Predictable configuration management in a randomized scheduling framework

Mark Burgess and Frode Eika Sandnes

Faculty of Engineering, Oslo University College, Cort Adelers Gate 30, N-0254 Oslo, Norway

Configuration management is an essential part of system administration and component based software configuration. Autonomous configuration is becoming more important with the emergence of agent technology and heterogeneous nomadic environments. One important issue in configuration management is security, as the services provided by a system is often publicly available. This article addresses security in configuration management systems and proposes strategies for increasing security by randomized scheduling of actions constrained by a set of precedence relations. The special class of time-triggered policies is security-enhanced by adding randomized time-offsets and granularity reductions. These strategies makes it more difficult for malicious parties to detect and exploit configuration patterns resulting in a reduced quality of service and ultimately a damaged system.

Keywords: Configuration management, security, scheduling, randomization

1 Introduction

For ten years or more, considerable effort has been invested in the analysis and design of protocols for enabling distributed system administration. Amongst these are control and monitoring protocols, such as the Simple Network Management Protocol (SNMP)[ZHGW99, OBC99], and the higher level, abstract languages for policy based management[HZ89, And94, Bur95, DDLS00]. These languages address the issues of what is to be done, if a system is found to be in a particular state, or has received a particular signal. They offer varying levels of sophistication and each has strengths and weaknesses.

Another important aspect of configuration management, which has received less attention, is that of how management functions are scheduled, both in response to specific events and as a general matter of the maintenance of the system. Policy based administration, employing agents or software robots, is an administrative method which scales to large numbers of hosts[Bur98] in distributed systems, precisely because every host is essentially responsible for its own state of configuration. However, the inter-dependencies of such networked machines mean that configuration management must reflect global properties of the network community, such as delegations and server functions. This presents special challenges. If hosts are misconfigured, with respect to important peers, problems can arise. It is thus important to have predictable scheduling properties.

Predictability does not necessarily imply determinism. It is well known that users, be they local or remote, are the main cause of problem events in computer systems[Bur00a]. Some of these problems are caused unintentionally, and others are caused with malicious intent. Problem events do not usually occur in any predictable fashion; more often they occur apparently at random. The issue of how one schedules checks and counter-measures, it therefore pivotal in assuring the stability and security of the system. For the purpose of our discussion, it is convenient to view the user-system interaction as being a hostile encounter, in which users' actions are deleterious to the system. Although this is not always true, it is essentially true on average and offers a framework for discussion as a game between users and the configuration system.

Various studies have been made examining policy-based configuration management, but few have addressed the problem in such dynamical terms, as a competition between forces which tend to disorder systems, and forces which re-order them. The concept of managed objects does not generally take into

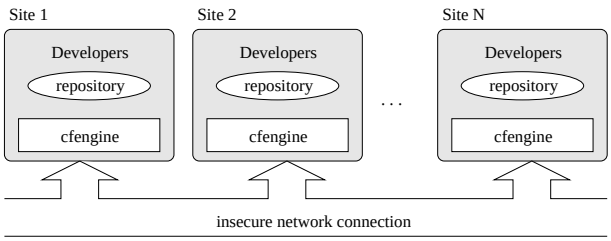

Fig. 1: cfengine is running as a agent on each host in the system.

account the effect of errors which accrue through usage. Such a dynamical view is important because, computer systems in real situations, are constantly undergoing changes. These changes result from the time-variant, concurrent actions of multiple users, and undisciplined human administrators. Today, no technology is impervious to those influences. This means that the boundary conditions of system performance are constantly changing.

The way a system responds to changes in its environment is important to configuration management. Change occurs both through planned revisions of policy and through unplanned events, such as misunderstandings between humans, undisciplined maintenance and regular usage. Throughout any change, one is interested in upholding consistency of environment for any dependent processes[KM90]. The aim of the current work is to summarize the core of an approach to system management, based on regulation of system state.

2 Configuration management and cfengine

Configuration management is an important part of both system administration and component based software development [Szy99]. Several studies have been conducted into the management of component configurations [KC99, KC00, KM90, NM98, ZS95]. These studies all identified the lack of a common framework for managing component configuration, a problem especially relevant in the context of heterogeneous component-based distributed systems. While much progress has been made with policy based configuration models, many issues remain. In this paper, we wish to address one of those issues: scheduling.

To anchor our discussion in a working model, we base our discussion around the policy-based agent cfengine[Bur95, Bur] which has been in widespread use on Unix-like systems since the mid 1990's. Cfengine provides a high level, declarative language interface for the task of distributed resource management, as well as a distributed agent for implementing policy. The advantage of policy-based configuration is that instructions automatically document system policy, and enforce the characteristics, interrelationships and dependencies of all sites from a single location. Cfengine has features similar to other distributed management systems such as IBM's Tivoli (TME 10) [Bru94, Far92, Sch99, Uel99], and its notion of policy is similar to that described in [Slo94, DDLS00], if somewhat less refined.

The cfengine agents enforce policy expressed in centrally maintained configuration files. Changes made to one such file can result in a system-wide response, or can identify actions to be taken on a single host. The high-level configuration language abstracts from the details of the different operating systems, allowing clarity of intent.

As part of its normal duties, cfengine is tasked to conduct component installation, updates, adapt out-of-the-box systems to site requirements, and perform garbage collection and security audits. The cfengine agent is run on each computer in the distributed environment and all transactions associated with the development and deployment of the software components are brokered via cfengine (See figure 1). For instance, Hewlett Packard is one large organization employing cfengine for software installation[BR97]. Some typical configuration issues, which apply to component management, include:

1. Creating files and directories with appropriate ownership and permissions.
2. Copying authorized files from a master source and assuring their permissions. Such transactions have to be secure in two senses of the word. They must prevent unauthorized parties from monitoring the

transmission, and also handle exceptional events such as copy-failure, before a file is completely installed, an operation that could corrupt the old copy of a file and leave hosts in an inconsistent state.

3. Linking files and file-trees from a master source temporarily, or permanently, using both relative links and absolute links.

Component updates are desirable whenever a new component version is released or whenever the environment is changed so that a different version of a component is needed. Cfengine can perform these tasks automatically as long all the dependencies are declared in a cfengine script.

In most distributed systems, the overall operation is dependent on availability of service. Certain server-processes must be present in the environment if the system is to operate successfully. Cfengine can be used to monitor processes, by checking for the presence, absence and behaviour of given processes and act thereupon ensuring a continuous high availability of service. It is a distributed management system where the workload is balanced evenly on the different hosts, and redundancy can easily be supported to provide fault-tolerance.

3 Classes as scheduling entities

Policy based configuration languages associate the occurrence of specified events or conditions, with responses to be carried out by an agent. Cfengine accomplishes this by classifying the state of a host, at the time of invocation, into a number of string identifiers. Some of these represent the time of invocation, others the nature of the environment, and so on. For example:

```
files:

 (linux|solaris).Hr12::

   /etc/passwd mode=0644 action=fixall inform=true
```

The class membership is described in the second line. In this instance, it specifies the class of all hosts which are of type linux or solaris, during the time interval from 12:00 hours to 12:59. Tasks to be scheduled are placed in classes which determine the host(s) on which they should be executed, or the time at which they should be executed. Host-classes are essentially labels which document the attributes of different systems. They might be based on the physical attributes of the machine, such as its operating system type, architecture, or on some human attributes, such as geographical location or departmental affiliation. Actions are placed in such classes and are only performed if the agent executes the code in an environment which belongs to one of the relevant classes. Thus, by placing actions in judiciously chosen classes, one specifies actions to be carried out on either individual machines or on arbitrary groups of machines which have a common feature relating them. Classes are evaluated in three main ways:

1. *automatically* as a result of characteristics of the environment in which the cfengine program is run, e.g. the operating system type of the host, name of the host and the day on which the script is run etc. Cfengine senses its runtime environment and switches on these classes.

2. *implicitly* by making the host a member of a named group of hosts, which then constitutes a class with the same name as the group. This is useful for specifying tasks to be performed on machines with a cultural or human connection, such as those belonging to a specific department at a university.

3. *explicitly* by defining an identifier to be a defined class in the control part of a cfengine program. This is useful for switching on and off certain tasks at run-time and is used in connection with the manual definition of classes listed below.

Creating an effective configuration for a system amounts to placing configuration actions in appropriate classes. Classes may depend on other classes, either implicitly or explicitly. This makes it possible to refer

to all sites in a given list except for a list of exceptions. Aliases can be made for commonly referred entities so that they can be referred to by meaningful names.

Classes form a number of overlapping sets, which covers the coordinate space of the distributed system (h,c,t), for different hosts h, with software components c, over time t. The aim of scheduling configuration control, is to cover this space as fully as possible, without using too many resources. This is a scheduling problem.

Classes sometimes become active in response to situations which conflict with policy, but this is not the only way in which reponses can be triggered. Some reponses are always scheduled for execution, but have no net result because of an important property referred to, in cfengine parlance, as *convergence*. Cfengine's notion of convergence towards an ideal state of configuration means that the policy agent takes no action, if no action is needed. This nullifies certain schedulable events, by virtue of their being unnecessary.

A further safe-guard which affects the scheduling of actions is a scheme of adaptive locks. All cfengine transactions are lockable entities, and each lock can persist for a certain time after the completion of an action. If the same action is scheduled too soon after completion, it will not be executed again until the lock has expired. This prevents positive feedback (recursion) storms which can sometimes take place when there are scheduled cycles, somewhat analogous to the millisecond dead-times exhibited by neurons in the brain (presumably an important factor which prevents feedback "fits" in verterbrates).

Apart from the assurances laid down by this scheduling framework, there is still the risk of randomly arriving events placing the system in jeopardy. Furthermore, repetetive (spam) attacks on specific resources need to be controlled by counter-measures. Most systems are vulnerable in different ways to exploitation due to public knowledge about their policy or configuration.

4 Management, resource allocation and scheduling

Scheduling takes many forms, such as job-shop scheduling, production scheduling, multiprocessor scheduling and so on. It can take place within any extent of time, space or other suitable covering-label.

The two main classes of scheduling are *dynamic* and *static* scheduling. Dynamical schedules can change their own execution pattern, while static ones are fully predetermined. In general, solving static scheduling problems is NP hard. This involves assigning the vertices (tasks) of an acyclic, directed graph onto a set of resources, such that the total time to process all the tasks are minimised. The total time to process all the tasks is usually referred to as the *makespan*.

An additional objective is often to achieve a short makespan while minimising the use of resources. Such multi-objective optimisation problems involve complex trade-offs and compromises, and good scheduling strategies are based on a detailed and deep understanding of the specific problem domain. Most approaches belong to the family of priority-list scheduling algorithms, differentiated by the way in which task priorities are assigned to the set of resources. Traditionally, heuristics methods have been employed in the search for high-quality solutions [KN84]. Over the last decade heuristics have been combined with modern search techniques such as simulated annealing and genetic algorithms [AD96].

In version 1 of cfengine, scheduling of tasks occurred in bulk, according to a sequence of simple list structures. This approach is extremely simple and works well enough, but it is unsatisfactory because it requires the author of a policy to understand details of the scheduling of tasks and recursion properties. In cfengine 2, this is replaced by methods described below.

4.1 Scheduling objectives and configuration management

The configuration management process can be understood as scheduling in several ways. First of all, within a single policy there is often a set of classes or triggers which are interrelated by precedence relations. These relations constrain the order in which policies can be applied, and these graphs have to be parsed. A second way in which scheduling enters, is through the reponse of the configuration system to arriving events. Should the agents activate once every hour, in order to check for policy violations, or immediately; should they start at random times, or at predictable times? Should the policies scheduled for specific times of day, occur always at the same times of day, or at variable times, perhaps random. This decision affects the

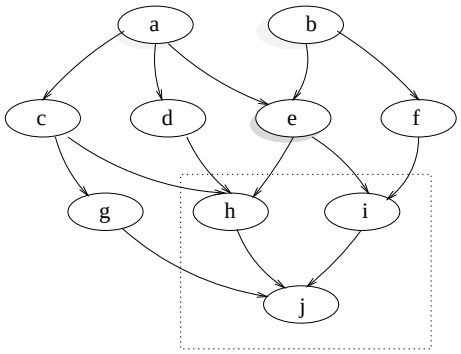

Fig. 2: Random scheduling of precedence constrained policies.

predictability of the system, and thus possibly its security in a hostile encounter. Finally, although scheduling is normally regarded as referring to extent over time, a distributed system also has two other degrees of 'spatial' extent: *h* and *c*. Scheduling tasks over different hosts, or changing the details of software components is also a possibility. It is possible to confound the predictability of software component configuration to present a 'moving target' to would-be attackers. The challenge is to accomplish this without making the system nonsensical to legitimate users. These are the issues we wish to discuss below.

A set of precedence relations can be represented by a directed graph, $G = (V,E)$, containing a finite, nonempty set of vertices, V, and a finite set of directed edges, E, connecting the vertices. The collection of vertices, $V = \{v_1, v_2, ..., v_n\}$, represents the set of n policies to be applied and the directed edges, $E = \{e_{ij}\}$, define the precedence relations that exists between these policies (e_{ij} denotes a directed edge from policy v_i to v_j).

This graph can be cyclic or acyclic. Cyclic graphs consist of *inter-cycle* and *intra-cycle* edges, where the inter-cycle edges are dependencies within a cycle and intra-cycle edges represent dependencies across cycles. When confronted with a cyclic graph then a set of transformations needs to be applied such that *intra-cycle* edges can be removed and the graph can be converted into an acyclic graph.

Configuration management is a mixture of a *dynamic* and *static* scheduling. It is dynamic in the sense that it is an ongoing real-time process where policies are triggered as a result of the environment. It is static in the sense that all policies are known *a priori*. Policies can be added, changed and removed arbitrarily in a dynamical fashion. However, this does not interfere with the static model because such changes would typically be made during a time-interval in which the configuration tool were idle or offline (in a quiescent state). The hierarchal policy model remains static, in the reference frame of each configuration, but may change dynamically between successive frames of configuration.

Cfengine's meta-policy of convergence also plays a role here, in avoiding trivial cycles. The policies of the system dictate how it is to be modified in order for it to converge towards its optimal state. Whenever there is an anomalous change in the system, the system must be modified and the policy graph dictates the partial ordering of the actions required to fix the anomaly. Since the tree primitives are designed to avoid cycles at the atomic level, the only remaining cycles are at the level of the policy tree.

5 Security and randomization

The predictability of a configuration is both an advantage and a disadvantage to the security of the system. While one would like the policy objectives to be constant, the details of implementation could legitimately vary without unacceptable loss. Predictability is often exploited by hostile users, as a means of circumventing policy. For instance, at Oslo University College, policy includes forced deletion of MP3 files older than one day, allowing users to download files for transfer to another medium, but disallowing prolonged storage. Hostile users quickly learn the time at which this tidying takes place and set up their own countermeasures in order to consume maximum resources. One way around this problem is to employ the methods of Game Theory[NM44, Bur00b, AGH99, BH95] to randomize behaviour. Cfengine *strategies* are random-

izable families of classes, in which a single family member activates with predetermined probability, on each execution of the agent.

In all scheduling problems involving precedence relations, the graph is traversed using topological sorting. Topological sorting is based around the concept of a freelist. One starts by filling the freelist with the entry nodes, i.e. nodes with no parents. At any time one can freely select, or schedule, any element in the freelist. Once all the parents of a node have been scheduled the node can be added to the freelist. Different scheduling strategies and problems differ in the way elements are selected from the freelist. Most scheduling problems involve executing a set of tasks in the shortest possible time. A popular heuristic for achieving a short schedules is the Critical Path/Most Immediate Successor First (CP/MISF) [KN84]. Tasks are scheduled with respect to their levels in the graph. Whenever there is a tie between tasks (when tasks are on the same level) the tasks with the largest number of successors are given the highest priority. The critical path defined as the longest path from an entry node to an exit node.

In configuration management, the selection of nodes from the freelist is often viewed as a trivial problem, and the freelist may, for instance, be processed from left to right, then updated, in an iterative manner. If instead one employs a strategy such as the CP/MISF, one can make modifications to a system more efficiently in a shorter time than by trivial strategy.

A system can be prone to attacks when a system is configured in a deterministic manner. By introducing randomness into the system, it becomes significantly harder to execute repetitive attacks on the system. One can therefore use a random policy implementation when selecting elements from the freelist. The randomized topological sorting algorithms can be expressed as:

```
freelist := all_entry_nodes;
unscheduled := all_nodes;
while (not unscheduled.empty())
  begin
    node := freelist[random];
    delay(random);
    process(node);          // do whatever
    scheduled.add(node);
    freelist.remove(node);
    for all nodes in unscheduled whose parents are all scheduled
      begin
        freelist.add(nodes);
        unscheduled.remove(nodes);
      end
  end
```

For example. Fig 2 illustrates a policy dependence graph. In this example, policy *e* is triggering a management response. Clearly, only the policies *h*, *i* and *j* depend on *e* and consequently needs to be applied. Since policy *j* depends on both *h* and *i*, policy *h* and *i* must be applied prior to *j*. Therefore, the free-list is first filled with the policies *h* and *i*. Policy *h* and *i* are then applied in the sequences *h*, *i* or *i*, *h*, both with a probability of 0.5.

Note that, by making a large number of observations, an external partner would be able to gradually establish some of the precedence relations, and thus partially predict updates. However, the effect of this is very much dependent on the topology of the graph. If the graph is a linear chain of policies then the randomization has no effect. At the other extreme, if the graph consists of independent policies with no inter-dependencies, then the order in which policies are applied is completely arbitrarily and impossible to predict.

Research into techniques for searching combinatorial spaces indicate that a random policy for some problems yield better results than a static left-to-right policy [Mic96]. Thus, in addition to gaining enhanced security one may also obtain a more efficient schedule by applying a random ordering. A random approach to the coverage of parameter spaces has been studied often in connection with Monte Carlo algorithms. Since events themselves occur at random, it would be interesting to study the efficacy of regular versus random scheduling of responses in minimizing the average time for expedition of the event.

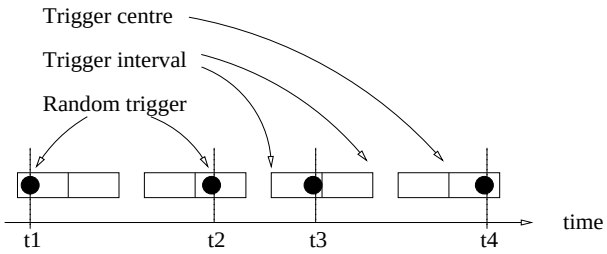

Fig. 3: Adding random offsets to time-triggered events.

6 Randomized time-triggers

One special family of scheduling classifiers, in configuration management is *time-triggers*. Such policies dictate operations which are to be conducted at specific times. This could, for example, apply to the backup of user files: all files in the user directories are to be copied to a special backup repository at 2 am every night. It could also typically refer to the cleaning up of files: e.g. all temporary user files get removed at 6 am every morning.

Time-triggered events can be detected and learned by malicious parties and users. Instead of specifying time-triggers with absolute time-stamps, it is desirable to introduce a level of randomness to certain time-triggers. There are two main ways to specify randomized time-triggers. *Absolute time with random offset* and *period with random offset*. The absolute time and with random offset specifies a given time, say every hour, or every day at a certain time with a random amount added. The period with random offset specifies a time-trigger with a certain period or frequency, also with a random offset added. Certain tasks are more naturally described using a time stamp and other using frequencies. Fig 3 illustrates time-triggered events with random offsets.

Cfengine supports the notion of local and distributed randomization in time. In local randomization, one uses a Monte-Carlo Metropolis approach to a vary amongst a list of distinct tactics. Such an approach is called a strategy, due to the connection with mixed strategies in Game Theory. For example:

```
strategies:

  { spread_load

  percent_10: "1"   # These classes get defined in
  percent_30: "3"   # these ratios of the sum 1+3+6
  percent_60: "6"   # i.e. 1/10, 3/10, 6/10
  }

 action:

 (Hr00.percent_10)||(Hr02.percent_30)::

      policy
```

In this example, one member of the set of classes `percent_10`, `percent_30`, `percent_60`, is defined on each invocation of the agent, with the indicated probability. When combined with time classes, using AND and OR, one has an action to be carried out only 40 percent of the time. Ten percent of the time, it will be done at 00:00 hour, and thirty percent of the time at 02:00 hours.

Distributed randomization is really a two-dimensional randomization. The `SplayTime` variable, in cfengine, allows load to be spread over a maximum interval of time, depending on where one is in the network name-space. This doubles as a load-balancing strategy.

```
SplayTime = (  number of minutes )
```

This declaration evaluates to a different offset on each host. A hashing algorithm selects a value between

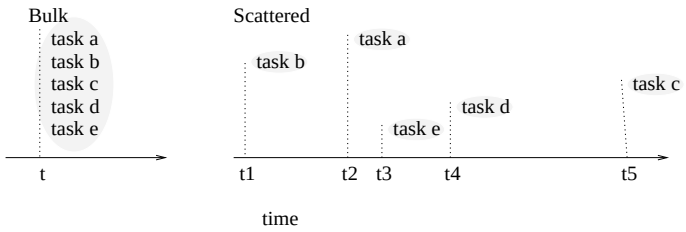

Fig. 4: Reducing the grain size and randomly scattering the fine-grained tasks.

zero and the specified number of minutes, based on the identity of the host running the agent. An additional random element can be added.

Randomization makes it much harder to identify time-trigger patterns. For example if a time trigger is set to 12:00 noon with a random offset of +/- 12 hours, then its virtually impossible to predict events. On systems with quotas, some users hog resources by identifying and exploiting fixed cleanup policies. Shared temporary directories are filled up with large files under the assumption that they will be accessible for the next X hours. Other users may be stopped from executing certain tasks as a result of this antisocial behaviour. With a randomized cleanup policy the users can never be completely assured that their files will remain in its temporary location the next moment. Such a strategy will therefore serve to deter speculation in shared resources without reducing the quality of service for ordinary users.

6.1 Reducing granularity

Configuration management tools, like cfengine, allow abstract descriptions such as time-triggered copying of multiple files and directories. The objective of this abstraction is to have clean and simple configuration scripts, uncluttered by detail. The configuration agent responds by performing the task as one entity, i.e. in one batch. These abstract descriptions define *coarse grained* operations. However, when introducing the notion of randomized time-triggered events, it is also natural to reduce the level of granularity. The set of files specified by the policy in the configuration script are split up into smaller entities such that the copying of files happens as a continuous random process. The files could be split into different groups on type, on size or number.

This serves two purposes. First, it is more difficult for parties to determine the mapping between files and their associated time-trigger. Second, proposed strategy will greatly assist load balancing as file copying operations can be highly resource intensive.

The conversion of one *coarse grained* operation into m *fine grained* operations can and should be done automatically by the configuration tool, making it transparent to the system administrator. Fig 4 illustrates the effect of dispersing a bulk operation in the time-domain.

6.2 Resolving cycles and storms

Any configuration or security system which offers an interactive dialogue, such as a system service or configurator responding with countermeasures, is vulnerable to a denial of service attack of the form: start request, system load increases, respond to load (loads further). This type of emergent condition can be avoided by breaking the cycle of dependency using some kind of policy.

Adaptive (persistent) locking, which introduces a time-limit and expiry policy on transactions, can be used to deal with repetition of an event over shorter time granules[BS97]. The transaction locking mechanism, used in cfengine for instance, can break cycles in many graphs. They are set by two parameters:

```
IfElapsed   = ( num_of_minutes )
ExpireAfter = ( num_of_minutes )
```

The former permits the execution of a transaction, only if the specified number of minutes has expired since the last similar transaction. The latter will forcibly break a cycle, when a new transaction is attempted, if it has not completed and unlocked within the time specified. For example, cycles which take an anomalous time to complete can exceed the allowed lifespan of a persistent lock, so that it has expired before the cycle comes around. These are sufficient to contain most recursive storms, caused by erroneous self-reference.

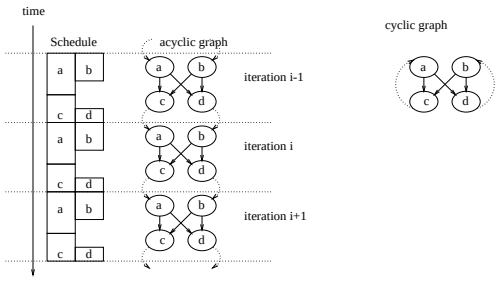

Fig. 5: Keeping schedules of iterations disjoint to ensure no intra cycle dependency violations.

Thse methods are often suffi cient, but do present a possibly exploitable loophole in the security of the system. One example of this is that attackers could aim a denial of service attack at a dependency (such as a naming service), which causes the confi guration system to become sluggish. This can distort the normal running of the system and lead to the artifi cial expiry of time limits. It is therefore of interest to combat cycles by a second method.

Locks are only effective at preventing extraneous actions. A counter example, which cannot be handled by a lock, is the opposite case where a cycle prevents something from getting started. For instance, a confi guration system which relies on its own confi guration to update itself (such as cfengine) can suffer from: syntax error, can't parse confi guration, cannot update and replace syntax error. This problem needs to be addresses by breaking the cycle into two separate pieces: one which updates the confi guration and one which does the rest. The same thing naturally applies to human resources, where one relies on, say, E-mail to report errors. The message cannot be sent or received until the system works.

There are no guarantees that the dependence hierarchy will not contain cyclic dependencies. Scheduling literature mostly assumes the graph to be acyclic, and only a few studies address the scheduling of iterative tasks. One technique for addressing this problem is that of *graph* unfolding and graph transformation - a technique in which a cyclic graph is expanded into multiple iterations followed by a transformation step where cycles are broken in strategic locations [YF97]. An alternative is to search for a confi guration where cyclic edges are removed such that the makespan is minimized [SM98]. Most real-world processes are iterative and recurrent in nature and are more naturally expressed as a cyclic task graph. It is common to view these cyclic problems as acyclic as this simplifi es the scheduling problem. A cyclic problem can be treated as acyclic as long as each scheduled cycle is disjoint from the other cycles so that no intra-cycle dependencies are violated (see Fig 5). It is therefore essential to check for cycles in the confi guration dependence hierarchy and thereafter disconnect these cycles.

There are many strategies for detecting cycles in graphs [Wei72, Rei68]. A very simple approach to detecting cycles in a graph involves representing a graph with n vertices as an $n \times n$ adjacency matrix A, where row i and column i represent policy i. Non-zero elements in the matrix denote dependencies. For example, a 1 in column 4 at row 3 indicate that policy 4 depends on policy 3, i.e. policy 3 must be applied before policy 4. The cycles of A is found by taking the matrix product of A, such that

$$A, A^1, A^2, A^3, ..., A^n \tag{1}$$

All nonzero diagonal elements will then represent a cyclic dependence of length equal to the power. For example:

$$A = \begin{bmatrix} \mathbf{0} & 0 & 1 & 0 \\ 1 & \mathbf{1} & 0 & 0 \\ 0 & 1 & \mathbf{0} & 1 \\ 0 & 0 & 1 & \mathbf{1} \end{bmatrix}, A^2 = \begin{bmatrix} \mathbf{0} & 1 & 0 & 1 \\ 1 & \mathbf{1} & 1 & 0 \\ 1 & 1 & \mathbf{1} & 1 \\ 0 & 1 & 1 & \mathbf{2} \end{bmatrix}$$

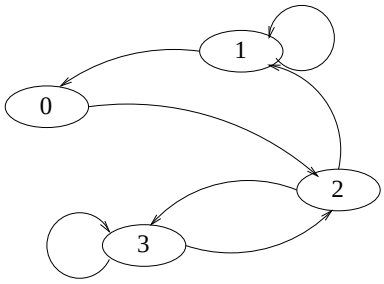

Fig. 6: Cyclic graph with two self-cycles, one cycle of length two and one cycle of length four.

$$A^3 = \begin{bmatrix} \mathbf{1} & 2 & 2 & 2 \\ 2 & \mathbf{3} & 2 & 2 \\ 2 & 4 & \mathbf{3} & 4 \\ 2 & 4 & 4 & \mathbf{5} \end{bmatrix}, A^4 = \begin{bmatrix} \mathbf{13} & 24 & 20 & 24 \\ 16 & \mathbf{29} & 24 & 28 \\ 24 & 44 & \mathbf{37} & 44 \\ 28 & 52 & 44 & \mathbf{53} \end{bmatrix}$$

Shorter cycles recur at intervals equal to the cycle length, i.e. cycles of length two will re-emerge after 4 multiplications. The matrices above show that there are two self-cycles, or cycles of length one, namely from task 0 to task 0, and from task 3 to task 3. There is one cycle of length two involving the third and the forth task (note that the self-cycles remerge in the diagonal). Further, there is one cycles of length three, and there is one cycle of length four involving all the tasks. This graph is also shown in Fig 6.

Having identified the cycles then cycles are broken off such that the cyclic graph becomes an acyclic graph where the unprocessed nodes are collected at the bottom, or towards the end, of the graph. Note that there is a relationship between the sequences generated during scheduling, and the paths obtained during reachability and data-flow analysis of distributed systems [CK94].

6.3 Decentralized scheduling and global goals

Configuration management tools administer checks and controls to a distributed network of hosts. The policy broker is the agent located at each host, whose function it is to interpret and implement policy, based on the details of its local environment. The administration task is therefore decentralized, but unified with a central or global goal. This is quite different to other scheduling scenarios where scheduling of a distributed set of resources is centralized. However, each configuration agent operates in its own domain; in cfengine, these agents operate with an almost complete disregard for the activities conducted by peer agents. This independence is intentional, and helps to avoid cycles and complex dependencies in the distributed domain. As far as we know, there is no evidence to suggest that this confers any limitations or significant restrictions on configuration management. From the viewpoint of a system administrator, the scheduling problem is centralized. However, viewed as a whole, the work of all the agents contribute toward the global optimal state for the distributed system.

7 Conclusions

This paper considers the use of scheduling analysis as a tool for ensuring effective and predictable configuration management. The policy agent cfengine is used as an example of some of these techniques.

Scheduling, in a distributed environment, is a powerful idea which extends in both time and 'space' (h,c,t). The main message of our discussion is that scheduling can be used to place reasonable limits on the behaviour of configuration systems: ensuring that policy checks are carried out often enough, but not so often that they can be exploited to overwork the system. It should neither be possible to exploit the action of the configuration system, nor prevent its action. Either of these would be regarded as a breach of security.

In a networked environment, individual hosts are exposed to threats not only from local users, but also from would-be remote attackers. Protection against so-called Denial of Service attacks has been notoriously hard to address, but we believe that the method of random scheduling of configuration described here can play a role in preventing the efficacy of such attacks. Note that even the accumulation of system garbage (i.e.

files which contravene policy) can also be regarded as a Denial of Service attack, repetitively consuming resources. In cfengine version 2, randomized strategies have been introduced as an experimental tool. It remains to be seen how effective these are in practice. We aim to present some specific practical applications in future work.

References

[AD96] Imtiaz Ahmad and Muhammed K. Dhodhi. *Multiprocessor Scheduling in a Genetic Paradigm*. Parallel Computing, vol 22, pp395-406, 1996.

[AGH99] S.P. Anderson, J.K. Goeree, and C.A. Holt. Stochastic game theory: Adjustment to equilibrium under noisy directional learning. *Working paper, University of Virginia*, 1999.

[And94] P. Anderson. Towards a high level machine configuration system. Proceedings of the Eighth Systems Administration Conference (LISA VIII) (USENIX Association: Berkeley, CA):19, 1994.

[BH95] J. Brandts and C.A. Holt. Naive bayesian learning and adjustment to equilibrium in signaling games. *Working paper, University of Virginia*, 1995.

[BR97] M. Burgess and R. Ralston. Distributed resource administration using cfengine. *Software practice and experience*, 27:1083, 1997.

[Bru94] Lee Bruno. Sophisticated Network Management. *Open Computing*, 11(12):100, December 1994.

[BS97] M. Burgess and D. Skipitaris. Adaptive locks for frequently scheduled tasks with unpredictable runtimes. *Proceedings of the Eleventh Systems Administration Conference (LISA XI) (USENIX Association: Berkeley, CA)*, page 113, 1997.

[Bur] M. Burgess. Cfengine www site. *http://www.cfengine.org*.

[Bur95] M. Burgess. A site configuration engine. *Computing systems (MIT Press: Cambridge MA)*, 8:309, 1995.

[Bur98] M. Burgess. Computer immunology. *Proceedings of the Twelth Systems Administration Conference (LISA XII) (USENIX Association: Berkeley, CA)*, page 283, 1998.

[Bur00a] M. Burgess. *Principles of Network and System Administration*. J. Wiley & Sons, Chichester, 2000.

[Bur00b] M. Burgess. Theoretical system administration. *Proceedings of the Fourteenth Systems Administration Conference (LISA XIV) (USENIX Association: Berkeley, CA)*, page 1, 2000.

[CK94] S. C. Cheung and J. Kramer. An integrated method for effective behaviour analysis of distributed systems. *Proceedings of the 16th IEEE Conference on Software Engineering*, 1994.

[DDLS00] N. Damianou, N. Dulay, E.C. Lupu, and M. Sloman. Ponder: a language for specifying security and management policies for distributed systems. *Imperial College Research Report DoC 2000/1*, 2000.

[Far92] Rik Farrow. Object-Oriented Network Management. *UNIX/world*, 9(11):93, November 1992.

[HZ89] B. Hagemark and K. Zadeck. Site: a language and system for configuring many computers as one computer site. *Proceedings of the Workshop on Large Installation Systems Administration III (USENIX Association: Berkeley, CA, 1989)*, page 1, 1989.

[KC99] Fabio Kon and Roy H. Campbell. Supporting automatic configuration of component-based distributed systems. In *Proceedings of the 5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS-99)*, pages 175–188, Berkeley, CA, 1999. USENIX Association.

[KC00] Fabio Kon and Roy H. Campbell. Dependence management in component-based distributed systems. *IEEE Concurrency*, 8(1):26–36, January/March 2000.

[KM90] Jeff Kramer and Jeff Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on software engineering*, 16(11):1293–1306, November 1990.

[KN84] Hironori Kasahara and Seinosuke Narita. *Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing*. IEEE Transactions on Computers, vol C-33, no 11, pp1023-1029, 1984.

[Mic96] Zbigniew Michalewicz. *Genetic Algorithms+Data Structures=Evolution Programs,3rd edition*. Springer, 1996.

[NM44] J.V. Neumann and O. Morgenstern. *Theory of games and economic behaviour*. Princeton University Press, Princeton, 1944.

[NM98] E. Niemelae and J. Marjeta. Dynamic configuration of distributed software components. *Lecture Notes in Computer Science*, 1543:149–150, 1998.

[OBC99] S. Omari, R. Boutaba, and O. Cherakaoui. Policies in snmpv3-based management. *Proceedings of the VI IFIP/IEEE IM conference on network management*, page 797, 1999.

[Rei68] Raymond Reitter. Scheduling parallel computations. *Journal of the Association for Computing Machinery (ACM)*, 15(4):pp590–599, 1968.

[Sch99] Stéphane Schitter. Integration of intrusion detection products in the tivoli enterprise console. Master's thesis, Eurécom Institute, June 1999.

[Slo94] M. Sloman. Policy driven management for distributed systems. *Journal of Network and Systems Management*, **2**:333, 1994.

[SM98] Frode Eika Sandnes and G. M. Megson. Improved static multiprocessor scheduling using cyclic task graphs: A genetic approach. *PARALLEL COMPUTING: Fundamentals, Applications and New Directions, North-Holland*, 12:703–710, 1998.

[Szy99] Clemens Szyperski. *Component Software - Beyond Object-Oriented Programming*. Addison Weseley, 1999.

[Uel99] Stefan Uelpenich. Extending the reach of tivoli distributed monitoring - creating a custom monitoring collection. *The Managed View*, 3(2):21–40, Spring 1999.

[Wei72] Herbert Weinblatt. A new algorithm for finding the simple cycles of a finite directed graph. *Journal of the Association for Computing Machinery (ACM)*, 19(1):pp45–56, 1972.

[YF97] Tao Yang and Cong Fu. *Heuristic Algorithms for Scheduling Iterative Task Computations on Distributed Memory Machines*. IEEE Transactions on Parallel and Distributed Systems, vol. 8m no. 6, 1997.

[ZHGW99] M. Zapf, K. Herrmann, K. Geihs, and J. Wolfang. Decentralized snmp management with mobile agents. *Proceedings of the VI IFIP/IEEE IM conference on network management*, page 623, 1999.

[ZS95] Andreas Zeller and Gregor Snelting. Handling version sets through feature logic. In Wilhelm Schäfer and Pere Botella, editors, *Proceedings of the 5th European Software Engineering Conference*, volume 989 of *Lecture Notes in Computer Science*, pages 191–204. Springer-Verlag, September 1995.