Evaluating cfengine's immunity model of site maintenance

Mark Burgess

Oslo College

1 Introduction

The immunity model of system maintenance draws its name from the immune systems of vertebrate organisms. The idea contends that all evolving large-scale cooperative organisms (animals, groups, societies) develop self-protecting sub-systems (immune and repair systems, police and emergency services) which try to keep those organisms in some kind of balance. Without such systems, organisms would quickly perish due to random decay and crime. Computers operating systems are artificial organisms, which work on the same cooperative principles as other organisms, but they have been designed rather than having evolved. They lack basic protecting systems, or immune-repair systems. The immunity model attempts to redress this imbalance by providing an expert counterforce to the forces of random or criminal attack on system integrity. One defines the ideal (desired) state for a computer system and agents (like immunity/repair cells) make sure that no dangerous deviations from that state are allowed to grow.

The immunity model uses a principle of convergence to switch on and off repair and garbage collection mechanisms. When the system is in its ideal state, the immune system becomes inert. When the system deviates from the ideal state, the immune system guarantees to bring it only closer to that state, i.e. the state of the system is never made worse by the agents. This is similar in spirit to danger models of vertebrate immunity[1].

Cfengine is an agent/language framework for implementing the automatic management of large or small system installations with the immunity model. It uses the idea of an expert software agent to perform automatic maintenance on each host in a Unix or NT cluster. Cfengine is suitable for site-wide management; it can be used for large scale cloning of hosts, or for individual specialization, with any degree of granularity. A single configuration file, consisting of a specialized, descriptive language is used to describe the ideal state of groups and classes of hosts on the network.

The purpose of this paper is to take a critical review of the success of the immunity model computer management. What are the criteria for judging a mechanism for maintenance? What aspects of system administration are not covered in this model? Can the immunity model be judged to be superior or inferior to other approaches to system management?

2 Analytical system administration

In spite of USENIX/SAGE efforts, system administration is not yet a continuing research dialog, but more a cycle of reinvention of one-off solutions. If one is to avoid such reinvention and advance the state of the field, more introspection and criticism of the technologies at hand is needed.

The belief that general principles and tools will solve problems at diverse locations has been slow to emerge in the field. System administrators are more inclined to put together some scripts of their own than trust someone else's software. This is possibly because previous software has imposed unreasonable limitations, or implicit policies on its users.

Actual models of system administration have been all but non-existent. Many users have learned to quickly dispose of vendor provided tools and find more homogeneous solutions.

There is a few exceptions to this mode of thinking. The first was perhaps the introduction of the PERL scripting language, which eliminated many of the problems associated with incompatible versions of the Unix shell commands and allowed portable scripts to be written. Cfengine also falls into this category, placing portability and uniformity before detailed functionality. Other systems like Tivoli have taken a brute force approach to the problems of inhomogeneity and widespread applicability, by providing an elaborate management superstructure for traditional scripts. Although these three example tools are quite different, with different target audiences, their jurisdictions overlap and it is natural to consider their suitability for the tasks of system administration.

As a contribution to an emerging dialogue, the present paper attempts a partial appraisal of the author's own contribution to the field: the immunity model and its test-implementation cfengine. Cfengine sets itself apart from other tools by building on specific principles, rather than leaving every detail to the script-writer. It absorbs a significant amount of frequently used code back into the language interface.

In order to evaluate cfengine's performance or value as a system administration paradigm, this paper is divided into two parts: an evaluation of the principles of operation, and an evaluation of the tool itself. It is the principles which are of enduring importance. Cfengine is simply a proof-of-principle implementation, constructed often on borrowed time, and while it incorporates much if the required technology, it has many cosmetic faults.

3 Model and principles

Evaluating models and principles of system administration, in an objectively scientific way, is a difficult problem. Often principles can only be shown to have theoretically desirable qualities, and evidence can be presented in favour of or against the success of strategies based on them. System administration has its own special obstacles: as a strongly social phenomenon, dealing with the behaviour of communities, it resists empirical analysis. Experimental verification of ideas is a practical impossibility: the conditions under which measurements are made are constantly changing, making statistical repeatability a pipe dream. The data one might collect to substantiate a claim are unlikely to be statistically significant or controllable.

An evaluation of the issues specific to the immunity model is additionally awkward since it overlaps with issues common to other potential models based on automation. If one were to pose the question, why is the immunity model better than some other model, it would have to be based on a very incomplete analysis of the model itself, since much of the model is based on the idea of automation, which is common to several approaches. The specific details which are particular to the immunity model[2] are:

- Convergence: every action should bring the system closer to its ideal configuration. Nothing should ever get worse.
- Distributed responsibility: each host has the responsibility for its own health. Human involvement and notification is avoided as far as possible.
- Competition rather than rigidity: provide counter-forces to resource consumption rather than expressing rigid quotas. This allows for fluctuations which require large temporary resources, which are not available in a quota design.

Relevant questions to ask about this model are therefore: To what extent is convergence important? To what extent does the distributed responsibility work in avoiding bottlenecks, such as those which occur when all messages have to filter through a single operator. Other more generic questions are also relevant however: should one rely on automation? Should

not responsibility for changes lie with humans? Can anything general be said about this? From a scientific viewpoint, one would like to ask:

- Does the model increase our understanding of the problems of system administration?
- Does the model have any predictive power?
- Does it generalize or improve upon earlier models?
- Does the result of the model contribute positively to system administration?

The evaluation of the immunity model does not depend upon an evaluation of cfengine (an implementation of the model) as an exercise in software-development per se; that is a separate issue. The concepts which under-pin this model are independent of the mechanisms which implement it. An evaluation of the model could be broken down into hypotheses for definiteness. For example,

1. *The immunity model saves system administrators time.*
2. *Some problems can be avoided completely.*
3. *Results in a quicker solution of problems when they occur.*
4. *The system scales linearly with number of machines.*
5. *The immune system is immune to complete failure.*
6. *There is no problem that an immune system cannot solve. (Obviously false)*

In order to support or oppose these statements differing kinds of evidence is required. For example, the first of these can only be verified by experience. The final one can easily be discredited by a counter example. The claim that the model scales linearly can be proven algebraically in the case where no access to server data is required, and must be measured in cases where file distribution through a potential server bottleneck is involved. Rather than presenting the discussion in terms of hypotheses (of which one could pose many) we shall consider the important issues in turn.

4 Methods of evaluation

Studies which involve multiple and intertwined variables, with rapidly changing boundary conditions (e.g. sociology, psychology and system administration) suffer from a fundamental problem. The scientific method becomes impossible to implement convincingly. No sooner have we measured something, than the conditions under which the measurement were made change. This makes repeated measurements incomparable, and the collection of statistics a nonsense. System administration falls into this category of problems. The result of this is that the body of system administration knowledge is composed largely of high level concepts, which are difficult to formalize and test. There are various ways to lessen the effects of this problem, but there is no way to avoid them completely.

Aiming for scientific progress, one seeks to undertake a series of studies. Sometimes these studies are theoretical, sometimes they are empirical and often they are a mixture of the two. In this case both is required. The aim of a study is to contribute to a larger discussion which will eventually lead to progress in the field. Progress in a field of study often requires the development of a suitable 'technology' for addressing the problems encountered there. That technology might be mathematical, computational or mechanical. It does not relate directly to the study itself, but it makes it possible for the study to take place. Progress is made, with the assistance of this improved technical ability, only if the results of

previous work can be improved upon, or if an increased understanding of the problem can be achieved, leading therefore to greater predictive power. The theoretical approach (here the immunity model) is important for the interpreting of the work, while the technology (here a program named cfengine) is necessary to make it possible.

System administration is beset with the intrinsic difficulty of how to evaluate a model in which computer users are constantly changing the conditions of the observations one would like to record. It would do no good to eliminate the users since they are the source of the problems which one is trying to solve. In principle this kind of noise in data could be eliminated by statistical sampling over very long periods of time, but in the case of real computer systems this is not possible either since seasonal variations in patterns of use, combined with continual evolution lead to qualitatively different behaviours which should not be mixed in a sample. How can this obstacle be overcome?

4.1 Problems with evaluation methods

The simplest and potentially most objective way to test a model of system administration is to combine heuristic experience with short, repeatable simulations. Experienced system administrators have the pulse of their system and can evaluate their performance in a way that only humans can. Their knowledge can be used to define repeatable benchmarks or criteria for different aspects of the problem. But this approach is not without its difficulties. Many of the administrators impressions would be very difficult to gauge numerically. For example: cfengine is designed to relieve administrators of a lot of tedious work leaving him or her to work on more important tasks. Can such a claim be verified? Here are some of the difficulties

Measure the time spent working on the system	The administrator has so much to do that he/she can work full time no matter how much one automates "tedious tasks".
Record the actions taken by the automatic system, which a human administrator would have been required to do by hand and compare.	There is no unique way to solve a problem. Some administrators fix problems by hand, while others will write a script for each new problem. The time/approach taken depends on the person.

If there are no rigid scientific criteria to rely on then how are we to objectively evaluate the performance of a software system for system administration? How can we be sure that we are not fiddling the results? In such a case, the results are inevitably subject to an interpretation. Heuristics and impressions must play a role here.

Case studies are usually no more than anecdotal evidence. They may provide indications which motivate a direction of research, but they cannot be seen as proof of anything since they offer only a single data point. In system administration, at least, sites are sufficiently different to make any collection of case studies scientifically invalid as a statistical sample.

System administration is full of intangibles; this limits models to those aspects of the problem which can be addressed in schematic terms. It is also sufficiently complex that it must be addressed at several different levels in an approximately hierarchical fashion.

4.2 Cooperation and dependency

Another testable issue is that of dependency. The fragile tower of components in any functional system is the fundament of its operation. If one component fails, how resilient is the remainder of the system to this failure? This is a relevant question to pose in the evaluation of a system administration model. How do software systems depend on one

another for their operation? If one system fails, will this have a knock-on effect on for other systems? What are the core systems which form the basis of system operation? In the present work it is relevant to ask how the model continues to work in the event of the failure of DNS, NFS and other network services which provide infra-structure. Is it possible to immobilize an automatic system administration model? In the immunity model, every effort has been made to eliminate the need for dependencies, but such dependencies can always be introduced unwittingly by administrators who make policy.

One possible weakness in the immunity model is that it lacks a global web of communication: if an error on one host requires a change in configuration on another, this information is not easily communicated to the remote host. This type of global optimization at the level of the network community is not available to individuals' internal configuration engines. This is why humans benefit from doctors' intelligence: autonomous immune systems cannot affect the environment which surrounds them, they can only 'complain'. But this is not necessarily a major loss; it also touches on trust issues: if such communication were to be allowed directly between hosts, it would imply an extra trust relationship. Such information might be used to bluff a configuration engine into making changes, or be used as a denial of service attack. Thus from a security perspective, this inconvenience is to be welcomed, since it forces an intelligent intervention, capable of judging the complex interrelationships.

4.3 Evidence of design faults

In the course of developing a model one occasionally discovers faults which are of a fundamental nature, faults which cause one to rethink the whole mode of operation. Sometimes these are fatal flaws, but that need not be the case. Cataloguing design faults is important for future reference to avoid making similar mistakes again. Design faults may be caused by faults in the model itself or merely in its implementation. Legacy issues might also be relevant here: how do outdated features or methods affect software by placing demands on onward compatibility, or by restricting optimal design or performance? To date there is no evidence of this kind of fault in the immunity model itself. Some features of the implementation, cfengine, have proven to be little used however. One example is the suggested sharing model used encouraged for NFS, which uses a universal naming hierarchy to mimic wide area file systems. Only a tiny number of users appear to have adopted this. Other seem to have leapfrogged directly to more advanced file-systems like AFS and DFS for sharing, which provide the same functionality and more.

4.4 Evaluation of system policies

System administration does not exist without human attitudes, behaviours and policies. These three fit together inseparably. Policies are adjusted to fit behavioural patterns; behavioural patterns are local phenomena. The evaluation of a system policy has only limited relevance for the wider community then: normally only relative changes are of interest, i.e. how changes in policy can move one closer to a desirable solution.

Evaluating the effectiveness of a policy in relation to the applicable social boundary conditions presents practical problems which sociologists have wrestled with for decades. Strictly speaking, this is not a part of the immunity model itself, but a layer on top of that. The problems lie in obtaining statistically significant samples of data to support or refute the policy. Only an experienced observer would be able to judge the value of a policy on the basis of incomplete data. Such information is difficult to trust however unless it comes from several independent sources. A better approach might be to test the policy with simulated data spanning the range from best to worst case. The advantage with simulated data is that the results are reproducible from those data and thus one has something concrete to show for the effort.

In addition to measurable quantities, humans have the ability to form value judgements in a way that formal statistical analyses can not. Human judgement is based on compounded experience and associative thinking and while it lacks scientific rigour it can be intuitively correct in a way that is difficult to quantify. The down side of human perception is that prejudice is also a factor which is difficult to eliminate. Also not everyone is in a position to offer useful evidence in every judgement:

- *User satisfaction*: software, system-availability, personal freedom
- *Sys-admin satisfaction*: time-saving, accuracy, simplifying, power, ease of use, utility of tools, security, adaptability.

4.5 Example: Mean time before failure

The probability of host failure is a measurement which is sometimes used to gauge the effectiveness of a reliability model. It can be calculated approximately from measurements of system logs, i.e. from the recorded time between system kernel panics and power failures. Whether or not manual reboots should be taken into account here is debatable; sometimes free-standing hosts are rebooted or switched off by ignorant users, or even succumb to power failure. Power failure is usually a detectable state transition however and can be excluded from these cases. Here we shall consider all reboots to be indications of failure, even though this is probably not true; this gives a worst case scenario. Users often reboot hosts in response to what they perceive as a problem with the system so this is consistent with the aim of the measurement. Unless systems fail often, this is not a statistically significant measure however; the best one can do is to measure 'uptime'.

The following table shows tentative data for the up-times of hosts belonging to different groups of hosts at Oslo College and the University over a period of four years, corrected for controlled power outages.

Av. uptime immune hosts group 1	108 days
Av. uptime immune hosts group 2	29 days
Av. uptime immune hosts group 3	40 days
Av. uptime non-immune hosts group 1	(no data)
Av. uptime non-immune hosts group 2	14
Av. uptime non-immune hosts group 2	30 days

Group 1 comprises servers to which ordinary users have no physical access but do have network access. These hosts generally run for as long as physical conditions permit, i.e. between power failures or disk crashes. Separate location. Group 2 is GNU/Linux hosts to which users have physical access, heavily used. Common location. Group 3 is Sun/Solaris hosts to which users have physical access. Common location.

These data must be considered only as heuristic values since they have not been collected under identical conditions. They apply for a period of time extending over three years, not necessarily overlapping. During that time the precise details of system policy have changed many times and the boundary conditions for the work have changed, both with respect to the number of hosts and users, the stability of the operating system software and the intensity of use. The data are not specific enough to be able to attach cause to the values, but it seems qualitatively clear that hosts to which users have physical access tend to be rebooted by users. Hosts which are physically secure seem to continue to run basically ad infinitum, barring power failure or hardware failure. There is no reason to suppose that one group of hosts has been used more than another group.

These data are suggestive but not conclusive. There are too many differences to consider this a controlled experiment. This is the main problem in collecting data to verify claims in system administration. What is important perhaps is not the comparison between immunity and non-immunity (since these occur in different sites and times) but the fact

that the immune hosts' lifetimes can be extended to several times any patterns of behaviour (daily or weekly) that users or software exert on them. In the case of physically secure machines, there does not seem to be a practical limit to the lifetime of a host.

Mainly indirect, anecdotal evidence supports the idea that automatic immune configuration improves a host's resistance to failure. For example, in a recent episode at Oslo College a GNU/Linux host which did not run cfengine was broken into, in spite of the fact that it was running a version of the operating system which was up to 18 months newer than other comparable systems on the network. The reason for the break-in was simply naive configuration and inexperience on the part of the owner. Comparable episodes have been reported by other users.

At this level that one obtains suggestive evidence for the benefits of automation. However, these episodes do nothing to confirm or deny the immunity model, or cfengine specifically.

4.6 Fluctuations and their convergence

The immunity model allows fluctuations in system parameters, i.e. the values may change with time, but generally have some more or less constant average value over long periods. This is in contrast to some models, such as those involving quotas where user-processes must subsist within rigid boundaries. For example, huge temporary files are allowed by an immunity model as long as they do not collectively build up and choke the system in the long term. How far should these fluctuations be allowed to deviate from the ideal? Can they accumulate leading to a systematic error or drift or are they random averaging out to nothing? This is a question of policy.

Quantitative measures of fluctuations can be given a rigorous meaning with regard to a theoretical model. However the practical problem is easily demonstrated by comparing two as near identical machines as is practically possible. Halting cfengine for one of these machines for 36 hours, while allowing cfengine to run normally on the other for the same period, permits a comparison of the state of disarray of the non-immune host after this period of time. The output generated reflects the actions which were actually performed, not the number of things which was checked. In both cases the same number of items was checked on the basis of the same system policy. The host running cfengine produced the report shown in figure 4.6.

```
metaverse# cfengine -q -I -K
cf:metaverse: Executing script /usr/sbin/rdate nexus >/dev/null 2>&1...(timeout=0)
cf:metaverse: Finished script /usr/sbin/rdate nexus >/dev/null 2>&1
cf:metaverse: Signalling process 98 (inetd) with SIGHUP
cf:metaverse: Executing shell command: /local/sbin/sshd
cf:metaverse: (Done with /local/sbin/sshd)
cf:metaverse: Signalling process 98 (inetd) with SIGHUP
```

Figure 1: The tasks carried out by cfengine on a a host running cfengine regularly. This is to be compared with the next figure.

while the host which had had cfengine disabled produced the output in figure 4.6. This later output shows how several files have fallen behind in their updates and how both file and process garbage has accumulated over the 36 hour period. This experiment can be repeated and the number of lines of output can be used as a measure of the degree of deviation from ideal state. Over fifty runs, the hosts which did not run cfengine regularly required significantly more work that those which had received regular maintenance, where 'significantly' means several times the number of lines. The actual measured average has no reliable meaning, since it depends both on policy and the nature of the work being reported, thus it is not quoted here. It is sufficient to record that the average value was

```
romulus# cfengine -q -I -K
cf:romulus: Updating image /etc/group from master /iu/nexus/local/iu/etc/group.linux on nexus
cf:romulus: /etc/group had permission 600, changed it to 644
cf:romulus: Updating image /etc/cfengine/inputs/cf.site from master /iu/nexus/local/gnu/lib/cfengine/inputs/cf.site on localhost
cf:romulus: /etc/cfengine/inputs/cf.site had permission 600, changed it to 755
cf:romulus: Updating image /etc/cfengine/inputs/cf.main from master /iu/nexus/local/gnu/lib/cfengine/inputs/cf.main on localhost
cf:romulus: /etc/cfengine/inputs/cf.main had permission 600, changed it to 755
cf:romulus: Updating image /etc/cfengine/inputs/cf.main~ from master /iu/nexus/local/gnu/lib/cfengine/inputs/cf.main~ on localhost
cf:romulus: /etc/cfengine/inputs/cf.main~ had permission 600, changed it to 755
cf:romulus: Updating image /etc/cfengine/inputs/cf.site~ from master /iu/nexus/local/gnu/lib/cfengine/inputs/cf.site~ on localhost
cf:romulus: /etc/cfengine/inputs/cf.site~ had permission 600, changed it to 755
cf:romulus: Updating image /etc/hosts.deny from master /local/iu/etc/hosts.deny on nexus
cf:romulus: /etc/hosts.deny had permission 600, changed it to 644
cf:romulus: Updating image /etc/hosts.allow from master /local/iu/etc/hosts.allow on nexus
cf:romulus: /etc/hosts.allow had permission 600, changed it to 644
cf:romulus: Executing script /usr/sbin/rdate nexus >/dev/null 2>&1...(timeout=0)
cf:romulus: Finished script /usr/sbin/rdate nexus >/dev/null 2>&1
cf:romulus: Deleting /tmp/kfm-cache-654/index.txt
cf:romulus: Deleting /tmp/kfm-cache-768/index.html
cf:romulus: Deleting /tmp/kfm-cache-768/index.txt
cf:romulus: Deleting /tmp/kfm-cache-532/index.html
cf:romulus: Deleting /tmp/kfm-cache-532/index.txt
cf:romulus: Deleting /var/spool/cfengine/_etc_hosts.deny.cfsaved
cf:romulus: Deleting /var/spool/cfengine/_etc_group.cfsaved
cf:romulus: Deleting /var/spool/cfengine/_etc_cfengine_inputs_cf.site.cfsaved
cf:romulus: Deleting /var/spool/cfengine/_etc_cfengine_inputs_cf.main.cfsaved
cf:romulus: Deleting /var/spool/cfengine/_etc_cfengine_inputs_cf.main~.cfsaved
cf:romulus: Deleting /var/spool/cfengine/_etc_cfengine_inputs_cf.site~.cfsaved
cf:romulus: Deleting /var/spool/cfengine/_etc_hosts.allow.cfsaved
cf:romulus: Signalling process 7575 (maudio) with SIGKILL
cf:romulus: Killed: henrikf   7575  0.0  1.3  4328   408  ?  S    Jan 15   0:00 maudio -media 258
cf:romulus: Signalling process 5427 (maudio) with SIGKILL
cf:romulus: Killed: sigvarr   5427  0.0  1.4  4328   428  ?  S    Jan 14   0:00 maudio -media 256
cf:romulus: Signalling process 5572 (maudio) with SIGKILL
cf:romulus: Killed: ulekler   5572  0.0  1.3  4328   404  ?  S    Jan 14   0:00 maudio -media 257
cf:romulus: Signalling process 7566 (kaudio) with SIGKILL
cf:romulus: Killed: henrikf   7566  0.0  1.5  4660   464  ?  S    Jan 15   0:00 kaudioserver
cf:romulus: Signalling process 5419 (kaudio) with SIGKILL
cf:romulus: Killed: sigvarr   5419  0.0  1.4  4660   444  ?  S    Jan 14   0:00 kaudioserver
cf:romulus: Signalling process 5564 (kaudio) with SIGKILL
cf:romulus: Killed: ulekler   5564  0.0  1.4  4660   444  ?  S    Jan 14   0:00 kaudioserver
cf:romulus: Signalling process 106 (inetd) with SIGHUP
```

Figure 2: The tasks carried out by cfengine on a a host which had been deprived of cfengine care for 36 hours. Compared to the previous figure, there is a lot of work to be done to bring the system back in line with policy.

a statistically significant number. The conclusion is therefore that hosts running cfengine regularly are significantly closer to their ideal state than hosts which did not.

The data collected in connection with feedback studies in ref. [3] show that there is a steady drift (increase) in disk usage but that behaviour is dominated by large fluctuations. This behaviour is typical of multiuser hosts. Single user hosts might be be better behaved depending on the habits of a user but worst case behaviour should always be expected.

The data indicate a pattern of usage which does not lend itself to rigid quotas. The indication is then that it is important to allow some elasticity. For example one many compare the idea of disk-quotas with enforced garbage collection. Many programs make use of temporary files (compilers, web browsers etc) which are often large. These files do not need to exist for long periods of time, but rigid disk quotas might prevent them from being created at all and thus some software will fail with rigid quotas. With a fluctuation model temporary files can be generated freely, but the immune system will clean them away later as a normal part of its operation.

The need for fluctuations is clear: if one takes the idea of a rigid system to extremes then, in a quota environment, users would not be able to change any feature of the system: no files would be created, no processes would be started. This would be a farcical situation, but the ideal is clearly a matter of degree. How large can one allow fluctuations in resource usage to become before system integrity is compromised? Another way of putting this is: how large can fluctuations become before an immunity engine would be unable to rescue the system?

4.7 Feedback regulation

In any feedback system there is the possibility of instability: of either wild oscillation or exponential growth. Stability can only be achieved if the state of the system is checked

often enough to adequately detect the resolution of the changes taking place. If the checking rate is too slow, or the response to a given problem is not strong enough to contain it, then control is lost. The immunity model admits and even encourages the use of feedback as a paradigm to control resource usage through competition.

In ref. [3] it was shown how uncontrollably spawning processes could be successfully controlled with a surprisingly simple cfengine program. This was initially unexpected. Even when the process table was full of user processes to the point where further user-forks failed, a privileged process run by root was able to stop the problem given the fixed size of the process table and eliminate every offending process in just three iterations. The same experiment was applied to disk usage using feedback methods with similar findings. The reason why it is possible to win over runaway resource usage is simple: the resources are *finite*. Experiments indicate that, as long as crucial functions are not prevented completely, a privileged immune model will always be able to regain control of fluctuations in a finite system.

This poses a question: is feedback really necessary at all? Feedback is an implicit form of recursion, but many recursive algorithms can be replaced by iterative ones. The advantage of recursion is that one does not need to specify the size of the data one is parsing, whereas iteration lends itself to finite size structures. In today's operating system hosts, the size of system resources often seems unlimited but this is only an illusion. Process tables have finite sizes and disk partitions have only finite numbers of blocks.

As an example, consider the case in which a process is forking uncontrollably. For example, consider the Unix shell script:

```
#!/bin/sh

echo README
README
```

This script is called README because a problem which occasionally occurs on Unix systems is that README files are carelessly given executable rights by a user who then inadvertently executes a README file as an executable script. Since README files often contain text which a Bourne shell interpreter can understand and often include the word 'README' itself, this accident often leads to an uncontrollably spawning process. The execution of the above script leads to a rapidly spawning process which quickly fills the process table. What is interesting to determine is the following: can a simple iterative cfengine program handle this situation and respond quickly enough to be able to kill all instances of the offending program? The answer turns out to be yes, because of the finite size of the process table. Consider the program:

```
control:

   actionsequence = ( processes )

processes:

  "README" signal=kill
```

In the first few trials this program was used on a GNU/Linux host in order to counter the README script. After starting the README script, the cfengine program was run at different time intervals after the start of README. If the time difference between starting README and the cfengine program was not too great, this single iteration was able to

kill all of the README processes. However in a real situation one cannot predict the time between the initiation of an uncontrollable process and the response from an immune reaction. With this simple program, after a certain threshold the number of copies became too great to catch all of the processes in a single iteration. Experiments showed however that repeated calls to the program could eventually be used to stop all the processes. After a time processes would not be able to spawn new children as the process table has a finite size. In GNU/Linux some processes are reserved for privileged users, meaning that cfengine will always be runnable.

This begs the question, could one simply have added the iteration to the program at the beginning. For example:

```
control:

   actionsequence = ( processes.1 processes.2 processes.3 )

processes:

  "README" signal=kill
```

After a number of trials it was possible to show that three iterations was enough to stop any runaway process with this experimental setup. Clearly the reason for this is the finite size of the process table. This result is somewhat surprising. One might have been tempted to look for a complicated solution to this problem involving feedback, but this proves completely unnecessary.

The finiteness of resources might therefore be an important clue to building immune systems. Virtual file-systems which present the appearance of unlimited resources might actually be an undesirable illusion for an immune system. They would rapidly lead to the inability of the immune system to parse the disks. This is something to be borne in mind by system designers.

4.8 Time scales

A straightforward comparison of the time-scales involved in automated maintenance, to those of manual human maintenance can be made for any operation which is programmable in an automatic system with current technology. It is easy to show that human administrators only compete with automatic systems in speed and efficiency at times of the day when they have nothing pressing to do. Indeed, it is always possible to arrange for an automatic system to beat a human, provided it can run in overlapping instantiations (see figure 1).

Alarm systems which merely notify humans of errors and then rely on a human response are intrinsically slower than automatic systems which repair errors, provided the alarms represent errors which can be corrected with current automation.

The response time $t_{\rm auto}$ of a automatic machine system M, (e.g. cfengine) falls between two bounds

$$n\,T_p + T_e(A) \geq t_{\rm auto} \geq T_e(A) \tag{1}$$

where T_p is the scheduling period for regular execution of the system (e.g. the cron interval, typically half-hour to an hour), $T_e(A)$ is the execution time of the automatic system (typically seconds). The integer $n \geq 0$ since the number of iterations of the automatic system required to fix a problem might be greater than one. The time required to make a decision about the correct course of action $T_d(A)$ is negligible for the automatic system.

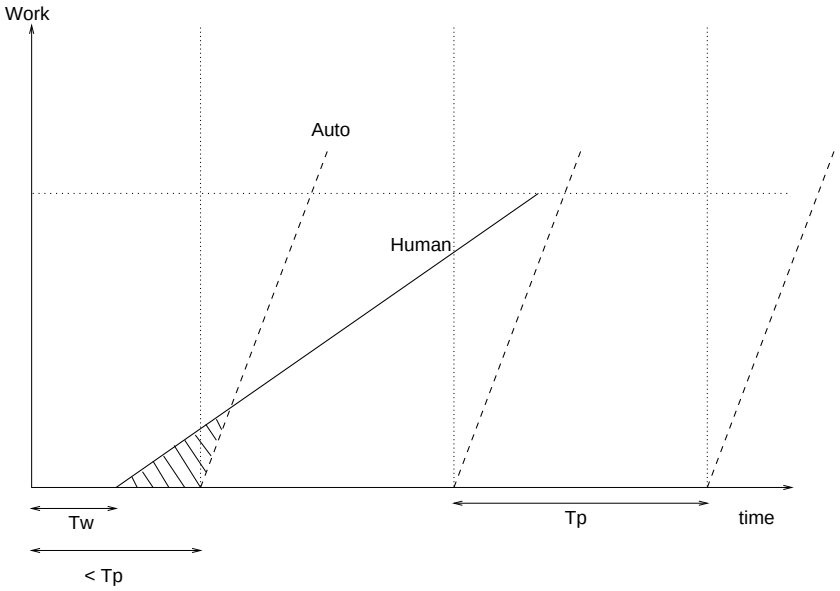

Figure 3: Overlapping work rates of human and automatic systems.

For a human being, making a decision based on a predecided policy, the response time $t_{\rm human}$ falls between the limits:

$$\infty \geq t_{\rm human} \geq T_w(H) + T_d(H) + T_e(H). \tag{2}$$

$T_d(H)$ is again the decision time, or time required to determine the correct policy response (typically seconds to minutes). $T_e(H)$ is the time required for a human to execute the required remedy (typically seconds to minutes). $T_w(H)$ is the time for which the human is busy or unavailable to respond to the request, i.e. the wait-time. The availability of human beings is limited by social and physiological considerations. In a simple way one can expect this to follow a simple pattern in which the response time is greatest during the night,

$$T_w(H) \sim C_1 + C_2 \, \sin(\Omega t),$$

with $C_1 > \frac{1}{2} C_2$ whereas

$$T_w(A) \simeq 0.$$

where $C_1 > C_2$ are constants and Ω is a daily frequency. We can note that human response times are usually much longer than the machine response times,

$$\begin{aligned} T_d(A) &\ll T_d(H) \\ T_e(A) &\ll T_e(H) \end{aligned} \tag{3}$$

Also,

$$\begin{aligned} T_e &\gg T_r \\ T_d &\gg T_r \end{aligned} \tag{4}$$

and that the periodic interval of execution of the automatic system is generally required to be greater than the execution time of the automatic system, thus avoiding overlapping executions (though this is not a problem, see the discussion of adaptive locks[4])

$$T_p \geq T_e. \tag{5}$$

It is always possible to choose the scheduling interval to be arbitrarily close to $T_e(A)$ (i.e. small) and then provided,

$$T_w(H) > T_e(A) \tag{6}$$

it must be true that the automatic system can always win over a human. This last inequality requires qualification however, since very long jobs (such as backups or file tree parses) increase exponentially in time with the size of the file tree concerned. This makes a prediction: it tells us that one should always arrange to allow such long jobs to be run last in a sequence of maintenance tasks, and also in overlappable threads. This means that long jobs will not hinder the rapid execution of a maintenance program.

Cfengine allows overlapping runs using its scheme of adaptive locks[4]. Thus, by scheduling long jobs last in a cfengine program, it is always possible for cfengine to beat a human, unless it is prevented from running.

4.9 Vulnerability of the model to subsystem failure

A concern for a model which is supposed to ensure security and longevity is that the integrity of the software might be compromised through an implicit dependency. Any software system which has control over the total operating system of a computer is vulnerable to abuse and attack by spoofing. How invulnerable can an immune system be? Vulnerabilities can arise through bugs as well as through dependencies and trust relationships. A key feature of the immunity model is that it trusts as few things as possible. Like any other internet software however, it has to implicitly trust the network services which provide the core communication protocols. Dependency on hostname lookup is a clear example. This dependency is obvious and provable whenever a host is referred to by name rather than by number; it is a function of the internet protocols, not of any dependent software.

If DNS (the Domain Name Service) fails or is spoofed, hostname lookups fail and the translation services are compromised. A trusted server might be substituted for a falsified server and be used to install false data on the system. The immunity model can continue to function in this case, but some functionality might be lost or used to trick the system into importing false data. This kind of spoofing attack affects all host-based software systems which import data from a network. Whether or not such loss of functionality is capable of causing complete system failure depends on the administration policy in use; however, any policy which did have this property would have to be considered as violating the tenets of the immunity model.

In general it would be safer to use machine level addresses in numerical form for optimum security. This method minimized the number of software components involved in authentication and is secure to DNS level spoofing attacks, but not to TCP spoofed connections. TCP spoofing can be prevented with proper router configuration so this is not a relevant issue for host-based immunity models. It is a security issue which should become increasingly difficult in years to come. If a server is spoofed, the failure of remote copying services (like cfengine's own copy service, or the NFS) can compromise individual operations, but none of these will leave the host in an inconsistent state. i.e. it is possible to substitute a false configuration but it is not possible to cause partial writes or contradictory file references. Thus, if spoofing can be eliminated by careful attention to security policy, the only danger comes from denial of service attacks. The only danger from protracted denial of service attacks is that a host configuration might become antiquated. Currently cfengine uses adaptive locking to provide partial protection from denial of service attacks, but there exist no foolproof methods for protection from such attacks.

4.10 Human involvement

The indications of the foregoing sections are that an automatic system combining iteration and feedback will be able to dampen out any fluctuations in a finite system. This is a

necessary feature of an immune model. This does not mean to say that a control based model would not be able to solve the fluctuation problem also. The issue one always comes back to is that of a human bottleneck. What makes an immune model different? The answer is that the diagnosis and response does not have to pass through a human go-between.

First of all there is problem detection. Without an automatic monitor one has to rely on divergent fluctuations being reported by a user. This could take several minutes, perhaps fifteen minutes before a user is able to contact a human administrator. Next there is the issue of whether an administrator would be able to secure access to a host which is consumed by a runaway process or a full disk. Often divergent behaviour makes login impossible for new users. Assuming that the administrator can log in, and overcome a feeling of panic in fighting a losing battle, it would be necessary to locate and kill the offending processes. Assuming that a suitable program had been written to perform this function at all (such programs are not commonly available) then the turn around time for a response is approximately a half hour after the occurrence of the problem. This is an optimistic assessment. With an immune model checking hosts at half hour intervals the same result could have been achieved *pessimistically* within half an hour and without involving a single human being. These numbers are used here based on the author's own experience at two different university environments.

An immunity model does not in itself contribute anything to the issue of assisting the cooperative effort between human administrators in a large network. However, when immunity is based on a single centralized policy, all human administrators are forced to see one another's decisions and changes, and thus there is a decision bottleneck, without forcing a performance bottleneck. The cfengine user survey indicates that most users believe this to be an efficient form of disciplined communication.

4.11 Prediction and understanding

The success or failure of a model lies not only in its ability to solve specific immediate problems but also in its ability to make predictions *before the fact* and to lead to a greater understanding *after the fact*. The fact that immune systems protect all animal life larger than tadpoles makes it intuitively clear that something about an immune system must provide a selective advantage over mechanisms without an immune system. An grain of further reflection makes it clear that immune models are no more than feedback prey-predator models which invoke the concept of selective competition for finite resources. Operating systems are about resource allocation between competing factions so it should not be a surprise that the introduction of another competitive force (the immune system) would be needed in order to regulate the availability of free resources in a competitive environment. There are dozens of examples of this abstract idea to be found in real cooperatives. The predictions of such models are fairly clear:

- *Finiteness* or the competition for finite resources within a finite system. It is always possible for a strong enough competitor to either win or to control an adversary.
- *Threshold behaviour*. Given competitors with fixed strength, there exists a threshold beyond which the system will run away in one direction, i.e. one of the competitors can win the competition if a threshold is crossed. In immune responses this has to do with the timing with which the immune response is activated. If the response happens too late, it might not be possible to gain the upper hand in the competition.

These behavioural elements of competition have been observed in ref. [3]. So far threshold behaviour has not been observed. This remains a theoretical possibility to be demonstrated (or preferably not) in the future.

Does the immunity model then lead to a greater understanding of the business of system administration? Within the boundaries of the definition adopted here there are several things one can learn from the success of the model, reported by its users (see below). The

first (almost a tautology) is the extent to which resource control plays a central role in the stability of both single operating systems and networked systems. The truth of this claim can be seen in the on-going experiments of the sites running cfengine. The numbers of megabytes of files deleted per day in order to maintain a status-quo is almost sufficient to prove the point. The same applies to other resources. Similarly the extent to which accidents and errors lead to incorrect configuration of file or process attributes. However this is, by now, an almost trivial point.

Another more important way in which the model contributes to understanding of system administration problems is that the method of convergent automation forces sites through a disciplinary procedure in which they must formalize their problems into a well-defined, implementable policy. While this inevitably leads to a simplification of many problems, it can usually be an acceptable simplification whose benefits are worth the lack of nuance.

It is also possible to turn the above around and use the model as a particular looking-glass for computer systems. How can the observed behaviour be explained in terms of the model? In the case of disk and other resource usage the prey-predator viewpoint is an obvious one, once pointed out, and provides an immediate insight into the possibilities for controlling fluctuations without too heavy a hand. Also in resource attributes, such as file permissions which can lead to security breaches, fluctuations due to human error can be seen as random perturbations which require dampening.

5 Evaluation of cfengine

We now turn to an evaluation of the immunity model's implementation tool *cfengine*. Cfengine was designed with an immunity model in mind, but it provides many semi-intelligent features which are not present in other system administration tools (e.g. tree linking with local override, process management, class based decisions etc.) and thus it also plays a role in system administration independently of the concepts surrounding it. Indeed, most users perceive cfengine as a system administration tool, without appreciating the model and principles on which it is based.

In general terms of policy implementation, cfengine is versatile: it allows one to emulate expensive technology (like disk mirroring) cheaply: i.e. at no physical cost and with the minimum of effort (just a single line in a configuration program). This has contributed to its popularity as a tool, independently of any connection to an immunity model.

Since each feature of cfengine is based directly on design principles, including convergences and independence, there is no question of whether or not it carries out the low level operations it was intended for apart from the possible occurrence of bugs. Since bugs are associated with the software development process which is not being studied here, they will be ignored in this paper. Cfengine was not designed in any stringent software development environment by a team of engineers, it was written, to the best of the author's ability, in order to realize certain theoretical requirements for automation of system administration experimentally. It is therefore not appropriate to consider it to be a software 'product' in the formal sense.

Whether or not cfengine performs an adequate job in implementing an administration policy does require verification however. Cfengine was designed to have the features of the immunity model: namely convergence to a specified state and independence from outside intervention. The evaluation of cfengine thus reduces to two things: the completeness of its repertoire of functions and what its users think of it, based on their own experiences. Neither of these gauges can be made fully objective in the scientific sense, but the first can be made plausible.

5.1 Repertoire

There is a finite number of things which can be manipulated in an operating system. Cfengine implements tools for examining files, creating files, aliasing files, replacing files, renaming files, removing files, editing files, changing access rights on files, starting and stopping processes, signalling processes, examining and configuring hardware devices. Moreover it is extensible by modular scripting. Cfengine does not know any commercial binary database formats so it does not have a native mechanism for reading from or writing to non-ASCII files, but this is not normally a limitation since most binary file formats used in operating system configuration (e.g. the WIN32 registry) can be converted to text format first by a translation wrapper. Support for translation wrappers is included in cfengine. Other operations which are in the form of transactions require specialized scripts or modules, e.g. in Perl. This extensibility through integrated, user-defined scripts makes cfengine not just a tool but a supportive environment. It is not cfengine's prerogative to compete with more appropriate languages for specialized tasks; its aim is to provide a framework for integrating all such tasks.

One area in which cfengine has made a standardizing contribution to file security models is in providing an operating system independent model for configuring access control lists. ACLs are common to many operating systems today but there is no standard implementation for these and the tools for manipulating them are clumsy and interactive. By providing an off-line method for ACL configuration, cfengine has made the use of ACL control viable for Unix-like systems where their use has been optional. In systems like NT where ACLs are the only security mechanisms, this will be essential.

Is cfengine's approach to extensibility better than other approaches? Other tools like OpenView and Tivoli can also be used to install user defined scripts, but these are not operating system independent, class sensitive or of net-wide validity; moreover their operation cannot usually be used to trigger responses from other parts of the system administration repertoire, only warnings to contact a human. Tivoli is exceptional in allowing system thresholds to trigger the execution of tasks or package copies, but not every contingency can be interpreted as a threshold transgression. A error is not a threshold problem. An error needs to be fixed once and then left alone. This requires a notion of convergence which could be incorporated into a Tivoli script run at regular intervals, but it is not as part of the Tivoli model. Tivoli itself will not switch off the script if the problem is fixed. It would have to be added by a user. One could, for instance use Tivoli simply as a front end to cfengine (though the irony might be dizzying). No doubt Tivoli will add such features in time. At present cfengine apparently offers a marginally more versatile environment for extensibility in the tasks it was designed to perform.

5.2 Hypotheses

Other ways of gauging cfengine's success or failure can be addressed by formulating hypotheses:

1. *Cfengine places no significant load on a computer.*
2. *Cfengine has no more limitations than any other system. (Untestable)*
3. *Cfengine's atomic operations are always convergent.*
4. *Cfengine's atomic operations are fault tolerant.*
5. *Cfengine covers every manipulative operation required for a general operating system.*
6. *An expert system approach to system administration is better than one involving only a human knowledge base.*

Not all of these hypotheses are testable. Some (e.g. number 3) are manifestly false, but can be made true with certain qualifications, so, again, rather than considering specific hypotheses, more clarity will result from a higher level discussion of the issues.

5.3 Expert system approach

The expert system base of cfengine is the device which embodies experience acquired during the evolution of an immunity model. This is where one collects experience and rules about states and actions to be taken. Since the release and publication of cfengine (1994/95), two other systems have adopted some of its features. Tivoli newly announced (Dec. 1998) that it would include an expert system in its management software and Host Factory has been using a database approach for a few years which has related related ideas, though it is more rigid than cfengine. Reference [5], which appeared about the same time as the publication of cfengine also advocates the use of such an approach. This can be taken as support for this approach to system management. The expert system approach also plays an important role in centralizing the information behind a system policy. It directly assists the business of cooperation between human administrators by providing a single point of reference and a self-documenting base of responses to crises.

5.4 Convergence and response time

One way of evaluating the performance of cfengine is to find the average slew rate for fluctuations of the system from its equilibrium position. In other words, we look for the average time for the system to return to normal after an 'incident'. This is not a clean measurement as it would be in the case of a pendulum swinging about its equilibrium position. After all it is not cfengine which is responsible for the response but the policy expressed in order to implement such a response which is being tested here. Cfengine is simply the reactor which detects the undesirable state and activates a pre-programmed response in order to fix it. What we wish to test is therefore how a realistic example responds to different kinds of pressure from user perturbations. We shall consider two separate tests: one concerning processes and the other concerning disk space.

The use of feedback methods to control disk-space has already been considered in ref. [3] with real data from a real system. It is also possible to push the system artificially by simulating particularly challenging incidents which would not normally arise. This has been done in several instances with

- Disk filling
- Process filling
- Adaptive lock contention

In each case the approaches could be made successful, within the boundaries of the policy. Each test relied upon the success of a given static policy for control. The exact time taken to dampen out fluctuations depended on the nature of the policy and the frequency of the immune checks. What was missing from these trials was the ability to adjust policy automatically to copy with the unexpected. The is presently beyond the capabilities of available technology.

What about the response time of an immune system compared to the response time of a human? One cannot measure the full effect of a system administrator being completely absent, e.g. how do users experience the total system when the admin is on holiday? However, one can ask what the effect on the stability of the total system was. Here one may conclude that, short of hardware failure or extreme unforseeable circumstances, the absence of the system administrator has little effect on the system.

5.5 Effect on load averages

Cfengine does not place large demands on system resources, with the possible exception of during large file-tree distribution. One would not therefore expect it to have a significant impact on load averages, except during major software distributions. What can happen however is that the extensive use of cfengine to perform large scale checking could lead to contention for disk scheduling. The adaptive locking features were intended to ensure that cfengine would not contend with itself for resource usage, but they do not help against resource races with other processes.

Experience over several years shows that the load presented by cfengine is minimal, seldom more than a few percent of available CPU time even during intensive tasks. This presupposes that user scripts which are CPU bound are not defined to be a part of cfengine. Resource contention, of course, has nothing to do with cfengine itself but with the method of file tree parsing which is common to a lot of software. On operating systems without proper disk scheduling algorithms this might be a significant mechanical problem, however for the operating systems which are actual for cfengine this is not an issue. In a test using a Sun Enterprise II server with medium load, a particularly contentious disk-intensive cfengine process was started. This resulted in 11 percent of the CPU usage and reduced but barely noticeable performance degradation for other tasks. While the values are not particularly significant what is important is that the cfengine process (even when working hard) was well within the limits of reason. Interactive users of the system were not noticeably affected.

To examine the load placed on the most active server hosts at Oslo College measurements of disk activity were made from the kernel data, using `vmstat` and `netstat`.

CPU efficiency has never been a criterion for cfengine. Safety/security and adaptability have come first, then disk access efficiency, memory efficiency and last of all CPU usage. Nonetheless, cfengine does not transgress the bounds of decency by placing noticeable load on hosts running it.

5.6 Limitations

The assertion that cfengine has no limitations which other administrative tools have is not a readily testable hypothesis; however the converse is readily apparent. Most other system administration tools are lacking in the ability to classify network resources according to an abstract model, they lack the notion of convergence towards a fixed state and most of the them lack any kind of feedback regulation.

Cfengine programs attach actions to methods which detect patterns. From experience, the main limitation in cfengine is in the techniques available for pattern detection. Patterns detect files or processes which have particular properties. At present pattern detection is based on regular expression string matches, date stamps and other file attributes.

Cfengine is missing some obvious, but infrequently required pattern matching criteria, for instance, the ability to search for files owned by a specific user when checking permissions. These deficiencies do not affect the principle of its operation, only its effectiveness. They can be added quite easily; the only restriction is in the programming time required to implement them. The main reason why cfengine works well even without these additional features is that experience shows that simple rules are often more efficient and more predictable than complex rules.

Biology also indicates that this is true. Biological immune systems have non-specific rules for garbage collection (macrophages and natural killer cells) and highly specific rules for signal detection (B and T cells), followed by a mechanism which marks certain identified patterns as garbage.

The addition of new pattern matching features will no doubt continue for a long time. Indeed this is the nature of the extensions discussed in the ref. on feedback. In the future

```
Suspicious file ^H^H^H in /u1/olseny/.netscape has no alphanumeric content
Suspicious file ^H in /u1/theigms/sdt/demon has no alphanumeric content
Suspicious file ! in /u2/bohmerm/help has no alphanumeric content
Suspicious file # in /u2/bohmerm/help has no alphanumeric content
Suspicious file : in /u2/bohmerm/help has no alphanumeric content
Suspicious file - in /u2/nesst/x has no alphanumeric content
Suspicious file -^H^H in /u2/rustanr has no alphanumeric content
Suspicious file $ in /u2/arshadm has no alphanumeric content
Suspicious file ^H in /u2/bersetg has no alphanumeric content
Suspicious file $ in /u2/waldenk has no alphanumeric content
Suspicious file  in /u2/mirzam has no alphanumeric content
Suspicious file ^H in /u2/mirzam has no alphanumeric content
Suspicious file ^B in /u3/henrikf/OS/FS_copied has no alphanumeric content
```

Figure 4: *Output policies for cfengine tend to start out too verbose in the beginning as a new feature is added. These are later made silent as more is learned about the behaviour of the feature. For example, here is early test-output from a security feature designed to reveal hidden files after adding a test for non-alphanumeric filenames. The result is interesting, but not useful. Clearly most of the files are accidents. For instance the control character indicates that a user has typed backspace to delete a filename, but that somehow the file has been made by accident. Other files on the other hand filename composed of punctuation marks can be legitimate. After receiving this output, the test was altered to check for only non-printable characters.*

feedback techniques will improve to encompass introspective analysis of the network and the operating system.

5.7 Human involvement

Another area where improvements could be made is in the amount of residual output generated by cfengine and how this output is collected and delivered to a human. The number of error messages being reported is sometimes too high. Some of the error messages are quite uninteresting. For example, 'cannot mount a file-system' etc, which one has little control over. Timing problems in the RPC substructure cause these errors and they resolve themselves.

In the original cfengine, the idea was to report all deviations from intended configuration. However, this resulted in hundreds of messages per day even from only thirty or forty machines, so as the system became trustworthy these messages were silenced one by one and replaced with a 'verbose' option for debugging purposes. The default was therefore to produce no output. In fact, a new system of graded output was developed in response to this issue.

Cfengine's current output model works on a 'want to know' basis. Cfengine reports nothing unless i) its configuration stipulates that it should, or ii) the problem is of a serious nature which cannot be resolved by rerunning cfengine. In practice most of the latter category concerns errors from system calls which resolve themselves after a certain time and so delivering these messages to a human administrator simply generates noise. This is a topic for future consideration.

5.8 User experience with cfengine

Since its general release in 1995, cfengine has attracted users from hundreds of sites around the world. NASA, ESA, CERN, the San Diego super-computing center, many universities

and numerous small and large companies (Nortel, Alcatel, IBM to name a few) are using cfengine to centralize the administration of their networks. Of the larger companies which have been attracted by cfengine, Hewlett Packard and Transarc Corporation are perhaps the most significant with regard to their active enthusiasm for the project. A number of intelligent features were added to cfengine on the suggestion of R. Ralston of Hewlett Packard[6] so that HP could use cfengine as a software installation engine for some of their products, and Transarc donated the materials and licenses for the DFS in order that we would develop support for their customers, many of whom were already using cfengine.

With the exception of occasional bug fixes, and the Distributed File System (DFS) supporting code (which was written by Demosthenes Skipitaris), cfengine was designed and written by the author. It comprises some 30,000 lines of C source code and has been adapted for use with nearly every kind of Unix operating system and NT.

The moderate success of cfengine seems to be attributable to two of its key features. The first of these is the usefulness of its abstract class model. The classification of machines and resources around the network enables huge networks to be described optimally within a single framework. This possibility for abstraction changes the paradigm of system automation from one of patching systems with numerous specialized scripts to one of building libraries of expert rules of behaviour. In a very real sense, the class feature makes cfengine a tool for building expert systems.

The second attribute for cfengine's success is the converging semantics of its operations. The fact that any system will converge to a stable configuration means, as one user put it, that 'things never get worse'. This may be contrasted with the approaches of nearly all other alternative systems. Finally, it is very important that the burden of work is spread around the network. By making every machine responsible for its own state, there is no bottleneck centralization. This is one of many advantages of a 'pull' model over a 'push' model. Indeed, with most alternative management systems the bottleneck is a human. File servers are more efficient at dealing with files requests than humans are at tracking systems and issuing commands to modify them. Cfengine apparently allows humans administrators to work at a higher, more intuitive level.

In spite of two separate attempts to gather feedback by questionnaires and surveys it has proven very difficult to get cfengine users to complete any formal questionnaire. The handful of users who reply to such attempts tend to be already active supporters of the software, thus the criticism tends to be at the level of minor details. In system administration, there is no on-going discussion of what is good and bad about software; rather there is an air of rivalry to see who can 'do it best'.

The on-going questionnaire which is distributed with cfengine software and which appears on the WWW site is the following:

1. *Are you using cfengine? If not, are you interested in using it? What are your reasons? If so, what version of cfengine are you using?*
2. *Approximately how many hosts does your cfengine configuration cover? What OSes?*
3. *Have you used any other system administration tools (e.g. Tivoli, Openview,Solstice?) If so, in what way were these tools good or bad compared to cfengine?*
   (a) *Ease of configuration*
   (b) *Power of expression*
   (c) *Limitations*
   (d) *Efficiency*
   (e) *Security*
4. *How would rate on a scale of 1 to 10 (where 10 is good) the following features of cfengine*
   (a) *Central configuration file*
   (b) *Classes*
   (c) *Pull rather than push in copying*

(d) *Convergence*

(e) *Mount model*

(f) *Its basic functionality*

(g) *Its reliability*

(h) *The idea of feedback through classes*

(i) *Anything else (please specify)*

5. *Do you feel that cfengine is missing any important/pressing features? If so please mention these.*
6. *Do you use cfengine mainly to warn about problems or to fix them automatically*
7. *Do you think that it is desirable to minimize the amount of cfengine output (faults, etc) which is sent to the administrator? Or would you prefer to see more?*
8. *Briefly, what kind of network model do you have? - Firewall. Would you use cfengine on the firewall? - Central file server with NFS clients (other file systems) - Many distributed file-servers.*
9. *Do you use cfengine for garbage collection? i.e. file and process tidying. If so, what kind of policy do you use to decide which types of file should be deleted?*
10. *Do you run NT? If so would you like to have the same kind of administrative model there, or is the domain server administration model adequate? How do you deal with garbage collection in NT?*
11. *Do you feel that cfengine saves you time in performing routine tasks?*
12. *Does cfengine simplify communication of system policy between cooperating administrators, by having a single configuration file(set) which is self-documenting?*

The results of this survey (from only around thirty sites) indicated that the concept of convergence was still largely not understood by those who replied. This is particularly depressing, since it is one of cfengine's main benefits over other software. Of course this does not mean that its users to not reap the benefits of convergence. Cfengine's NFS mount model is apparently not in widespread use. No reasons were given for this, but it might be reasonable to assume that few users are willing to change their old practices and adopt a global naming convention for their file-systems. Those users who are willing have indicated plans to move to use of AFS or DFS file-systems which eliminate the need for the mount model, since the purpose of the model is essentially to emulate these file-system name spaces using the NFS. From the returned questionnaires, cfengine rated high (7-10) on all of the points asked about, except for the mount model. No significant functionality was missing, but more flexibility in platform independent mount options was requested by one user. There is also a constant stream of requests for new and more acrobatic file-editing commands.

One user compared cfengine to Tivoli. In spite of Tivoli's cost and additional management abilities, above and beyond mere system administration, it was possible for the user to compare the actual configuration and maintenance models. In this user's judgement cfengine's class model was at least as powerful as Tivoli's mechanisms for selecting groups of hosts. Tivoli's strength was in encrypted communication links between clients and servers (This is allowed by cfengine but it's need is largely obviated by avoiding network dependencies). On the other hand, Tivoli has a very complex architecture of clients and servers, with many components. Cfengine is just a single program and a single file which has an appealing simplicity for basically the same functionality. Tivoli's trust model is based on CORBA and encrypted connection to trusted Tivoli management servers, while cfengine's trust model is based entirely on the security of its configuration file. This user had no conclusion in comparing the two systems since he was still in the process of evaluating both. Another user (working on behalf of IBM) slated Tivoli and favoured cfengine, hoping to eventually combine the two since Tivoli is primarily a management scheme which relies heavily on shell scripts. Owing to IBM's interests he was forced to use Tivoli, like it

Site	size
iu.hioslo.no	30 hosts, Solaris,GNU/Linux
Unix OS vendor	45 servers in Europe
American University	200 hosts, SGI, Sun
Space agency	300 hosts, SGI
Scandinavian University	450 hosts, type unknown
Telecom company	2500 hosts, Sun

Figure 5: *Some orders of magnitude for sites running cfengine from a single configuration. The names are not given, for reasons of privacy.*

or not. At least two other IBM centres are known to use cfengine on a substantial number of hosts, one of which has reported similar opinions.

Several users using Openview and similar tools complained of the verbosity of those packages, i.e. that the volume of output generated by those systems was so large that it became simply noise. The cfengine approach of silently fixing problems was cited as a positive attribute.

There have been few negative comments about cfengine. This is not surprising from users or potential users, since those who use cfengine like it and those who do not never bother to respond to the questionnaire. The harshest critics of academic work tend to be journal referees from behind their blanket of anonymity. Even the dozen or so referees which have vetted the publications on this work have made no critical comments about cfengine or the immunity model, though one referee referred to a particular system policy example as 'naive' without further qualification. In published work cfengine has been noticed but no detailed comments have been made concerning its approach; a small number of supportive comments have appeared in ref. [7]. A discussion of ways of generalizing cfengine's approach was provided in ref. [8].

Direct questioning of users and other researchers at conferences has not produced any criticism either, in spite of serious attempts by the author to provoke some. This could mean that cfengine is considered unworthy of attention, but it is more likely that it simply reflects the generality of approach (it is more difficult to criticize general approaches than specific details). The originators of alternative system administration approaches have remained largely stiff lipped when approached. Only the author of Host Factory has made any comment at all about cfengine; in this case positive, perhaps because cfengine is not directly in competition with Host Factory. There seems therefore to be no open disagreement that cfengine has adopted a sensible approach to system configuration. The direct criticisms which have been levelled at cfengine have been at the level of disagreements about cosmetics, such as the names used in the language, or minor qualms about the default behaviour. This has occasionally led to changes being made to the detailed semantics of specific operations, but only after a careful evaluation of the consequences. The chief aim has always been to preserve the idea of convergent, safe behaviour. Additional options to key operations are also frequently requested.

In addition to the user survey it is possible to gauge an impression of its suitability from messages sent to the Usenet discussion group. Figure 5.8 shows a few data points indicating the order of magnitude of the sites which use cfengine to coordinate their networks. This table indicates that the cfengine model scales, essentially without limit. This is an indication that the abstraction model is sufficient for dealing with differences between systems in a manner which is independent of the number of machines. What the table does not show is any great diversity in the machine parks at the named sites, so variation in operating system type is not reflected in these data. On the other hand, the type of operating system is just another class as far as cfengine is concerned, so this should not play any major role in the discussion.

Most users report that cfengine saves them time. Others say that, since using cfengine

they do not have less work to do, but that their work is more productive, i.e. that they can spend time on other tasks. Before adopting cfengine, many sites would use distribution schemes, such as `rdist`, `tar` together with shell scripts to perform their initial installations. Most had no way of subsequently checking the integrity of their systems.

> *Before we deployed cfengine, when we'd install a machine we'd tar up a group of files from a "central install" area, and then untar it onto a base os.*
>
> *The problems with this are as follows:*
>
> *1) Sometimes we deploy a new file to all machines that are up, and to the nasify areas, but any machines that are down when we distribute the update stay with the old version forever.*
>
> *2) Sometimes we deploy a file to all machines that are up, but we forget to update one of the "central install" areas.*
>
> *3) Each minor OS revision has it's own "central install" area, so sometimes we update the irix6.2 tree, but forget to update the irix6.3 tree.*
>
> *And so on. "It keeps getting worse".*

The immunity model solves these issues, since each host is responsible for updating itself from a file repository; also the cfengine class model makes the differences between operating system types a non-issue.

The following comment summarizes the response of most users of cfengine.

> *Quite often we had to reinstall a machine because it was simpler than trying to change configuration manually. With cfengine we are able to trim the system very comfortably. Also, if a machine is not up when the change is made, it gets fixed whenever it comes back on line.*
>
> *The fact that cfengine is being run every hour also has had the nice side effect of people doing things the right way. Before it was so easy to fix something on a machine (with the intention of adding this to the install scripts) and then forget about it. Every time the machine was reinstalled (or sometimes even rebooted) the problem came back. Now everybody is aware that many changes will be removed within the hour if they cheat.*
>
> *On the whole I think that cfengine hasn't decreased our work very much but the results have increased. We are able to increase uptime and reliability and probably also security.*

A Swiss organization adds,

> *Cfengine hides many of the system dependencies in system administration. With cfengine, I can delegate tasks to our operators, which are then able to maintain rather complex configurations.*
>
> *The common syntax on all systems simplifies the problem of representation: due to the well known syntax and the concentration on one config file it is much easier to solve a problem if the main system manager is on vacation.*

These comments indicate the benefits of the convergence approach. In particular, the reference to 'cheating' by trying to perform changes by hand is a case of the immune system moving against a careless mistake, rather than an outside threat or configuration error. This raises an important point however which has caused some problems in the tests at Oslo college.

The immunity model relies on the fact that the immune system (cfengine) will be run regularly to check the state of the system against its program. On some systems this has proven to be a problem. A comment from HAL,

> *..nearly 50 percent of our trouble calls arise because cfengine didn't get run and needs rerunning. Then the machines are fine again...*

At Oslo College, GNU/Linux machines have proven to be a problem. Normally cfengine is run hourly by `cron`. While solaris machines simply run ad infinitum, GNU/linux `cron` crashes regularly on the PC's for a reason which has never been satisfactorily determined. This means that cfengine can only be run by network connection (cfengine can then restart cron), but sometimes the network daemon crashes on GNU/linux systems also. In this case, cfengine never gets run without a direct login by a human. One speculation about this was that the clock drift on PC hardware was so large that `cron` became confused and terminated. This theory was corroborated by the large clock drift and the failure of the network time protocol NTP to keep the GNU/Linux PC's synchronized. Fortunately, owing to the adaptive lock scheme cfengine can never be run too often, so a simple script of the form

```
while ( 1 )

  cfengine
  sleep 3600

end
```

ought to suffice, perhaps along side the other alternatives. As it turned out this was not the correct explanation. The correct explanation turned out to be that cron would die accessing input files stored on NFS file-systems. This is presumably a bug in the Linux kernel. The cure has been to make a copy of the cron and cfengine input files on the local disks of GNU/Linux hosts. Since this policy was adopted no further problems have been observed with cron.

5.9 Resource efficiency

An objection which system administrators sometimes raise in connection with a new tool is that the size of a compiled program (the 'binary') makes it inefficient compared to shell programming. The Unix philosophy has always been to make lots of small programs which do one thing well, then combine them. While this is an admirable philosophy for interactive tools, it is not an efficient approach for batch programs.

Figure 6 shows some figures which help to understand this point. In the table, one sees that cfengine's binary footprint is approximately twice as large as Perl 5, and about twenty times larger than that of the Bourne shell. The resident sizes of these binaries (the amount of them paged in for execution at a given moment) is only a factor of two different in the worst case. These figures are not meaningful comparisons in themselves however, since the average perl script needs to load several modules and implement a very large user program in order to approach the functionality of cfengine. This pushes up all the size measurements by several percent. For the shell, the situation is much worse. The size given in the table is only the size of one shell interpreter. To this one must add the size of the shell commands being executed and perhaps sub-shells.

	Solaris 2.6	GNU/Linux libc5	solaris RSS
cfengine	2125096	885160	868
perl	618256	455096	756
sh	88620	318612	412
sed	246508	53784	632
awk	84560	98996	636

Figure 6: *A comparison of memory imprints compiled with gcc on Solaris 2.6 and GNU/Linux. Both systems use shared libraries. The larger size of the Solaris binaries is due to its RISC nature. The proportions are approximately the same for both systems. The size of a shell program is always compounded by the sizes of shell commands used in the script. The size of a perl program is compounded by additional modules at about two percent of the perl binary, or by shell commands which are larger. The final column is the resident memory size of the three for Solaris 2.6 binaries, taken from UCB* `ps aux` *in the run state. The difference between them here is much less, since the entire program text needn't be paged in at once.*

In order to approach the functionality of cfengine one would need to make extensive use of Unix pipes. This adds the binary imprint of each command to the interpreter and furthermore it significantly adds to the process dispatch and execution time. Pipes introduce waits and synchronization issues, while heavyweight processes add time through context switching and I/O access. Shell commands work on a line-by-line basis which is quite inefficient. Both cfengine and perl compile their input at runtime directly to memory.

It is thus difficult to gauge any real difference in the real-time memory usage while using cfengine and while using traditional script programming. What one does save is the time to process the input and the time to respond to it. Both cfengine and perl cut down on the number of waits and context switches as compared to shell. They also make a considerable saving in the time spent understanding and debugging a program. It is quite hard to write a program with bugs in the cfengine language since it operates at such a high level. The lower the level of the programming, the greater the scope for making mistakes. With the rapid increase in processor speeds and memory availability, differences in resource usage becomes an almost irrelevant issue. What is important about cfengine is the conceptual simplification it offers.

5.10 Future considerations

The theoretical basis for cfengine, combined with comments solicited and received by the author, seem to indicate that the basic concept of cfengine is sound. Any problems which exist lie in the details of implementation rather than in fundamental shortcoming. Many of these can be attributed to a lack of insight into what was really needed at the outset. Experience in using cfengine over a period of seven years has rectified many of these issues and it has been more than a year since any major new features has been sought by the user base. This, naturally, does not mean that cfengine is perfect. In the author's own opinion, it is a toy solution, built with very limited resources. Given a fresh slate then, what details of cfengine would be changed? What lessons can be learned and used to improve version 2 of cfengine?

Cfengine's least elegant feature is its language interface. This has grown fairly haphazardly over the course of its evolution, and extended accordingly, with certain solutions being introduced in such a way as to minimize the burden of change rather than to be ideal. A complete rethink of the language interface, not merely gratuitous but constructive, would be the author's first choice.

The actionsequence approach to task ordering is not completely satisfactory. It is based on the idea that cfengine will be run regularly enough to avoid repetition of global events.

It might prove to be a limitation with respect to feedback methods. However since no better approach is currently forthcoming there is no pressing need to review this immediately.

Pattern matching criteria in file parses are presently limited to regular expressions based on the filename. A general pattern matching structure including the ability to filter by file-type, owner, group and permissions, perhaps analogous to the ACL structures would be an improvement. Whether or not such structures would be widely useful is unclear, but for the sake of completeness, they need to be added. This would also help to secure against the covert phenomenon of *bluffing*, whereby users attempt to hide certain types of files by giving them false names. The ability for users to encrypt, compress and otherwise conceal files which contravene system policy is likely to be a Red Queen type contest, which cannot be won unless covert mechanisms are implemented at operatingsystem level. This parallels the American FBI debate about encryption technologies.

Interface control proved to be too simplistic when confronted with multi-homed hosts. Several changes were made to improve this part of cfengine, only to to find the introduction of routing sockets in BSD and other Unices, and lack of ioctl options in GNU/Linux broke the code for setting interface parameters. This code needs to be fixed.

The mount model, which was modelled on a scheme used at the University of Oslo, and mimics wide area file-systems like AFS and DFS, has been unpopular. It has not been criticized, but most sites seem to opt for one of the more advanced file-systems like AFS, rather than trying to coax NFS into a more orderly structure. A few users have actually praised the approach, but they are a minority. Many authors who do use the NFS prefer the automounter. Cfengine's interaction with the automounter has not always been ideal. Users who link files to automounted directories often experience 'mount storms' as cfengine stat's the destination files (on automounted partitions) and invokes a wave of automounting, with performance results.

Unix needs some way of clearing hanging zombie processes for processes whose process group reverts that of 'init'. Poorly written software leads to an accumulation of zombie processes which use up process slots. This can eventually lead to the system's demise.

Module communication needs to be improved. In particular, communication through variable passing is weak and limited. It would also be advantageous if modules could set variables.

6 Summary

In summary we return to the key questions asked at the start. As explained in the text, no unambiguous conclusions can be drawn from the evidence, however the following impressions seem to emerge:

Does the immunity model increase our understanding of the problems of system administration? Yes, in the sense that in order to automate we must find causal relationships which under-pin the operation of the system. In this respect it is similar to performance tuning. It adds a discipline which was previously absent. This discipline does not restrict what can be accomplished, but it is well-defined given an administrative policy and can yield a positive effect on the way that system administration is performed.

Does the model have any predictive power? The predictions of the model are scalability and improved stability. Evidence for both of these qualities has been observed.

Does the generalize earlier attempts? Yes, in the sense that the class model and abstraction layer make universal interface for scripting and much functionality has been absorbed into a domain specific language.

Does the immunity model save system administrators time? This is clearly true, but it does not imply that administrators will have less to do. It only means that their time can be spent on higher level issues.

Does the model result in a quicker solution of problems when they occur? Both answers to this question are possible. What is important is that the model guarantees a response,

even in the absence of a competent human. Is there a problem that an immune model cannot solve? Clearly this is so. That is not an argument against the model only an acknowledgement of the fact that there are more problems than solutions and that there is no such thing as a universal tool.

Is the immune model immune to complete failure? This depends on the nature of the operating system and on policy. In the long term, this contest is a race to stand still.

The difficulty of collecting objective evidence of the success or failure of models of system administration from wide user bases is a hindrance. Data collection will probably always be confined to a single site since foreign sites are wary of giving out information about themselves to others. This is awkward because the most valuable information comes from human reports. Machine metrics tend to have only a trite significance bordering on the intuitively obvious.

References

[1] P. Matzinger. Tolerance, danger and the extended family. *Annu. Rev. Immun.*, 12:991, 1994.

[2] M. Burgess. Computer immunology. *Proceedings of the 12th Systems Administration conference (LISA)*, page 283, 1998.

[3] M. Burgess. Automated system administration with feedback regulation. *Software practice and experience*, 28:1519, 1998.

[4] M. Burgess and D. Skipitaris. Adaptive locks for frequently scheduled tasks with unpredictable runtimes. *Proceedings of the 11th Systems Administration conference (LISA)*, page 113, 1997.

[5] P Hoogenboom and J. Lepreau. Computer system performance problem detection using time series models. *Proceedings of the USENIX Technical Conference, Summer 1993*, page 15, 1993.

[6] M. Burgess and R. Ralston. Distributed resource administration using cfengine. *Software practice and experience*, 27:1083, 1997.

[7] S. Traugott and J. Huddleston. Bootstrapping an infrastructure. Proceedings of the 12th USENIX/LISA conference on system administration.:181, 1998.

[8] A.L. 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, (SAGE/USENIX)*, page 123, 1999.