***Administration Models***

• **Reboot**: The reboot habit should, of course, be stifled. More mature

preemptive systems like Unix and perhaps NT should *almost never need to be* rebooted, indeed it can be damaging to do so. Rebooting a multi-user system is

dangerous since users might be logged in from remote locations and could lose data.

**Manual**: the default approach to system management is to allow qualified humans do everything by hand. This approach suffers from a lack of scalability. It suffers from human flaws and a lack of intrinsic documentation. Humans are not well-disciplined at documenting their work, or their intended configurations. There are also issues concerned with communication and work in a team which can interfere with the smooth running of systems.

***Control****: another approach to system administration is the use of control systems. Tivoli,* HP OpenView and Sun Solstice are examples of these. In the control approach, the system administrator follows the state of the network by defining error conditions to look for. A process on each host reports errors as they occur to the administrator. In this way the administrator has an overview of every problem on the network from his/her single location, and can either fix the problems by hand as they occur

***Immunology (self-maintenance****): a relatively new approach to system management* which is growing in popularity is the idea of equipping networked operating systems with a simple immune system.

In an immunological model, the aim is to minimize the involvement of a human being as far as possible. The tool **cfengine** introduced a technology which promotes this approach.

***Immunity and Convergence***

The immunity model is about self-sufficient maintenance and is of central importance to all scalable approaches to network management, since it is the only model which scales trivially with the number of networked hosts. The idea behind immunity is to automate host maintenance in such a way as to give each host responsibility for its own configuration.

All collective systems (including all biological phenomena) are moderated and stabilized by a cooperative principle *of feedback regulation. This regulating principle is sometimes* called the Prey-Predator scenario, because it is about *competition between different parts of a* system. When one part of the system starts to grow out of control, it tends to favor the production of an **antidote** which keeps that part in check

Similarly, the antidote cannot exist without the original system, so it cannot go so far as to destroy the original system, since it destroys itself in the process

The idea of immunity requires a notion of convergence. *Convergence means that maintenance* work (the counter force or antidote) tends to bring a host to a state of *equilibrium,* i.e. a stable state, which is the state we would actually like the system to be in. The more maintenance which is performed, the closer we approach the ideal state of the system

**Network Immunity Manager** provides an affordable, scalable, and easily managed method for providing per port intrusion detection and response to stop malicious network traffic at the edge of the network supports security policy setting and monitors network activity for anomalous behaviors that may indicate the presence of security threat s such as worms, or viruses.

**NIM** detects viruses by behavior anomaly detection on sampled traffic, and it accepts virus throttling alerts from switches.

**Network Immunity Manager** can be used to automatically detect and respond to virus attacks on per port basis. It lets you create automated threat response policies, and you can create on-demand reports for Security activity such as alerts generated and actions taken by policies

**NIM Architecture**



The sFlow standard (RFC 3176) describes a mechanism to capture traffic data in switched or routed networks. It uses a sampling technology to collect statistics from the device and is for this reason applicable to high speed networks (at gigabit speeds or higher).

**Virus Throttling** Also referred to as connection rate filtering, Virus Throttling (VT) can be used to block traffic from a host exhibiting a relatively high incidence of attempts to connect with other devices. You can set VT to block traffic permanently (until administrator re-configures to allow traffic), or to throttle (block) traffic temporarily from the host for a calculated period of time, and then allow traffic to resume. If the undesired behavior persists, the cycle is repeated. You can also use VT in Notify Only mode so that the administrator is advised when VT events occur, without blocking traffic.

***Network Organization***

A system administrator has to choose the players and assign them their

roles on the basis of the job which is intended for the computer system. There are two aspects of this to consider: the machine aspect and the human aspect.

The machine aspect relates to the use of computing machinery to achieve a functional infrastructure;

the human aspect is about the way people are deployed to build and maintain that infrastructure.

**Principle (Homogeneity/Uniformity I) *System homogeneity or uniformity means that*** *all hosts appear to be essentially the same. This makes hosts predictable for users and manageable for administrators. It allows for reuse of hardware in an emergency.*

Having chosen the necessary hardware and software, we have to address the function of each host within the community, i.e. the *delegation of specialized tasks called services to* particular hosts, and also the competition between users and hosts for resources, both local and distributed. For all of this to work with some measure of equilibrium, it has to be carefully planned and orchestrated

**LOAD BALANCING**

*For large numbers of hosts, distributed over several locations, consider a policy of delegating responsibility to a local administrator with closer knowledge of*

*the hosts' patterns of usage. Zones of responsibility allow local experts to do their jobs.*

***Bootstrapping Infrastructure***

In a network, instead of view system individually we view all systems as a virtual machine and this machine is assigned to execute a common task.

**Principle (Scalability) *Any model of system infrastructure must be able to scale*** *efficiently to large numbers of hosts (and perhaps subnets, depending on the local netmask).*

**Principle (Reliability) *Any model of system infrastructure must have reliability as one of*** *its chief goals. Down-time can often be measured in real money. MAINTENANCE*

**Corollary (Redundancy) *Reliability is safeguarded by redundancy, or backup services*** *running in parallel, ready to take over at a moment's notice.* [*RAID*](../emc/05_EMC%20Slides-PPS/Session_05.pps)

***Virtual Machine Model***

***Creating Uniformity Through Automation***

**Principle (Abstraction generalizes) *Expressing tasks in an operating-system independent*** *language reduces time spent debugging, promotes homogeneity and avoids unnecessary repetition.*

**Suggestion (Platform independent languages) *Use languages and tools which are independent*** *of operating system peculiarities, e.g. cf engine perl, python. More importantly, use the right tool for the right job.*

**Push Models and Pull Models**

*Push: the model is epitomized by the rdist program. Pushing files from a central*

location to a number of hosts is a way of forcing a file to be written to a group of hosts.

*Pull the pull model is represented by cfengine and rsync. With a pull model, each* host decides to collect files from a central repository, of its own volition. The advantage of this approach is that there is no need to open a host to control from outside, other than the trust implied by accepting configuration files from the distributing host.

**Reliability**

One of the aims of building a sturdy infrastructure is to cope with the results of failure. Failure can encompass hardware and software. It includes downtime due to physical error (power, net cables and CPUs) and also downtime due to software crashes. The net result of any failure is loss of service. Our only defense against actual failure is parallelism, or redundancy. RAID

**Cfengine: Policy Automation**

Cfengine is a system administration tool consisting of two elements: a language and a configuration engine.

Together these are used to instruct and enable all hosts on a network about how to configure and maintain themselves

Its primary function is to provide automated configuration and maintenance of computers, from a policy specification.

What makes cfengine different from scripting languages is the high level at which is operates. Rather than allowing complete programming generality, cfengine provides a set of *intelligent primitives for configuring and maintaining systems.*

As a system inevitably drifts from its ideal state, a cfengine policy brings it back to that ideal state.

Cfengine works from a central configuration, maintained from one location. That central configuration describes the entire network by referring to classes and types of host.

Here is a summary of cfengine's capabilities:

• Check and configure the network interface on network hosts.

• Edit text files for the system or for all users.

• Make and maintain symbolic links, including multiple links from a single command.

• Check and set the permissions and ownership of files.

• Tidy (delete) junk files which clutter the system.

• Systematic, automated (static) mounting of NFS file systems.

• Checking for the presence or absence of important files and file systems.

**How Does Cfengine Work?**

**Mostly Declarative Language**

Cfengine has its own mostly declarative language. This describes how things should be; with a little bit of imperative (which describes how to do it, or what to do).

Cfengine’s declarative language is used to write configuration “policy rules” or “promises” of how the system should be configured.

**Cfengine Design: Resilience**

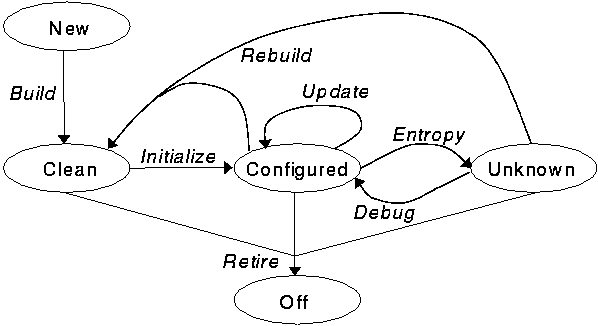
Cfengine pulls down its config from a policy server, or if the policy server is unavailable or has not been configured, from local cache at /var/cfengine/inputs.

In other words, it can run standalone, as as part of a distributed system.

Therefore Cfengine will keep running even if there is a network outage or a policy server outage. It will use its cached copy of the last known policy.

**Cfengine Design: Scalability**

Each host is responsible for carrying out checks and maintenance on itself, based on its local copy of policy. Cfengine could scale to an infinite number of hosts.



***The primary principle of Cfengine is automatic convergence from Clean or Unknown states back to Configured***

**Promises and Patterns**

Cfengine 3 is about Promises and Patterns of Promises.

**Promises**

A promise is a Cfengine policy statement - for example, that /etc/shadow is only readable by root - and it implies Cfengine will endeavor to keep that promise.

**Patterns**

A “configuration” is a design arrangement or a pattern you make with system resources. The cfengine language makes it easy to describe and implement patterns using tools like lists, bundles and regular expressions. While promises are things that are kept, the efficiencies of configuration come from how the promises form simple re-usable patterns.