to detect it and find a more permanent fix.

9 Search Filters

Large scale configuration control and regulation often requires searching through files or processes for very specific faults. Tools for performing this kind of search have been lacking. On Unix systems, advanced file searches can be accomplished with the `find` command, but the syntax of this command is clumsy and difficult to incorporate into complex rules.

In cfengine, the concept of a filter was introduced in order to pick out specific files or processes in a search. A filter is a way of selecting or pruning during a search over files or processes. Since filter rules could apply to several objects, cfengine allows you to define filter conditions as separate objects to be applied in different contexts. In a sense, filters classify the files found during a search, in much the same way that class attributes classify hosts in a domain.

Filter objects can be used in `copy`, `editfiles`, `files`, `tidy` and `processes`. In most cases one writes

```
.. filter=filteralias
```

in the appropriate command. The exception is `editfiles`, where the syntax is

```
{
..
Filter "filteralias"
..
}
```

Example:

```
files:

/tmp filter=testfilteralias action=alert r=inf
```

Filters are defined in a separate section. Filters for files and processes are defined together. They differ only in the criteria they contain. e.g.

```
FromCtime: "date(2000,1,1,0,0,0)"
ToCtime: "now"
```

```
FromMtime: "tminus(1,0,0,2,30,0)"
ToMtime: "inf"
```

Examples: processes started between 18th Nov 20 and now.

```
{ filteralias

FromSTime: "date(2000,11,18,0,0,0)"
ToSTime: "now"
}
```

All processes which have accumulated between 1 and 20 hours of CPU time.

```
{ filteralias

FromTTime: "accumulated(0,0,0,1,0,0)"
ToTTime: "accumulated(0,0,0,20,0,0)"
}
```

10 Anomaly Research

There is no system available in the world today which can claim to detect and classify the functioning state of a computer system. Cfengine does not attempt to provide a “product” solution to this problem; rather it incorporates a framework, based on the current state of knowledge, for continuing research into this issue.

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. Data are gradually aged so that older values count less[12, 21].

The current long-term data recorded by the daemon are: number of users, number of root processes, number of non-root processes, percentage disk full for root disk, number of incoming and outgoing sockets for netbiosns, netbiosdgm, netbiossn, irc, cfengine, nfsd, smtp, www, ftp, ssh and telnet. On hosts that have the tcpdump program installed, cfenvd is able to learn a number of network traffic anomalies.

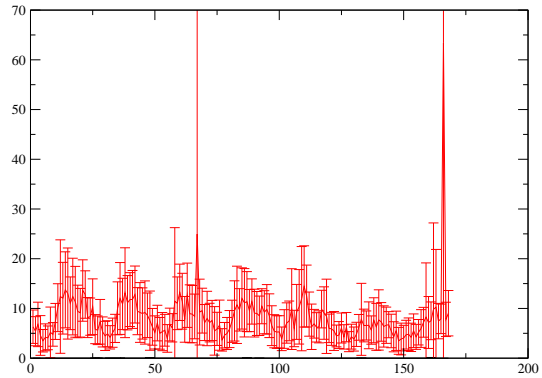


Figure 5: Weekly average (with standard deviation error bars) of incoming web request events.

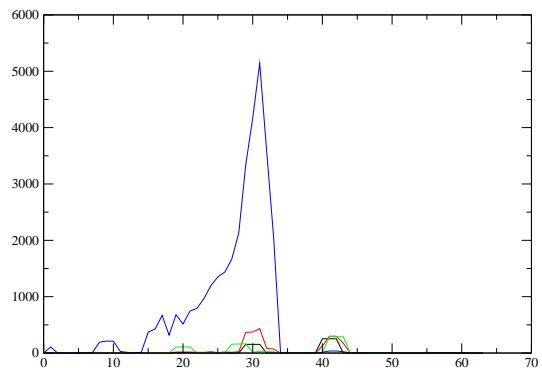


Figure 6: Distribution of outgoing network traffic about mean

These data have been studied previously, and their behaviour is relatively well understood. In future versions, it is expected to extend this repertoire, as more research is done.

The use of the daemon will not be reliable until about six to eight weeks after installing and running it, since a suitable training period is required to build up enough data for stable characterization. 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. It regards normal behaviour as a relative concept, which has more to do with local stability than global constancy.

Cfenvd sets a number of classes in cfengine which describe the current state of the host in relation to its recent history. The classes describe whether a parameter is above or below its average value, and how far from the average the current value is, in units of the standard-deviation (see above). This information could be utilized to arrange for particularly resource-intensive maintenance to be delayed until the expected activity was low.

The cfenvgraph command can be used to dump a graph of averages for visual inspection of the normal state database. The format of the file is

$$t, y_1, y_2, y_3 \dots$$

which can be viewed using *gnuplot* or *xgmr* or other graphical plotting program. This would allow the policy-maker to see what is likely to be a good time for such work (say 06:00 hours), and then use this time for the job, unless an anomalous load is detected. For instance, since significant server activity could place a significant load on a host

copy:

```
Hr06.!(www_in_high_dev3|www_in_high_anomalous)::  
  
/www-user-database dest=/www-backup
```

The concept of load average is not used here, primarily because it is ambiguous and too unspecific.

One way of understanding cfengine’s behaviour in relation to environment and policy is as a Markov fluctuation model of change[22], seeking an equilibrium configuration. Recently, a transaction manager was introduced into cfengine, to cope with the need for long-term, persistent memory (so-called non-Markovian variables). In the original model, only first order Markov processes were available; everything about the system had to be deduced from the environment upon each new invocation. The only exception was the use of adaptive locks, which were used to regulate repetition. That approach has succeeded in solving many problems, but it lacks the ability to track long-term changes to the system, such as seasonal variations and changing patterns of usage. In version 2 of cfengine, classes can be based on long term memory which degrades over a period of approximately a month (the time over which computers may be regarded as being statistically stable[4]). This allows for a more refined statistical sense of anomaly.

The immunity analogy is also useful here. Immunological memory is like a stack of previously combatted problems, which works like an ordered list of changing priorities. This enables repeatedly actual problems to be dealt with more quickly than otherwise. It represents a change in biological policy toward threat. The same problem arises in configuration management, in the face of unpredictable faults or attacks.

The transaction manager collects data, including statistical samples of system performance variables, and an environment daemon collates and classifies these data into regular cfengine classes, which can be used to activate actions to counter emerging problems, or even modify strategy. Experience with such mechanisms is currently limited, but they are a logical generalization of the first order classifications, which allow for more complete knowledge of competition in the face of environmental complexity. The data in such a database, are a natural candidate for analysis, to be fed back into a refinement of policy.

The challenge of future anomaly detection is the find a stochastic anomaly language for all anomalous environmental occurrences. For statistical character-

istics one has the shape of the distribution about the mean, the mean itself and the scales represented by the moments of the distribution. Recently the entropy of this distribution about the mean has been used to register anomalies:

```
anomaly_hosts.icmp_in_high_anomaly
    & !entropy_icmp_in_high::
```

```
ShowState(incoming.icmp)
```

See the result of this rule to an environmental anomaly in fig. 7. This statistical characterization was described fully in [21].

11 Peer services and voluntary cooperation

The autonomy of individual devices is a sovereign principle in cfengine. No outside influence can directly command the agent to change its policy or carry out an action that conflicts with its policy. Cfengine is based on voluntary cooperation with peers. This is especially important for its applicability to the pervasive scenario.

Traditional client server methods are risky for certain administrative tasks. There are three reasons why the traditional model of client-server communication is undesirable for pervasive computing services (see fig. 8).

1. Traditionally, network servers are assumed to be 'available'. Clients and servers do not expect to wait significantly for processing of their requests. This *impatient expectation* leads to pressure on the provider to live up to the expectations of its clients. Such expectation is often formalized for long term relationships as Service Level Agreements (SLA). For ad hoc encounters, this can be bad both for the provider and for the client. Both parties wait synchronously for a service dialogue to complete, during which their resources can easily be tied up and exploited maliciously (so-called Denial of Service or 'spam' attacks).

Non-malicious, random processes can also be risky. It is known from studies of Internet traffic that random arrival processes are often long

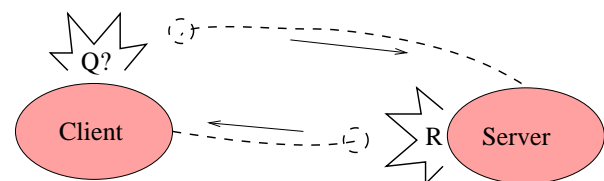
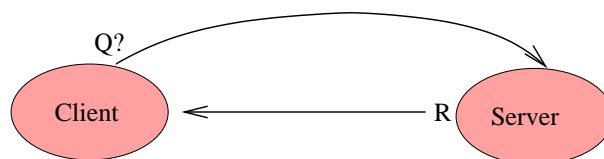


Figure 8: In a traditional client-server model, the client drives the communication transaction. In voluntary RPC, each party controls its own resources, using only 'pull' methods.

tailed distributions, i.e. include huge fluctuations that can be problematical for a server[23]. Providers must therefore over-dimension service capacity to cope with seldom events, or accept being choked part of the time. The alternative is to allow asynchronous processing, by a schedule that best serves the individuals in the transaction.

2. The traditional service provider does not have any control over the demand on its processing resources. The client-server architecture drives the servers resources at the behest of the client. This opens us to risk of direct attacks like Denial of Service attacks and load storms from multiple hosts. In other words, ad hoc encounters do not have to trust only individuals but also the entire swarm of clients in a milieu collectively. The alternative is that hosts should be allowed to set their own limits.
3. The traditional service provider receives requests that are 'force-fed' from the network. These might contain attacks such as buffer overflows, or data content attacks like viruses. The alternative is for the server to agree only to check

```

cf:cube: Anomalous (2dev) incoming (non-DNS) UDP traffic on cube/Wed Jun 16 18:28:47 2004
current value 0 av 5.1 pm 14.3
cf:cube: -----
cf:cube: In the last 40 minutes, the peak state was:
{
DNS key: 67.43.48.7 = 67.43.48.7 (24/55)
DNS key: 67.43.48.4 = 67.43.48.4 (24/55)
DNS key: 128.39.89.10 = nexus.iu.hio.no (6/55)
DNS key: 128.39.74.129 = pc129-74.iu.hio.no (1/55)
-
Frequency: 67.43.48.4      |***** (24/55)
Frequency: 67.43.48.7      |***** (24/55)
Frequency: 128.39.89.10    |***** (6/55)
Frequency: 128.39.74.129   |* (1/55)
}
-
Scaled entropy of addresses = 16.9 %
(Entropy = 0 for single source, 100 for flatly distributed source)
-
cf:cube: -----
cf:cube: State of incoming.udp peaked at Wed Jun 16 18:06:08 2004

```

Figure 7: Entropy as a scalar characteristic of the IP distribution

what the client is offering and download it if and when it can be handled safely.

For many tasks of a general administrative nature, time is not of the essence, and a reply within minutes rather than seconds will do. Avoiding messages from a client is a useful precaution: some clients might be infected with worms or have parasitic behaviour.

The dilemma for a server is clearly that it must expose itself to risk in order to provide a service to arbitrary, short-term customers. In a long term relationship, mutual trust is built on the threat of future reprisals, but in an opportunistic or transitory environment, no such bond exists. Clients could expect to ‘get away’ with abusing a service knowing that they will be leaving the virtual country soon. Cfengine provides a mechanism called Voluntary RPC to help prevent this from happening.

In the voluntary cooperation model this risk is significantly reduced in the opening phase of establishing relations: neither “client” nor “server” demand any resources of each other, nor can they force information on each other. All cooperation is entirely

at the option of the collaborator, rather than at the behest of the client. Let us provide some examples of how voluntary collaboration might be used,

After intentions have been established, one needs a way of safely exchanging data between contracted peers in such a way that neither party can subsequently violate the voluntary trust model. To implement this, we demand that a client cannot directly solicit a response from a server. Rather, it must advertise that it has a request and wait to see if a server will accept it of its own free choice.

1. Host A: Advertises a service to be carried out, by placing it in a public place (e.g. virtual bulletin board).
2. Host B: Scout A looks to see if host A has any jobs to do, and accepts job. Host B advertises the result when completed by putting the result in a public place.
3. Host A: looks to see if host B has replied.

This is done entirely using ‘pull’ methods, i.e. each

host collects the data that is meant for it; no host can send information directly to another, thereby placing demands on the other's resources. The algorithm is a form of batch processing by posting to local 'bulletin boards'. There is still risk involved in the interaction, but each host has finer control over its own level of security. Several levels of access control are applied:

- First the remote client must have permission to signal the other of its presence
- Then the server must then approve the request for service and connect to the client.
- The client must then agree to accept the connection from the server (assuming that it is not a gratuitous request) and check whether it has solicited the request.

As an example of this mechanism, applied to distributed system configuration, the Voluntary RPC has been implemented as a cfengine collaborative application. Remote services are referred to one another as "methods" or function calls. An automated negotiation mechanism has not been implemented as this remains an open question. We begin by assuming that a negotiation has taken place that determines which hosts will be clients and which will be servers and what the nature of the service is. This establishes each hosts "Method Peers", i.e. the hosts it is willing to provide services for. On the server host the update.conf rules must also contain a list of hosts to check for pending method requests.

```
MethodPeers = ( client1 client2 )
```

The client *and* server hosts must then have a copy of the invitation:

```
methods:
```

```
client_host||server_host::

MethodExample("Service request",parameter)

action=cf.methodtest
```

```
server=server_host
returnclasses=ready
returnvars=retval
```

```
ifelapsed=120
expireafter=720
```

```
alerts:
```

```
MethodExample_ready::

"Service returned: $(MethodExample.retval)"
```

The predicate classes in the line ending in double colons tells the software that this method stanza will be read both by the client and the server host. On the client it is a request, and on the server it is the invitation. The action and server attributes determine where the methods are defined and who should execute them. Return values and classes are declared serving as access control lists for returned parameters. Since a remote method is not under our local control, we want to have reasonable checks about the information being returned. Methods could easily be asked to return certificates, for instance, to judge their authenticity. The ifelapsed and expireafter attributes apply additional scheduling policy constraints, as mentioned above.

On server host a copy of the invitation is needed to counter-sign the agreement. The module itself must then be coded on the server. The invitation is based on the naming of the hosts; this has caused some practical problems due to the trust model. The use of IP addresses and ties to domain names has proven somewhat unreliable due to the variety of implementations of Domain Name Service resolution software. A name might be resolved into an IPv6 address on one end of a connection, but as an IPv4 address on the other side. This leads to mismatches that are somewhat annoying in the current changeover period, but which are not a weakness of the method in principle.

```
control:
```

```
MethodName          = ( MethodExample )
MethodParameters    = ( value1 value2 )
```

```

MethodAccess      = ( patterns )

# value1 is passed to this program, so lets add to
# it and send the result back for fun

var1 = ( "${value1}...and get it back" )

actionsequence = ( editfiles )

#####

classes:

    moduleresult = ( any )
    ready = ( any )

#####

editfiles:

{ /tmp/file1

AutoCreate
AppendIfNoSuchLine "Important data...${value2}"
}

#####

alerts:

moduleresult::

    ReturnVariables("${var1}")
    ReturnClasses(ready)

```

12 Summary

This summary of cfengine outlines how the general theory of system maintenance is applied and implemented in the software. Further information about the detailed software implementation may be found in the extensive manuals at [24]. The key points to understanding cfengine management are

- The stochastic nature of the environment.
- The discrete nature of host configuration.

- Classification of environmental changes into a patchwork of sets.
- Policy constraints.
- Operator responses to policy deviations (anomalies).
- Convergence and idempotence of configuration operations.

Cfengine is an on-going project. It is the only system known to the author that addresses non-hierarchically organized systems, with partial autonomy and intermittent connectivity.

References

- [1] M. Burgess. Evaluation of cfengine's immunity model of system maintenance. *Proceedings of the 2nd international system administration and networking conference (SANE2000)*, 2000.
- [2] M. Burgess. Computer immunology. *Proceedings of the Twelfth Systems Administration Conference (LISA XII) (USENIX Association: Berkeley, CA)*, page 283, 1998.
- [3] M. Burgess. On the theory of system administration. *Science of Computer Programming*, 49:1, 2003.
- [4] M. Burgess, H. Haugerud, T. Reitan, and S. Straumsnes. Measuring host normality. *ACM Transactions on Computing Systems*, 20:125–160, 2001.
- [5] M. Burgess. Cfengine's immunity model of evolving configuration management. *Science of Computer Programming*, 51:197, 2004.
- [6] P.D'haeseleer, S. Forrest, and P. Helman. An immunological approach to change detection: algorithms, analysis, and implications. *In Proceedings of the 1996 IEEE Symposium on Computer Security and Privacy (1996)*.

- [7] A. Somayaji, S. Hofmeyr, and S. Forrest. Principles of a computer immune system. *New Security Paradigms Workshop, ACM*, September 1997:75–82.
- [8] M. Burgess. System administration as communication over a noisy channel. *Proceedings of the 3rd international system administration and networking conference (SANE2002)*, page 36, 2002.
- [9] C.E. Shannon and W. Weaver. *The mathematical theory of communication*. University of Illinois Press, Urbana, 1949.
- [10] A. Couch and M. Gilfix. 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)*, page 123, 1999.
- [11] M. Burgess and D. Skipitaris. Adaptive locks for frequently scheduled tasks with unpredictable runtimes. *Proceedings of the Eleventh Systems Administration Conference (LISA XI) (USENIX Association: Berkeley, CA)*, page 113, 1997.
- [12] M. Burgess. 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)*, page 169, 2002.
- [13] D.E. Comer and L.L. Peterson. Understanding naming in distributed systems. *Distributed Computing*, 3:51, 1989.
- [14] H. Lewis and C. Papadimitriou. *Elements of the Theory of Computation, Second edition*. Prentice Hall, New York, 1997.
- [15] M. Burgess. Theoretical system administration. *Proceedings of the Fourteenth Systems Administration Conference (LISA XIV) (USENIX Association: Berkeley, CA)*, page 1, 2000.
- [16] M. Burgess. *Analytical Network and System Administration — Managing Human-Computer Systems*. J. Wiley & Sons, Chichester, 2004.
- [17] T.M. Cover and J.A. Thomas. *Elements of Information Theory*. (J.Wiley & Sons., New York), 1991.
- [18] A. Couch and Y. Sun. On observed reproducibility in network configuration management. *Science of Computer Programming*, (to appear), 1994.
- [19] A. Couch and Y. Sun. On the algebraic structure of convergence. *Submitted to DSOM 2003*, 2003.
- [20] S. Traugott. Why order matters: Turing equivalence in automated systems administration. *Proceedings of the Sixteenth Systems Administration Conference (LISA XVI) (USENIX Association: Berkeley, CA)*, page 99, 2002.
- [21] M. Burgess. Probabilistic anomaly detection in distributed computer networks. *Machine Learning Journal*, (submitted).
- [22] G.R. Grimmett and D.R. Stirzaker. *Probability and random processes (3rd edition)*. Oxford scientific publications, Oxford, 2001.
- [23] V. Paxson and S. Floyd. Wide area traffic: the failure of poisson modelling. *IEEE/ACM Transactions on networking*, 3(3):226, 1995.
- [24] M. Burgess. Cfengine www site. <http://www.iu.hio.no/cfengine>, 1993.