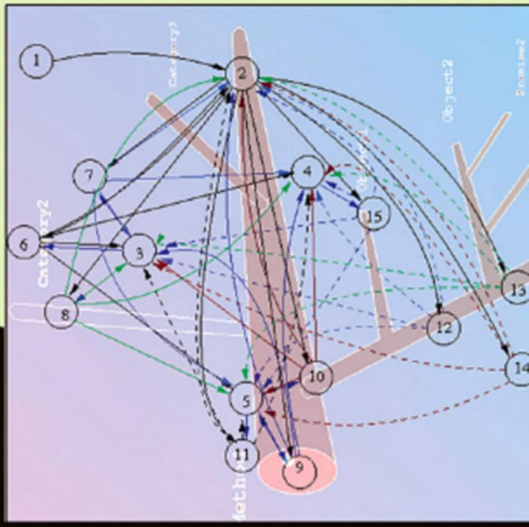
Handbook of
Network and System Administration



Jan Bergstra
Mark Burgess
(Editors)

HANDBOOK OF NETWORK AND SYSTEM ADMINISTRATION

Edited by

Jan Bergstra
Informatics Institute, University of Amsterdam
Amsterdam, The Netherlands

Mark Burgess
Faculty of Engineering, University College Oslo
Oslo, Norway

ELSEVIER

Amsterdam – Boston – Heidelberg – London – New York – Oxford – Paris
San Diego – San Francisco – Singapore – Sydney – Tokyo

Elsevier
Radarweg 29, PO Box 211, 1000 AE Amsterdam, The Netherlands
Linacre House, Jordan Hill, Oxford OX2 8DP, UK

First edition 2007

Library of Congress Cataloging-in-Publication Data
A catalog record for this book is available from the Library of Congress

British Library Cataloguing in Publication Data
A catalogue record for this book is available from the British Library

ISBN: 978-0-444-52198-9

Printed and bound in The Netherlands

07 08 09 10 11 10 9 8 7 6 5 4 3 2 1

Contents

1. The Arena

1.1. *Commentary*

The arena of system management is as broad as it is long. What began in consolidated computer centres has spread to the domestic arena of households and even to the outdoors, through a variety of personal consumer electronic items. It is an open question how much management such computing devices need in such different milieux.

In this part the authors describe some of the issues involved in the administration of systems in a more traditional organizational settings. Many matters have been left out; others are covered in later chapters, such as Chapter 3.2 about ad hoc networking in Part 3.

During the 1970s one spoke of computer operations, during the 1980s and 1990s 'system administration' or sometimes 'network administration' was the preferred appellation. Now in the 2000s we have become *service oriented* and departments and businesses are service providers. The organization is required to provide a service to customers within a 'business' framework.

Efficiency and productivity are key phrases now, and most computers in a business or university environment are located within some kind of data centre. Since the limits of performance are constantly being pushed Automation is an important approach for scalability. From an economic perspective, automation is a good investment for tasks that are frequently repeated and relatively easy to perform. As they become infrequent or difficult, the economics of automation become less attractive.

Configuration management has been one of the main areas of research in the field over the past twenty years. In other management arenas, as well as in software engineering, configuration management has a high level meaning and concerns the arrangement of resources in an enterprise. Do not be confused by the low-level interpretation of configuration management used by system administrators. This concerns the bits and bytes of files and databases that govern the properties of networks of computing devices.

This page intentionally left blank

– 1.2 –

Scaling Data Centre Services

Mark Burgess

Faculty of Engineering, University College Oslo, Room 707, Cort Adelers Gate 30, 0254 Oslo, Norway
E-mail: mark.burgess@iu.hio.no

1. Introduction

The management of computer resources is a key part of the operation of IT services. Its trends tend to proceed in cycles of distribution and consolidation. Today, the data centre plays an increasingly important role in services: rather than managing computer resources in small private rooms, many companies and organizations are once again consolidating the management of systems into specialized data centres. Such organized management is linked to the rise of de facto standards like ITIL [4,25,26] and to the increased use of outsourcing and service oriented computing which are spreading through the industry. Management standards like ITIL are rather abstract however; this chapter offers a brief technical overview of the data centre environment.

2. Computing services

Computing services are increasingly required to be scalable and efficient to meet levels of demand. The delivery of high levels of computing service is associated with three common phrases:

- High Performance Computing (HPC).
- High Volume Services (HVS).
- High Availability Services (HAS).

These phrases have subtly different meanings.

HANDBOOK OF NETWORK AND SYSTEM ADMINISTRATION
Edited by Jan Bergstra and Mark Burgess

2.1. *High performance computing*

HPC is the term used to describe a variety of CPU intensive problems. This often involves supercomputers and/or networked clusters of computers to take a large computational problem, divide it into to smaller chunks and solve those chunks in parallel. HPC requires very high-capacity and/or low-latency connections between processors. The kinds of problems solved in this way are typically of a scientific nature, such as massive numerical problems (e.g., weather and climate modelling), or search problems in bioinformatics.

For problems that require massive computational power but which are less time-critical, another strategy is *Grid computing*, an extension of the idea in which the sharing takes place over anything from a short cable to a wide area network.

2.2. *High volume services*

This refers to applications or services, normally based on content provision, that are designed to handle large numbers of transactions. HVS are a generalization of HPC to non-specifically CPU-oriented tasks. High volume services are achieved by implementing strategies for workload sharing between separate server-hosts (often called load balancing).

HVS focuses on serving large volumes of requests, and must therefore address scalability. System throughput demands a careful analysis of system bottlenecks. Load-sharing is often required to attain this goal. HVS includes the economics and the performance tuning of services. The ability to deliver reliable service levels depends on both the resources that are available in the data centre and the pattern of demand driven by the users.

2.3. *High availability services*

These are applications designed for *mission-critical* situations, where it is essential that the service is available for the clients with a *low latency*, or short response time. To achieve this, a mixture of redundancy and fault detection is implemented in the system. HAS is about ensuring that a service is available with an assured quality of response time, with target service level. One must decide what 'available' means in terms of metric goals (e.g., upper bound response time).

3. Resources

To achieve a high performance (and hence high volume) computing we need resource management. The resources a computer system has to utilize and share between different tasks, users and processes are:

- CPU.
- Disk.

- Memory.
- Network capacity.[1]

Many technologies are available to do optimization and perform this sharing, as noted in the forthcoming sections.

Computer processing is limited by *bottlenecks* or throughput limitations in a system. Sometimes these performance issues can be separated and conquered one by one, other times interdependencies make it impossible to increase performance simply by making changes to components. For instance, some properties of systems are *emergent*, i.e., they are properties of the whole system of all components, not of any individual part.

3.1. *CPU*

For CPU intensive jobs, the basic sharing tools are:

- Multi-tasking and threading at the operating system level.
- Multi-processor computers, vector machines and Beowulf clusters which allow parallelism.
- Mainframe computing.
- Grid Engine job distribution systems for parallel computing.
- MPI (Message Passing Interface) programming tools.

Flynn's taxonomy is a classification scheme for parallel computers based on whether the parallelism is exhibited in the instruction stream and/or in the data stream. This classification results in four main categories, SISD (single instruction single data), SIMD (single instruction multiple data), MISD (multiple instruction single data) and MIMD (multiple instruction multiple data). Different strategies for building computers lead to designs suited to particular classes of problems.

Load-balancing clusters operate by having all workload come through one or more load-balancing front ends, which then distribute work to a collection of back end servers. Although they are primarily implemented for improved performance, they commonly include high availability features as well. Such a cluster of computers is sometimes referred to as a server farm.

Traditionally computers are built over short distance buses with high speed communications. Dramatic improvements in network speeds have allowed wide area connection of computer components. Grid computing is the use of multiple computing nodes connecting by such wide area networks. Many grid projects exist today; many of these are designed to serve the needs of scientific projects requiring huge computational resources, e.g., bioinformatics, climate and weather modelling, and the famous Search for Extra-terrestrial Intelligence (SETI) analysis project.

3.2. *Disk*

Fast, high volume storage is a key component of delivery in content based services. Because disk-storage requires mechanical motion of armatures and disk heads, reading from

[1] The use of the term 'bandwidth' is commonly, if incorrectly, used in everyday speech. Bandwidth refers to frequency ranges. Channel capacity (or *maximum throughput*) is proportional to bandwidth – see chapter System Administration and the Scientific Method in this volume.

and writing to disk is a time-consuming business, and a potential bottleneck. Disk services that cache pages in RAM can provide a large speed-up for read/write operations.

RAID includes a number of strategies for disk performance and reliability. Redundant Arrays of Independent Disks[2] are disk arrays that have special controllers designed to optimize speed and reliability. RAID deals with two separate issues: fault tolerance and parallelism. Fault tolerant features attempt to reduce the risk of disk failure and data-loss, without down-time. They do this using redundant data encoding (parity correction codes) or mirroring. Parallism (mirroring and striping of disks) is used to increase data availability. These are often conflicting goals [11,15,16,31].

Disk striping, or parallelization of data access across parallel disks can result in an up-to-N-fold increase in disk throughput for N disks in the best case.

Error correction, sometimes referred to as *parity*, is about reliability. Error correction technology adds additional search and verification overheads, sometimes quoted at as much as 20% to the disk search response times. It can be a strategy for increasing up-time however, since RAID disks can be made hot-swappable without the need for backup.

Ten years ago networks were clearly slower than internal databuses. Today network speeds are often faster than internal databuses and there is a potential gain in both performance and administration in making disk access a network service. This has led to two architectures for disk service:

- Storage Area Networks (SAN).
- Network Attached Storage (NAS).

A Storage Area Network is an independent network of storage devices that works in place of disks connected on a local databus. This network nevertheless appears as a low level disk interface, usually SCSI. Disk storage appears as locally attached device in a computer operating system's device list. The device can be formatted for any high level filesystem.

SAN uses ISCSI (SCSI over Internet) using a dedicated network interface to connect to a storage array. This gives access to an almost unlimited amount of storage, compared to the limited numbers of disks that can be attached by the various SCSI versions. SAN can also use Fibre channel, which is another fibre-based protocol for SCSI connection.

NAS works more like a traditional network service. It appears to the computer as a filesystem service like NFS, Samba, AFS, DFS, etc. This service is mounted in Unix or attached as a logical 'drive' in Windows. It runs over a normal network protocol, like IP. Normally this would run over the same network interface as normal traffic.

3.3. *Network*

Network service provision is a huge topic that evolves faster than the timescale of publishing. Many technologies compete for network service delivery. Two main strategies exist for data delivery:

- Circuit switching.
- Packet switching.

[2] Originally the 'I' meant Inexpensive, but that now seems outdated.

These are roughly analogous to rail travel and automobile travel respectively. In circuit switching, one arranges a high-speed connection between certain fixed locations with a few main exchanges. In the packet switching, packets can make customized, individually routed journeys at the expense of some loss of efficiency.

Network throughput can be improved by adapting or tuning:

- Choice of transport technology, e.g., Fibre, Ethernet, Infiniband, etc.
- Routing policies, including the strategies above.
- Quality of service parameters in the underlying transport.

Most technological variations are based on alternative trade-offs, e.g. high speed over short distances, or stable transport over wide-areas. The appropriate technology is often a matter of optimization of performance and cost.

4. Servers and workstations

Computer manufacturers or 'vendors' distinguish between computers designed to be 'workstations' and computers that are meant to be 'servers'. Strictly speaking, a server is a *process* that provides a service not necessarily a whole computer; nevertheless, this appellation has stuck in the industry.

In an ideal world one might be able to buy a computer tailored to specific needs, but mass production constraints lead manufacturers to limit product lines for home, businesses and server rooms. Home computers are designed to be cheap and are tailored for gaming and multi-media. Business computers are more expensive versions of these with different design look. Servers are designed for reliability and have their own 'looks' to appeal to data centre customers.

A server computer is often designed to make better use of miniaturization, has a more robust power supply and better chassis space and expansion possibilities. High-quality hardware is also often used in server-class systems, e.g. fast, fault tolerant RAM, and components that have been tested to higher specifications. Better power supply technology can lead to power savings in the data centre, especially when many computers are involved in a large farm of machines.

We sometimes think of servers as 'large computers', but this is an old fashioned view. Any computer can now act as a server. Basic architectural low-level redundancy is a selling point for mainframes designs, as the kind of low-level redundancy they offer cannot easily be reproduced by clustering several computers.

4.1. *Populating a data centre*

How many computers does the data centre need, and what kind? This decision cannot be made without an appraisal of cost and performance. In most cases companies and organizations do not have the necessary planning or experience to make an informed judgment at the time they need it, and they have to undergo a process of trial-and-error learning. Industry choices are often motivated by the deals offered by sellers rather than by an engineering appraisal of system requirements, since fully rational decisions are too complicated for most buyers to make.

One decision to be made in choosing hardware for a server function is the following. Should one:

- Buy lots of cheap computers and use high-level load sharing to make it perform?
- Buy expensive computers designed for low-level redundancy and performance?

There is no simple answer as to which of these two approaches is the best strategy. The answer has fluctuated over the years, as technological development and mass production have advanced. Total cost of ownership is usually greater with custom-grown systems and cheap-to-buy computers. The cost decision becomes a question of what resources are most available in a given situation. For a further discussion of this see the chapter on System Administration and Business in this volume.

Computer 'blades' are a recent strategic design aimed at data centres, to replace normal chassis computers.

- Blades are small – many of these computers can be fitted into a small spaces specially designed for racks.
- Power distribution to blades is simplified by a common chassis and can be made redundant (though one should often read the small-print).
- A common switch in each chassis allows efficient internal networking.
- VLANs can be set up internally in a chassis, or linked to outside network.
- Blade chassis are increasingly designed for the kind of redundancy that is needed in a large scale computing operation, with two of everything provided as options (power, switch, etc.).
- They often have a separate network connection for an out-of-band management console.

The blade servers are a design convenience, well suited to life in data centre where space, power and cooling are issues.

4.2. *Bottlenecks – why won't it go faster?*

Service managers are sometimes disappointed that expensive upgrades to not lead to improved performance. This is often because computing power is associated with the CPU frequency statistics. Throwing faster CPUs at service tasks is a naive way of increasing processing. This will only help if the service tasks are primarily CPU bound.

So where are the main performance bottlenecks in computers? This is probably the wrong question to ask. Rather we have to ask, where is the bottleneck in an *application*? The resources needed by different applications are naturally different. We must decide whether a process spends most of its time utilizing:

- CPU?
- Network?
- Disk?
- Memory?

We also need to understand what *dependencies* are present in the system which could be a limiting factor [6].

Only when we know how an application interacts with the hardware and operating system can we begin to tune the performance of the system. As always, we have to look at

the resources being used and choose a technology which solves the problem we are facing. Technologies have design trade-offs which we have to understand. For instance, we might choose Infiniband for short connection, high-speed links in a cluster, but this would be no good for connecting up a building. Distance and speed are often contrary to one another.

Read versus write performance is a key factor in limiting throughput. Write-bound operations are generally slower than read-only operations because data have to be altered and caching is harder to optimize for writing. Redundancy in subsystems has a performance cost, e.g., in RAID 5 data are written to several places at once, whereas only one of the copies has to be read. This explains why RAID level 5 redundancy slows down disk writing, for example. The combination of disk striping with each disk mirrored is often chosen as the optimal balance between redundancy and efficiency.

Applications that maintain *internal state* over long-term *sessions* versus one-off transactions often have complications that make it harder to share load using high-level redundancy. In such cases low-level redundancy, such as that used in mainframe design is likely to lead to more efficient processing because the cost of sharing state between separate computers is relatively high.

5. Designing an application architecture

5.1. *Software architecture*

In an ideal world, software engineers would be aware of data centre issues when writing application software. Service design involves concerns from the low-level protocols of the network to the high-level web experiences of the commercial Internet.

While low-level processor design aims for greater and greater integration, the dominant principle in soft-system architecture is the *separation of concerns*. Today's architypical design for network application services is the so-called *three tier architecture*: webserver, application, database (see Figure 1).

Service design based entirely on the program code would be a meaningless approach in today's networked environments. A properly functioning system has to take into account everything from the user experience to the basic resources that deliver on the service promises. Design considerations for services include:

- Correctness.
- Throughput and scalability.
- Latency (response time).
- Reliability.
- Fault tolerance.
- Disaster recovery.

High volume services require efficiency of resource use. High availability requires, on the other hand, both efficiency and a reliability strategy that is fully integrated with the software deployment strategy. Typically one uses load sharing and redundancy for this.

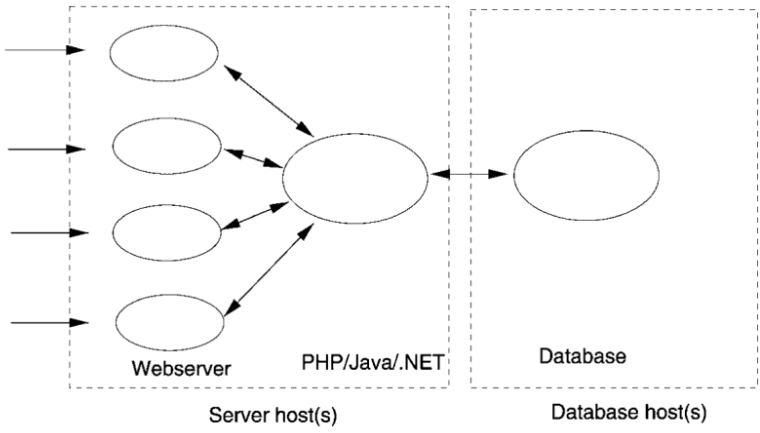


Fig. 1. A three tier architecture.

5.2. *Scalability*

Like security, scalability is a property of systems that researchers and designers like to claim or even boast, backed up with at best spurious evidence, as if the mere mention of the word could bring fortune. Like security, it has become a word to mistrust. Part of the problem is that very little attention has been given to defining what scalability means. This chapter is not the place for a comprehensive discussion of scalability, but we can summarize some simple issues.

Scalability is about characterizing the output of a system as a function of input (usually a task). The ability of a system to complete tasks depends on the extent to which the task can be broken up into independent streams that can be completed in parallel, and thus the topology of the system and its communication channels. Some simple ideas about scalability can be understood using the flow approximation of queueing systems (see chapter of System Administration and the Scientific Method in this volume) to see how the output changes as we vary the input.

We shall think of scalability as the ability of a system to deal with large amounts of input, or more correctly, we consider how increasing the input affects the efficiency with which tasks are discharged at the output. This sounds somewhat like the problem posed in queueing theory. However, whereas queueing theory deals with the subtleties of random processes, scalability is usually discussed in a pseudo-deterministic flow approximation.

If we think of rain falling (a random process of requests) then all we are interested in over time is how much rain falls and whether we can drain it away quickly enough by introducing a sufficient number of drains (processors or servers). Thus scalability is usually discussed as throughput as a function of load. Sometimes the input is a function of a number of clients, and sometimes the output is a function of the number of servers (see Figure 2). There are many ways to talk about scaling behavior.

Amdahl's law, named after computer designer Gene Amdahl, was one of the first attempts to characterize scalability of tasks in High Performance Computing, in relation to the number of processors [1]. It calculates the expected 'speed-up', or fractional increase

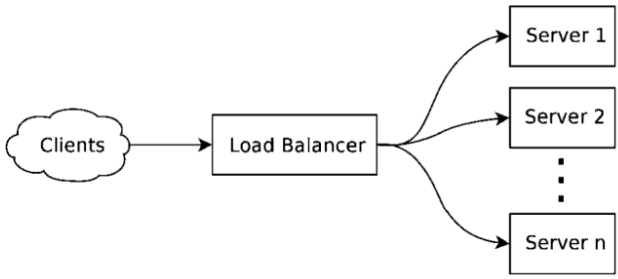


Fig. 2. The topology for low-level load sharing using a commercial dispatcher.

in performance, as a result of parallelizing part of the processing (load balancing) between N processors as:

$$S = \frac{t_{\text{serial}}}{t_{\text{parallel}}} = \frac{t(1)}{t(N)}. \tag{1}$$

Suppose a task of total size $\sigma + \pi$, which is a sum of a serial part σ and a parallelizable part π, is to be shared amongst N servers or processors and suppose that there is an overhead o associated with the parallelization (see Figure 3). Amdahl's law says that:

$$S = \frac{\sigma + \pi}{\sigma + \pi/N + o}. \tag{2}$$

In general, one does not know much about the overhead o except to say that it is positive ($o \geqslant 0$), thus we write

$$S \leqslant \frac{\sigma + \pi}{\sigma + \pi/N}. \tag{3}$$

This is usually rewritten by introducing units of the *serial fraction*, $f \equiv \sigma/(\sigma + \pi)$, so that we may write:

$$S \leqslant \frac{1}{f + (1 - f)/N}. \tag{4}$$

This fraction is never greater than $1/f$, thus even with an infinite number of processors the task cannot be made faster than the processing of the serial part. The aim of any application designer must therefore be to make the serial fraction of an application as small as possible. This is called the bottleneck of the system.

Amdahl's law is, of course, an idealization that makes a number of unrealistic assumptions, most notably that the overhead is zero. It is not clear that a given task can, in fact, be divided equally between a given number of processors. This assumes some perfect knowledge of a task with very fine granularity. Thus the speedup is strictly limited by the largest chunk (see Figure 3) not the smallest. The value of the model is that it predicts two issues that limit the performance of a server or high performance application:

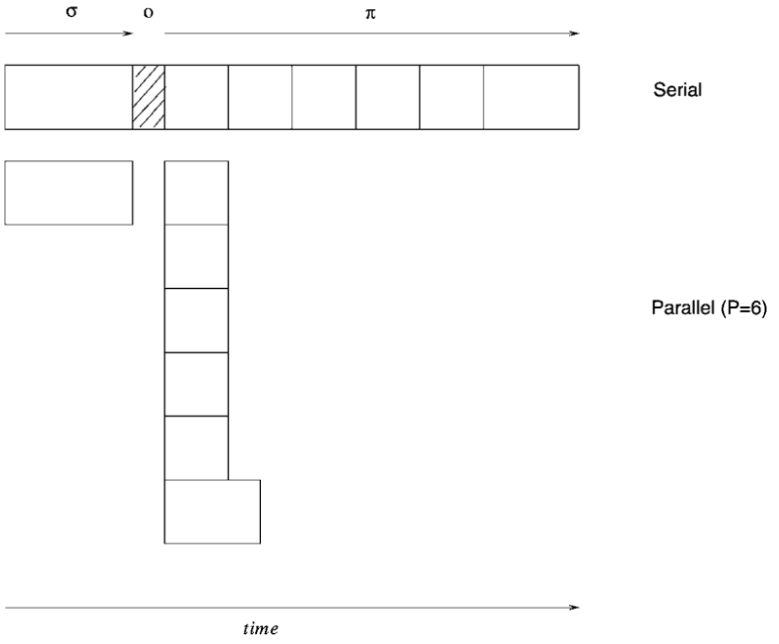


Fig. 3. Representation of Amdahl's law. A typical task has only a fraction that is parallelizable. The serial part cannot be sped up using parallel processing or load balancing. Note that one is not always able to balance tasks into chunks of equal size, moreover there is an overhead (shaded) involved in the parallelization.

- The serial fraction of the task.[3]
- The processor entropy or even-ness of the load balancing.

Amdahl's law was written with High Performance Computing in mind, but it applies also to server load balancing for network services. If one thinks of an entire job as the sum of all requests over a period of time (just as drain-water is a sum of all the individual rain-drops) then the law can also be applied to the speed up obtained in a load-balancing architecture such as that shown in Figure 2. The serial part of this task is then related to the processing of headers by the dispatcher, i.e., the dispatcher is the fundamental limitation of the system.

Determining the serial fraction of a task is not always as straightforward as one thinks. Contention for dependent resources and synchronization of tasks (waiting for resources to become available) often complicates the simple picture offered by the Amdahl law.

Karp and Flatt [21] noted that the serial part of a program must often be determined empirically. The Karp–Flatt metric is defined as the empirically determined serial fraction, found be rearranging the Amdahl formula (4) to solve for the serial fraction.

$$f = \frac{S^{-1} - N^{-1}}{1 - 1/N}. \tag{5}$$

[3] It is often quoted that a single woman can have a baby in nine months, but nine women cannot make this happen in a month.

Here we assume no overhead (it will be absorbed into the final effective serial fraction). Note that, as the number of processors becomes large, this becomes simply the reciprocal of the measured speed up. This formula has no predictive power, but it can be used to measure the performance of applications with different numbers of processors. If S is small compared to N then we know that the overhead is large and we are looking for performance issues in the task scheduler.

Another way of looking at Amdahl's law in networks has been examined in refs. [7,8] to discuss centralization versus distribution of processing. In network topology, serialization corresponds to centralization of processing, i.e. the introduction of a bottleneck by design. Sometimes this is necessary to collate data in a single location, other times designers centralize workflows unnecessarily from lack of imagination. If a centralized server is a common dependency of N clients, then it is clear that the capacity of the server C has to be shared between the N clients, so the workflow per client is

$$W \simeq \frac{C}{N}. \tag{6}$$

We say then that this architecture scales like $1/N$, assuming C to be constant. As N becomes large, the workflow per client goes to zero which is a poor scaling property. We would prefer that it were constant, which means that we must either scale the capacity C in step with N or look for a different architecture. There are two alternatives:

- Server scaling by load balancing (in-site or cross-site) $C \to CN$.
- Peer-to-peer architecture (client-based voluntary load balancing) $N \to 1$.

There is still much mistrust of non-centralized systems although the success of peer to peer systems is now hard to ignore. Scaling in the data centre cannot benefit from peer to peer scaling unless applications are designed with it in mind. It is a clear case where application design is crucial to the scaling.

5.3. *Failure modes and redundancy*

To gauge reliability system and software engineers should map out the failure modes of (or potential threats to) the system [6,18] (see also the chapter on System Reliability in this volume). Basic failure modes include:

- Power supply.
- Component failure.
- Software failure (programming error).
- Resource exhaustion and thrashing.
- Topological bottlenecks.
- Human error.

So-called 'single points of failure' in a system are warning signs of potential failure modes (see Figure 4). The principle of separation of concerns tends to lead to a tree-like structure which is all about *not* repeating functional elements and is therefore in basic conflict with the idea of redundancy (Figure 4(a)). To counter this, one can use dispatchers, load balancers and external parallelism (i.e. not built into the software, but implemented

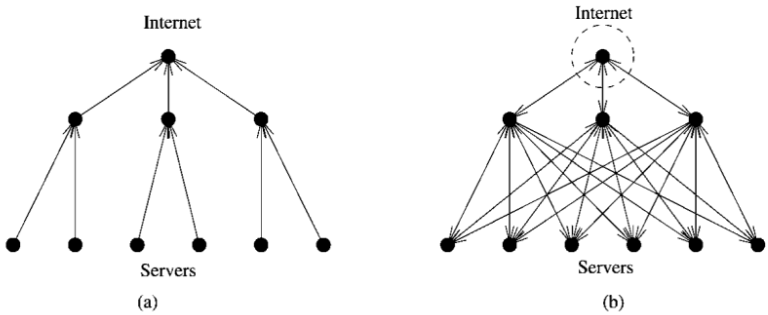


Fig. 4. Load balancing is a tricky matter if one is looking for redundancy. A load balancer is a tree – which is a structure with many intrinsic points of failure and bottlenecks.

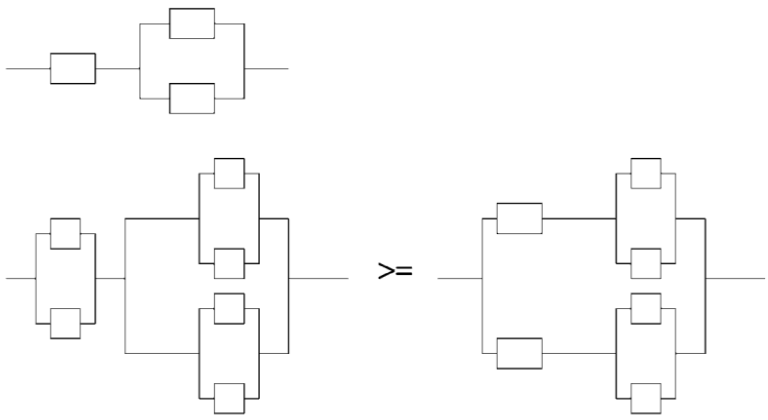


Fig. 5. Redundancy folk theorem.

afterwards). This can cure some problems, but there might still be points of failure left over (Figure 4(b)). Ideally one tries to eliminate these through further redundancy, e.g., redundant Internet service provision, redundant power supplies, etc. One should not forget the need for human redundancy in this reckoning: human resources and competence are also points of failure. Redundancy is the key issue in handling quality of service: it answers the issues of parallelism for efficiency and for fault tolerance.

When planning redundancy, there are certain thumb rules and theorems concerning system reliability. Figure 5 illustrates the folk theorem about parallisms which says that redundancy at lower system levels is always better than redundancy at higher levels. This follows from the fact that a single failure at a low level could stop an entire computing component from working. With low-level redundancy, the weakest link only brings down a low-level component, with high-level redundancy, a weakest link could bring down an entire unit of dependent components.

A minimum dependency and full redundancy strategy is shown in Figure 6. Each of the doubled components can be scaled up to n to increase thoughput.

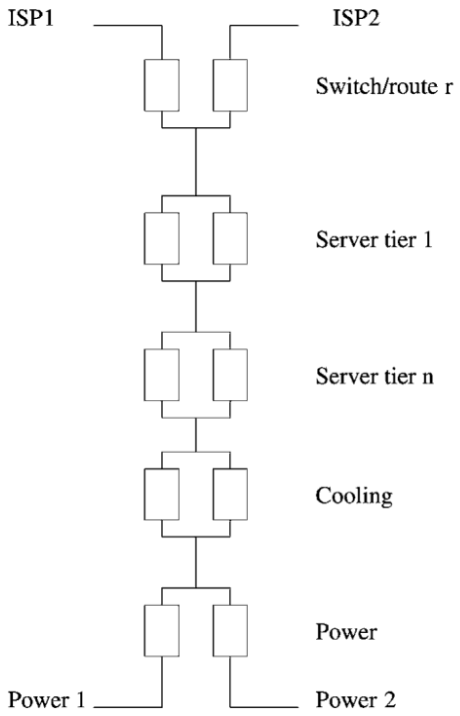


Fig. 6. Minimum strategy for complete redundancy.

Security is another concern that can be addressed in terms of failure modes. It is often distinguished from system failure due to 'natural causes' by being a 'soft failure', i.e., a failure relative to a policy rather than continuation. The distinction is probably unnecessary, as we can simply consider all failures relative to policy.

A secure system can be defined as follows [6]: *A secure system is one in which all the risks have been identified and accepted as a matter of policy.* See also the chapter by Bishop in this volume. This definition underlines the point that there is always an arbitrary threshold (or policy) in making decisions, about when faults are sufficiently serious to warrant a change of attitude, or a response.

An important enemy in system reliability is *human error*, either by mistake or incompetence. Human fault paths deal with many issues. Hostile parties expose us to risk through:

- Greed.
- Vanity.
- Bribery and blackmail.
- Revenge.
- Vandalism.
- Warfare.

Friends, on the other hand, expose us to risk by:

- Forgetfulness.
- Misunderstanding/miscommunication.

- Confusion/stress/intoxication.
- Arrogance.
- Ignorance/carelessness.
- Slowness of response.
- Procedural errors.
- Inability to deal with complexity.
- Inability to cooperate with others.

The lists are long. It is also worth asking why such incompetence could be allowed to surface. Systems themselves are clearly at fault in many ways through:

- Design faults – inadequate specification.
- Run-time fault – system does not perform within its specification.
- Emergent faults – behavior that was not planned for or explicitly designed for, often provoked by the environment.
- Changing assumptions – about technology or the environment of the system, e.g., a system is designed for one scenario that becomes replaced by another.

Documentation of possible fault modes should be prioritized by their relative likelihood.

5.4. *Safe and reliable systems*

The construction of safe systems [5,6] is both controversial and expensive. Security has many interpretations, e.g., see ISO 17799 [3] and RFC 2196 (replacing RFC 1244), or The Orange Book Trusted Computer Security Evaluation Criteria (TSEC) (now somewhat dated).

Security, like fault analysis, begins with a threat evaluation. It includes issues from human interface design to resistance to hostile actions. In all cases a preventative response is a *risk reducing strategy*.

There are many unhealthy attitudes to security amongst system designers and even administrators, e.g. 'Encryption solves 90% of the world's security problems' as stated by a naive software analyst known to the author. Security is a property of complete systems, not about installable features. Encryption deals with only a risk of information capture, which is one part of a large problem that is easily dealt with. Encryption key management, on the other hand, is almost never addressed seriously by organizations, but is clearly the basis for the reliability of encryption as a security mechanism.

Just as one must measure fault issues relative to a policy for severity, security requires us to place *trust boundaries*. The basis of everything in security is what we consider to be trustworthy (see the definition in the previous section). Technology can shift the focus and boundaries of our trust, but not eliminate the assumption of it. A system policy should make clear where the boundaries of trust lie. Some service customers insist on source code-review of applications as part of their Service Agreement. Threats include:

- Accidents.
- Human error, spilled coffee, etc.
- Attacks.
- Theft, spying, sabotage.
- Pathogenic threats like virus, Trojan horse, bugs.

- Human–computer interfaces.
- Software functionality.
- Algorithms.
- Hardware.
- Trust relationships (the line of defense).

Restriction of privilege is one defense against these matters. Access is placed on a 'need to know/do' basis and helps one to limit the scope of damage. System modelling is an important tool in predicting possible failure modes (see chapters on System Administration and the Scientific Method by the author, and the chapter on Security Management and Policies by Bishop).

In short, a well-designed system fails in a predictable way, and allows swift recovery.

6. Data centre design

The data centre is a dry and noisy environment. Humans are not well suited to this environment and should try to organize work so that they do not spend more time there than necessary. Shoes and appropriate clothing shall be worn in data centres for personal safety and to avoid contaminating the environment with skin, hair and lost personal items. Unauthorized persons should never be admitted to the data centre for both safety and security reasons.

Data centres are not something that can be sold from a shelf; they are designed within the bounds of a specific site to cope with a specific tasks. Data centres are expensive to build, especially as performance requirements increase. One must consider power supply, cooling requirements, disasters (flooding, collision damage, etc.) and redundancy due to routine failure. This requires a design and requirement overview that companies rarely have in advance. The design process is therefore of crucial importance.

It is normal for inexperienced system administrators and engineers to underestimate the power and cooling requirements for a data centre. Today, with blade chassis computers and dense rack mounting solutions it is possible to pack even more computers into a small space than ever before, thus the density of heat and power far exceeds what would be required in a normal building. It is important to be able to deliver peak power during sudden bursts of activity without tripping a fuse or circuit-breaker.

With so much heat being generated and indeed wasted in such a small area, we have to think seriously in the future about how buildings are designed. How do we re-use the heat? How can cooling be achieved without power-driven heat-exchangers?

6.1. *Power in the data centre*

Power infrastructure is the first item on the agenda when building a data centre. Having sufficient access to electrical power (with circuit redundancy) is a prerequisite for stable operation. Data centres are rated in a tier system (see Section 6.10) based on how much redundancy they can provide.

Every electrical device generates heat, including cooling equipment. Electrical power is needed to make computers run. It is also needed for lighting and cooling. Once power needs have been estimated for computers, we might have to add up to 70% again for the cost of cooling, depending on the building. In the future buildings could be designed to avoid this kind of gratuitous compounding of the heat/energy problem with artificial air-conditioning by cooling naturally with underground air intakes.

Power consumption is defined as the flow of energy, per unit time, into or out of a system. It is measured in Joules per second, or Watts or Volt-Amperes, all of which are equivalent in terms of engineering dimensions. However, these have qualitatively different interpretations for alternating current sources.

Computers use different amounts of electrical current and power depending on where they are located. Moreover, since electricity companies charge by the current that is drawn rather than the power used, the current characteristics of a device are important as a potential source of cost saving.

Electronic circuitry has two kinds of components:

- *Resistive components* absorb and dissipate power as heat. They obey Ohm's law.
- *Reactive components* absorb and return energy (like a battery), by storing it in electromagnetic fields. They include capacitors (condensers) and inductors (electro-magnetic coils).

Reactive components (inductors and capacitors) store and return most of the power they use. This means that if we wait for a full cycle of the alternating wave, we will get back most of what we put in again (however, we will still have to pay for the current). Resistive components, on the other hand, convert most of the energy going into them into heat.

In direct current (DC) circuits, the power (energy released per unit time) is simply given by $P = IV$, where I is the constant current and V is the constant voltage. In a device where a DC current can flow, reactive components to not have an effect and Ohm's law applies: $V = IR$, where R is the electrical resistance. In this case, we can write

$$P = IV = I^2R = \frac{V^2}{R}. \tag{7}$$

For an alternating current (wave) source, there are several cases to consider however. Ohm's law is no longer true in AC circuits, because capacitors and inductors can borrow power for a short time and then return it, like a wave shifting pebbles on a beach. The voltage and current are now functions of time: $I(t)$, $V(t)$. The instantaneous power consumption (the total energy absorbed per unit time) is still $P = IV = I(t)V(t)$. However, mains power is typically varying at 50–60 Hz, so this is not an true reflection of the long term behavior, only what happens on the scale of tenths of a second.

For a system driven by a single frequency (clean) wave, the short term borrowing of current is reflected by a phase shift between the voltage (wave) and the current, which is the response of the system to that driving force (pebbles). Let us call this phase shift ϕ.

$$\begin{aligned} V(t) &= V_0 \sin(2\pi f t), \\ I(t) &= I_0 \sin(2\pi f t + \phi), \end{aligned} \tag{8}$$

where f is frequency and $T = 1/f$ is the period of the wave. We can compute the average power over a number of cycles nT by computing

$$\langle P \rangle_\phi = \frac{1}{nT} \int_0^{nT} I(t)\, V(t)\, \mathrm{d}t. \tag{9}$$

We evaluate this using two trigonometric identities:

$$\sin(A + B) = \sin A \cos B + \cos A \sin B, \tag{10}$$

$$\sin^2 X = \frac{1}{2}(1 - \cos 2X) \tag{11}$$

Using the first of these to rewrite $I(t)$, we have

$$\langle P \rangle = \frac{I_0 V_0}{nT} \int_0^{nT} \left[\sin^2\left(\frac{2\pi t}{T}\right) \cos\phi + \sin\left(\frac{2\pi t}{T}\right) \cos\left(\frac{2\pi t}{T}\right) \sin\phi \right] \mathrm{d}t. \tag{12}$$

Rewriting the first term with the help of the second identity allows us to integrate the expression. Most terms vanish showing the power that is returned over a whole cycle. We are left with:

$$\langle P \rangle_\phi = \frac{1}{2} I_0 V_0 \cos\phi. \tag{13}$$

In terms of the normally quoted root-mean-square (RMS) values:

$$V_{\mathrm{rms}} = \sqrt{\frac{1}{nT} \int_0^{nT} \big(V(t)\big)^2}. \tag{14}$$

For a single frequency one has simply:

$$\begin{aligned} V_{\mathrm{rms}} &= V_0/\sqrt{2}, \\ I_{\mathrm{rms}} &= I_0/\sqrt{2}, \end{aligned} \tag{15}$$

$$\langle P \rangle_\phi = I_{\mathrm{rms}} V_{\mathrm{rms}} \cos\phi. \tag{16}$$

The RMS values are the values returned by most measuring devices and they are the values quoted on power supplies. The cosine factor is sometimes called the 'power factor' in electrical engineering literature. It is generalized below. Clearly

$$\langle P \rangle_{\phi>0} \leqslant \langle P \rangle_{\phi=0}. \tag{17}$$

In other words, the actual power consumption is not necessarily as bad as the values one would measure with volt and ammeters. This is a pity when we pay our bill, because the power companies measure current, not work-done. That means we typically pay more than we should for electrical power. Large factories can sometimes negotiate discounts based on the reactive power factor.

6.2. *Clipped and dirty power*

A direct calculation of the reduction in power transfer is impractical in most cases, and one simply defines a power factor by

$$PF = \cos\phi \equiv \frac{\langle P \rangle}{I_{\rm rms} V_{\rm rms}}. \tag{18}$$

More expensive power supply equipment is designed with 'power factor corrected' electronics that attempt to reduce the reactance of the load and lead to a simple $\phi = 0$ behavior. The more equipment we put onto a power circuit, the more erratic the power factor is likely to be. This is one reason to have specialized power supplies (incorporated with Uninterruptible Power Supplies (UPS)).

Actual data concerning power in the computer centres is hard to find, so the following is based on hearsay. Some authors claim that power factors of $\cos\phi = 0.6$ have been observed in some PC hardware. Some authors suggest that a mean value of $\cos\phi = 0.8$ is a reasonable guess. Others claim that in data centres one should assume that $\cos\phi = 1$ to correctly allow for enough headroom to deal with power demand.

Another side effect of multiple AC harmonics is that the RMS value of current is not simply $1/\sqrt{2} = 1/1.4$ of the peak value (amplitude). This leads to a new ratio called the 'crest factor', which is simply:

$$\text{Crest} = \frac{I_0}{I_{\rm rms}}. \tag{19}$$

This tells us about current peaking during high load. For a clean sinusoidal wave, this ratio is simply $\sqrt{2} = 1.4$. Some authors claim that the crest factor can be as high as 2–3 for cheap computing equipment. Some authors claim that a value of 1.4–1.9 is appropriate for power delivered by a UPS. The crest factor is also load dependent.

6.3. *Generators and batteries*

Uninterruptible Power Supplies (UPS) serve two functions: first to clean up the electrical power factor, and second to smooth out irregularities including power outages. Batteries can take over almost instantaneously from supply current, and automatic circuit breakers can start a generator to take over the main load within seconds. If the generator fails, then battery capacity will be drained.

UPS devices require huge amounts of battery capacity and even the smallest of these weighs more than a single person can normally lift. They must be installed by a competent electrician who knows the electrical installation details for the building.

Diesel powered generators should not be left unused for long periods of time, as bacteria, molds and fungi can live in the diesel and transform it into jelly over time.

6.4. *Cooling and airflow*

Environmental conditions are the second design priority in the data centre. This includes cooling and humidity regulation. Modern electronics work by dumping current to ground through semi-conducting (resistive) materials. This is the basic transistor mode of operation. This generates large amounts of heat. Essentially all of the electrical power consumed by a device ultimately becomes heat.

The aim of cooling is to prevent the temperature of devices becoming too great, or changing too quickly. If the heat increases, the electrical and mechanical resistance of all devices increases and even more energy is wasted as heat. Eventually, the electrical properties of the materials will become unsuitable for their original purpose and the device will stop working. In the worst case it could even melt.

Strictly speaking, heat is not dangerous to equipment, but temperature is. Temperature is related to the density of work done. The more concentrated heat is, the higher the temperature. Thus we want to spread heat out as much as possible.

In addition to the peak temperature, changes in temperature can be dangerous in the data centre. If temperature rises or falls too quickly it can result in mechanical stress (expansion and contraction of metallic components), or condensation in cases of high humidity.

- Polymer casings and solder-points can melt if they become too hot.
- Sudden temperature changes can lead to mechanical stresses on circuits, resulting in cracks in components and circuit failures.
- Heat increases electrical and mechanical resistance, which in turn increases heat production since heat power conversion goes like $\sim I^2R$.

Air, water and liquefied gases can be used to cool computing equipment. Which of these is best depends on budget and local environment. Good environmental design is about the constancy of the environment. Temperature, humidity, power consumption and all variables should be evenly regulated in space and time.

6.5. *Heat design*

Cooling is perhaps the most difficult aspect of data centre design to get right. The flow of air is a complicated science and our intuitions are often incorrect. It is not necessarily true that increasing the amount of cooling in the data centre will lead to better cooling of servers.

Cooling equipment is usually rated in BTUs (British Thermal Units), an old fashioned measurement of heat. 1 Watt = 3413 BTUs. A design must provide cooling for every heat generating piece of equipment in the data centre, including the humans and the cooling equipment itself. Temperature must be regulated over both space and time, to avoid gradient effects like condensation, disruptive turbulence and heat-expansion or contraction of components.

The key to temperature control is to achieve a constant ambient temperature throughout a room. The larger a room is, the harder this problem becomes. If hot spots or cold spots develop, these can lead to problems of condensation of moisture, which increases in like-

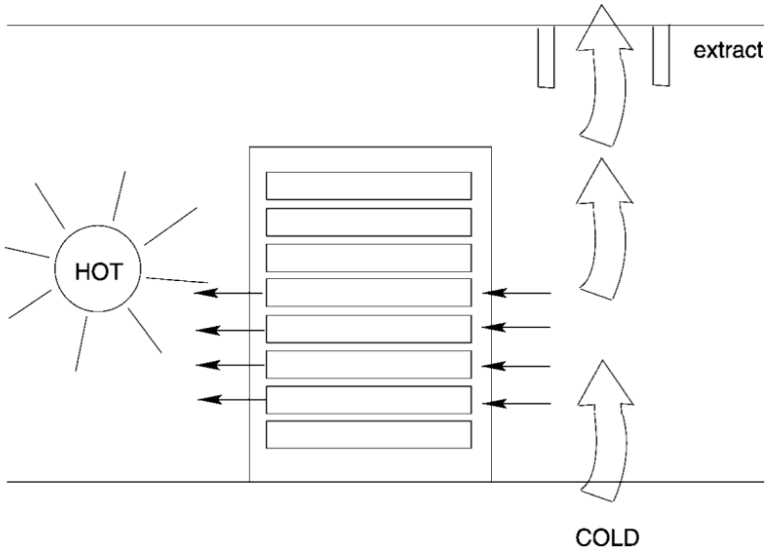


Fig. 7. A bad cooling design. Too much cold air comes through the floor, driving all the cold air past the racks into the extractor instead of passing through them. This leaves hot spots and wastes most of the cooling capacity.

lihood with increasing humidity and is both harmful to the computing equipment and can even lead to bacterial growth which is a health hazard (e.g., Legionnaires disease).

If air moves too quickly past hot devices, it will not have time to warm up and carry away heat from hot spots; in this case, most of the air volume will simply cycle from and back to the cooling units without carrying away the heat. If air moves too slowly, the temperature will rise. (See Figures 7 and 8.)

As everyone knows, hot air rises if it is able to do so. This can be both a blessing and a curse. Raised flooring is a standard design feature in more advanced data centres which allows cool air to come up through holes in the floor. This is a useful way of keeping air moving in the data centre so that no hot spots can develop. However, one should be careful not to have cold air entering too quickly from below, as this will simply cause the hot air to rise into the roof of the centre where it will be trapped. This will make some servers too hot and some too cold. It can also cause temperature sensors to be fooled into misreading the ambient temperature of the data centre, causing energy wastage from over cooling, or over regulation of cooling.

Over-cooling can, again, lead to humidity and condensation problems. Large temperature gradients can cause droplets of water to form (as in cloud formation). Hot computer equipment should not be placed too close to cooling panels as this will cause the coolers to work overtime in order to cool the apparent imbalance, and condensation-ice can form on the compressors. Eventually they will give up and water can then flood into the data centre under the flooring. The solution, unfortunately, is not merely to dry out the air, as this can lead to danger of static electrical discharges and a host of related consequences.

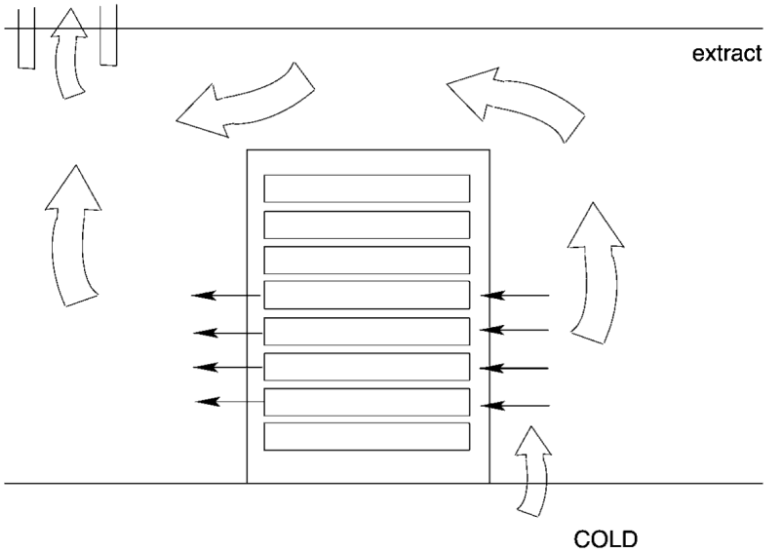


Fig. 8. A good cooling design. A trickle cold air comes through the floor, spreading out near the racks into the room where has time to collect heat. The air circulates from the cold aisle into the warm aisle where it is extracted and recycled.

6.6. *Heat, temperature and humidity*

Heat and temperature are related through the specific heat capacity (symbol c_s, also called specific heat) of a substance is defined as heat capacity per unit mass. The SI unit for specific heat capacity is the Joule per kilogramme Kelvin, J kg^{-1} K^{-1} or J/(kg K), which is the amount of energy required to raise the temperature of one kilogram of the substance by one Kelvin. Heat capacity can be measured by using calorimetry.

For small temperature variations one can think of heat capacity as being a constant property of a substance, and a linear formula for the relates temperature and heat energy.

$$\Delta Q = mc_s \Delta T, \tag{20}$$

where ΔQ is a change of heat energy (Watts $\times$ time). In other words temperature increase ΔT, in a fixed mass m of any substance, is proportional to the total power output and the time devices are left running.

If air humidity is too high, it can lead to condensation on relatively cool surfaces. Although a small amount of clean, distilled water is not a serious danger to electrical equipment, water mixed with dust, airborne dirt or salts will conduct electricity and can be hazardous.

Computer equipment can work at quite high temperatures. In ref. [24], experienced engineers suggest that the maximum data centre temperature should be 25 degrees Celcius (77 Fahrenheit), with a relative humidity of at least 40% (but no more than 80%). At less

than 30% humidity, there is a risk of static electrical discharge, causing read errors or even unrecoverable runtime failures.

Heat is transmitted to and from a system by three mechanisms:

- *Conduction* is the direct transfer of heat by material contact (molecular vibration). For this, we use the heat capacity formula above.
- *Convection* is a transport of gas around a room due to the fact that, as a gas is heated its density decreases under constant pressure, so it gets lighter and rises. If the hot air rises, cooler air must fall underneath it. This results in a circulation of air called convection. If there is not enough space to form convection cells, hot air will simply sit on top of cold air and build up.
- *Radiation* is the direct transfer of heat without material contact. Electromagnetic radiation only passes through transparent substances. However, a substance that is transparent to visible light is not necessarily transparent to heat (infra-red), which has a much longer wavelength. This is the essence of the Greenhouse effect. Visible light can enter a Greenhouse through the glass walls, this energy is absorbed and some of the energy is radiated back as heat (some of it makes plants grow). However not all of the long wavelength heat energy passes through the glass. Some is absorbed by the air and heats it up. Eventually, it will cool off by radiation of very long wavelengths, but this process is much slower than the rate at which new energy is coming in, hence the energy density increases and temperature goes up.

If a material is brought into contact with a hotter substance, heat will flow into it by *conduction*. Then, if we can remove the hotter substance e.g. by flow convection, it will carry the heat away with it. Some values of c_s for common cooling materials are shown in Table 1.

Water is about four times as efficient at carrying heat per degree rise in temperature than is air. It is therefore correspondingly better at cooling than air.

Melting temperatures in Table 2 show that it is not likely that data centre heat will cause anything in a computer to melt. The first parts to melt would be the plastics and polymers and solder connections (see Table 2).

Today's motherboards come with temperature monitoring software and hardware which shuts the computer off the CPU temperature gets too hot. CPU maximum operating temperatures lie between 60 and 90°C.

Table 1

Substance	Phase	Specific heat capacity (J/(kg K))
Air (dry)	gas	1005
Air (100% humidity)	gas	1030
Air (different source)	gas	1158
Water	liquid	4186
Copper (room temp)	solid	385
Gold (room temp)	solid	129
Silver (room temp)	solid	235

Table 2

Material	Melting point (°C)
Silicon	1410
Copper	1083
Gold	1063
Silver	879
Polymers	50–100

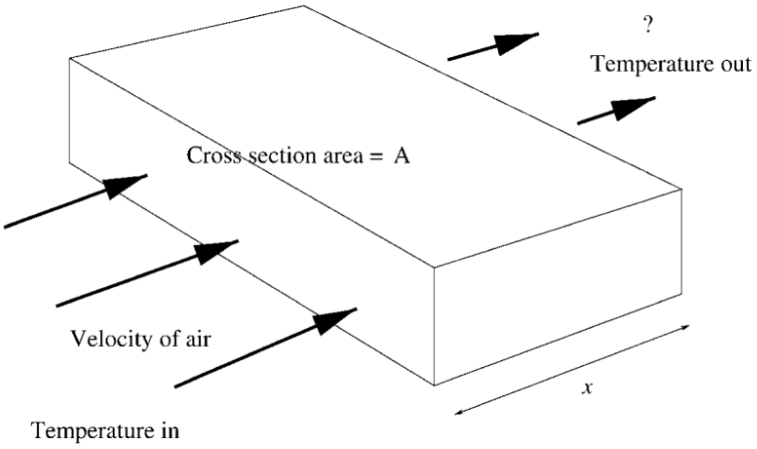


Fig. 9. Schematic illustration of air flowing through and around a rack device.

6.7. *Cooling fans*

Fans are place in computers to drive air through them. This air movement can also help to keep air circulating in the data centre. Rack doors and poorly placed fans can hinder this circulation by generating turbulence.

Cooling fans exploit the same idea as convection, to transport energy away by the actual movement of air (or sometimes fluid). The rate of flow of volume is proportional to the fixed area of the channel cross-section and the length per unit time (velocity) of the coolant passing through it.

$$\frac{\mathrm{d}V}{\mathrm{d}t} \propto A\frac{\mathrm{d}x}{\mathrm{d}t}, \tag{21}$$

$$F \propto Av. \tag{22}$$

You should bear in mind that:

- It is the temperature of components inside the computer that is the main worry in cooling design.

- The temperature of the air coming out depends on many factors, including the ambient temperature around the box, the rate of air flow through the device and around it, the amount of convection in and around the device, humidity etc.
- Air is a poor carrier of heat, so if air is getting hot, you know that the metallic components are sizzling! Also, air is a good insulator so it takes some time to warm up. If air moves too fast, it will not carry away the heat.

In very high speed computers that generate large amounts of heat quickly, liquids such as water and nitrogen are use for cooling, piped in special heat exchangers. The density of a liquid means that it is able to carry much more heat, and hence a smaller flow is needed to carry a larger amount of waste power. Water-based cooling is many times more expensive than air-cooling.

6.8. *Placement of cooling flow ducts*

The use of raised flooring is common in datacentres. Cold air usually passes underneath raised flooring from compressors and escapes into the data centre through holes in the floor.

If cold air rose from everywhere quickly and evenly, hot air would simply rise to the roof and there would be no circulation (see Figure 7). A standard strategy to maintain convectional circulation is to limit the number of holes in flooring to prevent air from circulating too quickly, and the have alternating hot and cold aisles. In a cold aisle, there are holes in the floor. In a hot aisle there are no holes in the floor, but there are cooling intakes (see Figure 8).

To ensure sufficient circulation, there should be at least ten centimetres of clearence between racks in the roof and computer racks should not be within a metre or more of cooling units.

6.9. *Fire suppression*

A data center fire suppression system uses gas rather than water or foam. There are many gaseous suppression systems that use gases hazardous to humans and environment and are therefore unsuitable. The gas used to suppress the fire should not be toxic to people nor damage the equipment causing data to be lost. Inergen and Argonite are both gasses commonly used in data centers. Inergen is a gaseous mixture composed of nitrogen, argon and carbon dioxide. Argonite is a gas composed of argon and nitrogen. Since the fire suppression systems is based on gas one should make sure that in case of gas release there is no leakage to the outside.

6.10. *Tier performance standards*

The Uptime Institute has created a 4 Tier rating system of a data centre [33]. Tier 1 level is the most basic where no redundancy is required, while Tier 4 level is a 100% redundant

data center. A system is as good as the weakest link and hence terms like 'near 4 Tier' or 'Tier 3 plus' do not exist.

The ratings use the phrases *capacity components* and *distribution paths*. Capacity components are active components like servers which deliver the service provided by the data centre. Distribution paths refer to communication infrastructure, such as network.

The Tier 1 describes a basic site, i.e., a non-redundant data center. Communication paths, cabling, power and cooling are non-redundant, the computer system is affected by a component failure and the system has to be shut down to do maintaince work.

Tier 2 describes a data center that has redundant capacity components, but non-redundant distribution paths. The computer system might be affected by a capacity component failure, and will certainly be affected by a path failure. The system might be inoperative during maintenance work.

Tier 3 describes a concurrently maintainable data center. In other words, the data center is equipped with redundant capacity components and multiple distribution paths (of which one is always active). The computer system will not be affected either in case of a capacity component failure or path failure and does not require shutdown to do maintaince work, but interruptions could occur.

Tier 4 describes a fault tolerant data center. In other words, the data center is equipped with redundant capacity component sand multiple distribution active paths. The computer system will not be affected of any worst-case failure of any capacity system or distribution element and does not require to be shut down to do maintaince work, no interruptions may occur.

7. Capacity planning

Simple results from queueing theory can be used as estimators of the capacity needs for a system [2,9]. Most service delivery systems are I/O bound, i.e. the chief bottleneck is input/output. However, in a data centre environment we have another problem that we do not experience on a small scale: power limitations.

Before designing a data center we need to work out some basic numbers, the scales and sizes of numbers involved.

- How fast can we process jobs? (CPU rate and number of parallel servers.)
- How much memory do we need: how many jobs will be in the system (queue length) and what is the average size of a job in memory?
- How many bytes per second of network arrivals and returns do we expect?
- How many bytes per second of disk activity do we expect?
- What is the power cost (including cooling) of running out servers?

7.1. *Basic queueing theory*

Service provision is usually modelled as a relationship between a client and a server (or service provider). A stream of requests arrives randomly at the input of a system, at a

mean rate of λ transactions per second, and they are added to a queue, whereupon they are serviced at a mean rate μ transactions per second by a processor.

Generalizations of this model include the possibility of multiple (parallel) queues and multiple servers. In each case one considers a single incoming stream of transaction requests; one then studies how to cope with this stream of work.

As transaction demand is a random process, queues are classified according to the type of arrival process for requests, the type of completion process and the number of servers. In the simplest form, Kendall notation of the form $A/S/n$ is used to classify different queueing models.

- A: the arrival process distribution, e.g., Poisson arrival times, deterministic arrivals, or general spectrum arrivals.
- S: the service completion distribution for job processing, e.g., Poisson renewals, etc.
- n: the number of servers processing the incoming queue.

The basic ideas about queueing can be surmised from the simplest model of a single server, memoryless queue: $M/M/1$ (see chapter by Burgess, System Administration and the Scientific Method in this volume).

Queues are described in terms of the statistical pattern of job arrivals. These processes are often approximated by so-called memoryless arrival processes with mean arrival rate λ jobs per second and mean processing rate μ jobs per second. The quantity $\rho = \lambda/\mu < 1$ is called the traffic intensity [6,19]. The expected length of the simplest $M/M/1$ queue is

$$\langle n \rangle = \sum_{n=0}^{\infty} p_n n = \frac{\rho}{1-\rho}. \tag{23}$$

Clearly as the traffic intensity ρ approaches unity, or $\lambda \to \mu$, the queue length grows out of control, as the server loses the ability to cope.

The job handling improves somewhat for s servers (here represented by the $M/M/s$ queue), where one finds a much more complicated expression which can be represented by:

$$\langle n \rangle = s\rho + P(n \geqslant s)\frac{\rho}{1-\rho}. \tag{24}$$

The probability that the queue length exceeds the number of servers $P(n \geqslant s)$ is of order ρ^2 for small load ρ, which naturally leads to smaller queues.

It is possible to show that a single queue with s servers is always at least as efficient as s separate queues with their own server. This satisfies the intuition that a single queue can be kept moving by any spare capacity in any of its s servers, whereas an empty queue that is separated from the rest will simply be wasted capacity, while the others struggle with the load.

7.2. *Flow laws in the continuum approximation*

As we noted, talking about scalability, in the memoryless approximation we can think of job queues as fluid flows. Dimensional analysis provides us with some simple continuum

relationships for the 'workflows'. Consider the following definitions:

$$\text{Arrival rate } \lambda = \frac{\text{No. of arrivals}}{\text{Time}} = \frac{A}{T}, \tag{25}$$

$$\text{Throughput } \mu = \frac{\text{No. of completions}}{\text{Time}} = \frac{C}{T}, \tag{26}$$

$$\text{Utilization } U = \frac{\text{Busy time}}{\text{Total time}} = \frac{B}{T}, \tag{27}$$

$$\text{Mean service time } S = \frac{\text{Busy time}}{\text{No. of completions}} = \frac{B}{C}. \tag{28}$$

The utilization U tells us the mean level at which resources are being scheduled in the system. The 'utilization law' notes simply that:

$$U = \frac{B}{T} = \frac{C}{T} \times \frac{B}{C} \tag{29}$$

or

$$U = \mu S. \tag{30}$$

So utilization is proportional to the rate at which jobs are completed and the mean time to complete a job. Thus it can be interpreted as the probability that there is at least one job in the system. Often this law is written $U = \lambda S$, on the assumption that in a constant flow the flow rate in is equal to the flow rate out of the system $C/T \to \lambda$. This is reasonable at low utilization because μ is sufficiently greater than λ that the process can be thought of as deterministic, thus:

$$U = \lambda S. \tag{31}$$

This handwaving rule is only applicable when the queue is lightly loaded and all jobs are essentially completed without delay; e.g., uppose a webserver receives hits at a mean rate of 1.25 hits per second, and the server takes an average of 2 milliseconds to reply. The law tells us that the utilization of the server is

$$U = 1.25 \times 0.002 = 0.0025 = 0.25\%. \tag{32}$$

This indicates to us that the system could probably work at four hundred times this rate before saturation occurs, since $400 \times 0.25 = 100\%$. This conclusion is highly simplistic, but gives a rough impression of resource consumption.

Another dimensional truism is known as Little's law of queue size. It says that the mean number of jobs in a queue $\langle n \rangle$, is equal to the product of the mean arrival rate λ (jobs per second) and the mean response time R (seconds) incurred by the queue:

$$\langle n \rangle = \lambda R. \tag{33}$$

Note that this equation has the generic form $V = IR$, similar to Ohm's law in electricity. This an analogy is a direct consequence of a simple balanced flow model. Note that R differs from the mean service time. R includes the waiting time in the queue, whereas the mean service time assumes that the job is ready to be processed.

In the $M/M/1$ queue, it is useful to characterize the expected response time of the service centre. In other words, what is the likely time a client will have to wait in order to have a task completed? From Little's law, we know that the average number of tasks in a queue is the product of the average response time and the average arrival rate, so

$$R = \frac{Q_n}{\lambda} = \frac{\langle n \rangle}{\lambda} = \frac{1}{\mu(1-\rho)} = \frac{1}{(\mu - \lambda)}. \tag{34}$$

Notice that this is finite as long as $\lambda \ll \mu$, but as $\lambda \to \mu$, the response time becomes unbounded.

Load balancing over queues is a topic for more advanced queueing theory. Weighted fair queueing and algorithms like round-robin etc., can be used to spread the load of an incoming queue amongst multiple servers, to best exploit their availability. Readers are referred to refs. [6,13,19,35] for more discussion on this.

7.3. *Simple queue estimates*

The $M/M/1$ queue is simplistic, but useful for its simplicity. Moreover, there is a point of view amongst system engineers that says: most systems will have a bottleneck (a point at which there is a single server queue) somewhere, and so we should model weakest links as $M/M/1$. Let us use the basic $M/M/1$ formulae to make some order-of-magnitude calculations for capacity planning.

For example, using the formulae for the $M/M/1$ queue, let us calculate the amount of RAM, disk rate and CPU speed to handle the average loads:

7.3.1. *Problem*

1. 10 downloads per second of image files 20 MB each.
2. 100 downloads per second of image files 20 MB each.
3. 20 downloads per second of resized images to a mobile phone, each image is 20 MB and takes 100 CPU cycles per byte to render and convert in RAM.

Given that the size of the download request is of the order 100 bytes.

7.3.2. *Estimates*

1. If we have 10 downloads per second, each of 20 MB/s, then we need to process

$$10 \times 20 = 200 \text{ MB/s}. \tag{35}$$

Ideally, all of the components in the system will be able to handle data at this rate or faster. (Note that when the averages rates are equal ($\lambda = \mu$), the queue length is not zero but infinite as this means that there will be a significant probability that

the stochastic arrival rate will be greater than the stochastic processing rate.) Let us suppose that the system disk has an average performance of 500 MB/s, then we have:

$$\begin{aligned}\lambda &= \frac{200}{20} = 10\,\text{j/s},\\ \mu &= \frac{500}{20} = 25\,\text{j/s}.\end{aligned} \tag{36}$$

Note that we cannot use MB/s for λ, or the formulae will be wrong. In the worst case, each component of the system must be able to handle this data rate. The average queue length is:

$$\langle n\rangle = \frac{\rho}{1-\rho} = \frac{\lambda}{\mu-\lambda} = 2/3 \simeq 0.7. \tag{37}$$

This tells us that there will be about 1 job in the system queue waiting to be processed at any given moment. That means the RAM we need to service the queue will be 1×100 bytes, since each job request was 100 bytes long. Clearly RAM is not much of an issue for a lightly loaded server.

The mean response time for the service is

$$R = \frac{1}{\mu-\lambda} = \frac{1}{25-10} = \frac{1}{15}\ \text{s}. \tag{38}$$

2. With the numbers above the server would fail long before this limit, since at queue size blows up to infinity as $\lambda \to 25$ and the server starts thrashing, i.e. it spends all its time managing the queue and not completing jobs.
3. Now we have an added processing time. Suppose we assume that the CPU has a clock speed of 10^9 Hz, and an average of 4 cycles per instruction. Then we process at a rate of 100 CPU cycles per byte, for all 20 MB.

$$\frac{20 \times 10^6 \times 100}{10^9} = 2\ \text{s}. \tag{39}$$

Thus we have a processing overhead of 2 seconds per download, which must be added to the normal queue response time.

$$\begin{aligned}\lambda &= \frac{400}{20} = 20\,\text{j/s},\\ \mu &= \frac{500}{20} = 25\,\text{j/s},\\ \langle n\rangle &= \frac{\rho}{1-\rho} = \frac{\lambda}{\mu-\lambda} = 4,\\ R &= \frac{1}{\mu-\lambda} + \text{CPU} = \frac{1}{25-20} + \text{CPU} = \frac{1}{5} + 2\ \text{s}.\end{aligned} \tag{40}$$

Notice that the CPU processing now dominates the service time, so we should look at increasing the speed of the CPU before worrying about anything else.

Developing simple order of magnitude estimates of this type forces system engineers to confront resource planning in a way that they would not normally do.

7.4. *Peak load handling – the reason for monitoring*

Planning service delivery based on average load levels is a common mistake of inexperienced service providers. There is always a finite risk that a level of demand will occur that is more than we can the system can provide for. Clearly we would like to avoid such a time, but this would require expensive use of redundant capacity margins (so-called 'over-provisioning').

A service centre must be prepared to either provide a constant margin of headroom, or be prepared to deploy extra server power on demand. Virtualization is one strategy for the latter, as this can allow new systems to be brought quickly on line using spurious additional hardware. But such extended service capacity has to be planned for. This is impossible if one does not understand the basic behavior of the system in advance (see the chapter on System Administration and the Scientific Method).

Monitoring is essential for understanding normal demand and performance levels in a system. Some administrators think that the purpose of monitoring is to receive alarms when faults occur, or for reviewing what happened after an incident. This is a dangerous point of view. First of all, by the time an incident has occurred, it is too late to prevent it. The point of monitoring ought to be to predict possible future occurrences. This requires a study of both average and deviant ('outlier') behavior over many weeks (see the chapter on System Administration and the Scientific Method).

The steps in planning for reliable provision include:

- Equipping the system for measurement instrumentation.
- Collecting data constantly.
- Characterizing and understand workload patterns over the short and the long term.
- Modelling and predicting performance including the likelihood and magnitude of peak activity.

Past data can indicate patterns of regular behavior and even illuminate trends of change, but they cannot truly predict the future.

In a classic failure of foresight in their first year of Internet service, the Norwegian tax directorate failed to foresee that there would be peak activity on the evening of the deadline for delivering tax returns. The system crashed and extra time had to be allocated to cope with the down-time. Such mistakes can be highly embarrassing (the tax authorities are perhaps the only institution that can guarantee that they will not lose money on this kind of outage).

Common mistakes of prediction include:

- Measuring only average workload without outlier stress points.
- Not instrumenting enough of the system to find the source of I/O bottlenecks.
- Not measuring uncertainties an data variability.
- Ignoring measurement overhead from the monitoring itself.

- Not repeating or validating measurements many times to eliminate random uncertainties.
- Not ensuring same conditions when making measurements.
- Not measuring performance under transient behavior (rapid bursts).
- Collecting data but not analyzing properly.

8. Service level estimators

Service level agreements are increasingly important as the paradigm of service provision takes over the industry. These are legal documents, transforming offers of service into obligations. Promises that are made in service agreements should thus be based on a realistic view of service delivery; for that we need to relate engineering theory and practice.

8.1. *Network capacity estimation*

The data centre engineer would often like to know what supply-margin ('over-provision') of service is required to be within a predictable margin of an Service Level Agreement (SLA) target? For example, when can we say that we are 95% certain of being able to deliver an SLA target level 80% of the time? Or better still: what is the full probability distribution for service within a given time threshold [2]?

Such estimates have to be based either on empirical data that are difficult to obtain, or simplified models that describe traffic statistics. In the late 1980s and early 1990s it became clear that the classical Poisson arrivals assumptions which worked so well for telephone traffic were inadequate to describe Internet behavior. Leland et al. [22] showed that Ethernet traffic exhibited self similar characteristics, later followed up in ref. [29]. Paxson and Floyd [28] found that application layer protocols like TELNET and FTP were modelled quite well by a Poisson model. However, the nature of data transfer proved to be bursty with long-range dependent behavior.

Web traffic is particularly interesting. Web documents can contain a variety of in-line objects such as images and multimedia. They can consist of frames and client side script. Different versions of HTTP (i.e., version 1.0 and 1.1) coexist and interact. Implementations of the TCP/IP stack might behave slightly different, etc. depending on the operating system. This leads to great variability at the packet level [12]. Refs. [14,27] have suggested several reasons for the traffic characteristics.

In spite of the evidence that for self-similarity of network traffic, it is known that if one only waits for long enough, one should see Poisson behavior [20]. This is substantiated by research done in [10] and elsewhere, and is assumed true in measurements approaching the order of 10^{12} points. However this amounts to many years of traffic [17].

8.2. *Server performance modelling*

To achieve a desired level of quality of service under any traffic conditions, extensive knowledge about how web server hardware and software interacts and affects each other

is required. Knowing the expected nature of traffic and queueing system only tells us how we might expect a system to perform; knowledge about software and hardware interaction enables performance tuning.

Slothouber [32] derived a simple model for considering web server systems founded on the notion of serial queues. Several components interact during a web conversation and a goes through different stages, with different queues at each stage. The response time is therefore an aggregate of the times spent at different levels within the web server.

Mei et al. [34] followed this line of reasoning in a subsequent paper. One of the focal points in their work was to investigate the effects of for response time and server blocking probability due to congestion in networks. High end web servers usually sit on a network with excess bandwidth to be able to shuffle load at peak rates. However, customers inherently has poorly performing networks with possible asynchronous transfer mode, such as ADSL lines. Therefore, returning ACKs in from client to server can be severely delayed. In turn this causes the request to linger for a longer time in the system than would be the case if the connecting network was of high quality. Aggregation of such requests could eventually cause the TCP and HTTP listen queue to fill up, making the server refuse new connection requests even if it is subject to fairly light load.

These models are simple and somewhat unrealistic, but possess the kind of simplicity we are looking for our comparisons.

8.3. *Service level distributions*

The fact that both arrival and service processes are stochastic processes without a well-defined mean, combined with the effect of processor sharing has on response time tails, leads to the conclusion that there is an intrinsic problem in defining average levels for service delivery. Instead, it is more fruitful to consider the likelihood of being within a certain target. For instance, one could consider 80% or 90% compliance of a service level objective. Depending on the nature of traffic the differences between these two levels can be quite significant.

This becomes quite apparent if we plot the Cumulative Distribution Functions (CDF) for the experimental and simulated response time distributions (see Figure 10) [2]. Because of the large deviations of values, going from 80% to 90% is not trivial from a service providers point of view. The CDF plots show that going from 80% to 90% in service level confidence means, more or less, a double of the service level target.

The proper way to quote service levels is thus distributions. Mean response times and rates are useless, even on the time scale of hours. A cumulative probability plot of service times is straightforwardly derived from the experimental results and gives the most realistic appraisal of the system performance. The thick cut lines in the figure show the 80% and 90% levels, and the corresponding response times associated with them. From this one can make a service agreement based on probably compliance with a service time, i.e. not merely expectation value but histogram.

For instance, one could consider 80% or 90% compliance of a maximum service level target. Depending on the nature of traffic, the differences between these two levels can be quite significant. Since customers are interested in targets, this could be the best strategy available to the provider.

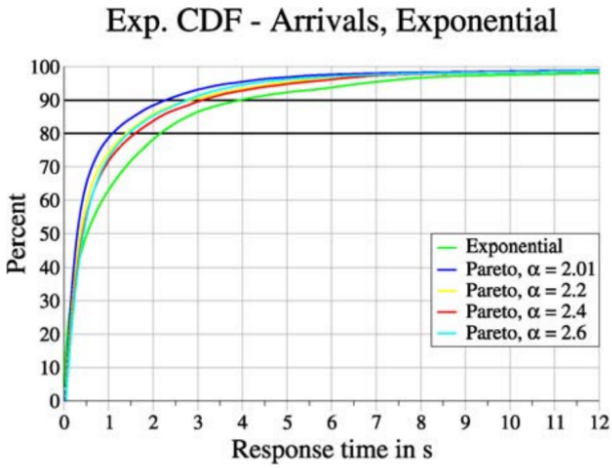


Fig. 10. The cumulative percentage distribution for response processing for sample exponential and Pareto arrivals. The thick cut lines show the 80% and 90% levels, and the corresponding response times associated with them. The theoretical estimators are too pessimistic here, which is good in the sense that it would indicate a recommended over-capacity, but the value if surely too much.

8.4. *Over-capacity planning*

The basic results for $M/M/1$ queues can be used to make over-estimates for capacity planning. The best predictive prescription (with noted flaws) is provided in ref. [2]:

1. Calculate the server rate μ for the weakest server that might handle a request. E.g. for a CPU bound process

$$\begin{aligned}\mu &= \text{av. job size} \times \frac{\text{av. instructions}}{\text{job size}} \times \frac{\text{CPU cycles}}{\text{instruction}} \times \frac{\text{time}}{\text{cycles}} \\ &= \frac{\text{av. instructions} \times \text{RISC/CISC ratio}}{\text{MHz}}. \end{aligned} \tag{41}$$

2. Estimate traffic type at maximum load.
3. Construct the Cumulative Probability Distribution as a function of response times using FCFS.
4. Make SLA promises about probability of transaction rate based on where the curves cross basic thresholds.

With this recipe, one overestimates service promise times to achieve a given level of certainty (or equivalently overestimates required capacity for a given time) in a non-linear way – the amount can be anything up to four hundred percent! This explains perhaps why most sites have seemingly greatly over-dimensioned webservers. Given the problems that can occur if a queue begins to grow, and the low cost of CPU power, this over-capacity might be worth the apparent excess.

Readers are also referred to the cost calculations described in the chapter System Administration and the Business Process by Burgess, in this volume.

9. Network load sharing

Load sharing in the data centre is an essential strategy for meeting service levels in high volume and high availability services. Figure 2 shows the schematic arrangement of servers in a load balancing scenario. Classical queueing models can also be used to predict the scaling behavior of server capacity in a data centre.

Since service provision deals with random processes, exact prediction is not an option. A realistic picture is to end up with a probability or confidence measure, e.g., what is the likelihood of being able to be within 80% or 90% of an SLA target value? The shape of this probability distribution will also answer the question: what is the correct capacity margin for a service in order to meet the SLA requirements.

Once again, in spite of the limitations of simple queueing models, we can use these as simple estimators for service capacity [9]. The total quality of service in a system must be viewed as the combined qualities of the component parts [30]. It is a known result of reliability theory [18] that low level parallelism is, in general, more efficient than high-level parallelism, in a component-based system. Thus a single $M/M/n$ queue is generally superior in performance to n separate $M/M/1$ queues (denoted $(M/M/1)^n$).

To understand load, we first have to see its effect on a single machine (Figure 11). We see that a single computer behaves quite linearly up to a threshold at which it no longer is able to cope with the resource demands placed upon it. At that point the average service delivery falls to about 50% of the maximum with huge variations (thrashing). At this point one loses essentially all predictability.

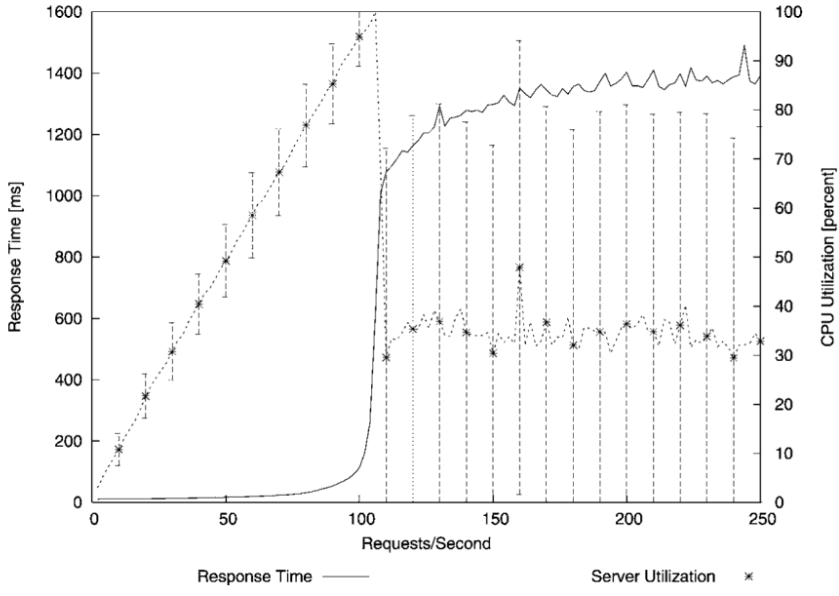


Fig. 11. Response time and CPU utilization of a single server as a function of requests per second, using Poisson distribution with exponential inter-arrival times.

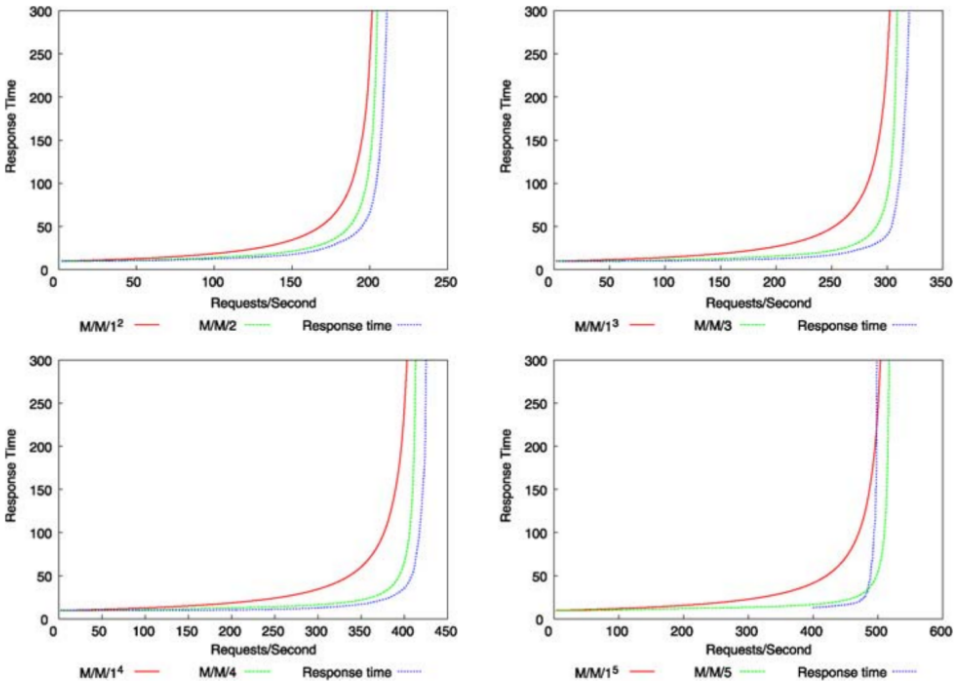


Fig. 12. A comparison of $M/M/n$ and $(M/M/1)^n$ queue estimators with response times from a Poisson traffic on real web servers.

Experiments show that, as long as one keeps *all* servers in a server farm underneath this critical threshold, performance scales approximately linearly when adding identical servers hence we can easily predict the effect of adding more servers as long as load balancing dispatchers themselves behave linearly. Figure 13 shows the addition of servers in four traffic saturation experiments. In the first graph we see that the response time is kept low in all scenarios. This fits with the assumption that even a single server should be capable of handling requests. The next graph shows double this, and all scenarios except the one with a single server are capable of handling the load. This fits the scaling predictions. With four times as much load, in the third graph, four servers just cut it. In the fourth graph we request at six times the load we see that the response rate is high for all scenarios. These results make approximate sense, yet the scaling is not exactly linear; there is some overhead when going from having 1 server to start load balancing between 2 or more servers.

As soon as a server crosses the maximum threshold, response times fall off exponentially. The challenge here is how to prevent this from happening in a data centres of inhomogeneous servers with different capacities. Dispatchers use one of a number of different sharing algorithms:

- Round robin: the classic load sharing method of taking each server in turn, without consideration of their current queue length or latency.

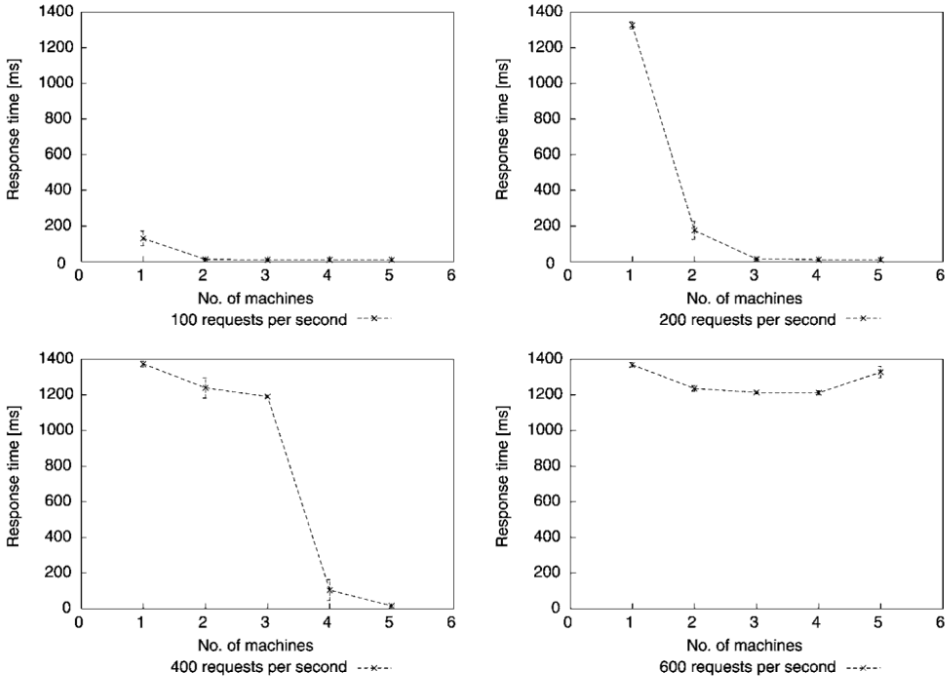


Fig. 13. How response rates scale when adding extra servers. For low traffic, adding a new server results in a discontinuous improvement in response time. At very high load, adding a server seems to reduce performance, leading to the upward curve in the last figure as the limitations of the bottleneck dispatcher become apparent.

- Least connections: the dispatcher maintains state over which back end server currently has fewest on-going TCP connections and channels new arrivals to the least connected host.
- Response time: the dispatcher measures the response time of each server in the back end by regularly testing the time it takes to establish a connection. This has many tunable parameters.

Tests show that, as long as server load is homogeneous, a round robin algorithm is most efficient. However the Least Connections algorithm holds out best against inhomogeneities [9].

10. Configuration and change management

A flexible organization is ready to adapt and make changes at any time. Some changes are in response to need and some are preemptive. All change is fraught with risk however. It is therefore considered beneficial to monitor and control the process of change in an organization in such a way that changes are documented and undergo a process of review. Two phrases are used, with some ambiguity, in this connection:

- Change management.
- Configuration management.

Change management is the planning, documentation and implementation of changes in system practices, resources and their configurations. The phrase configuration management invites some confusion, since it is used with two distinct meanings that arise from different cultures. In system administration arising from the world of Unix, configuration management is used to refer to the setup, tuning and maintenance of data that affect the behavior of computer systems. In software engineering and management (e.g., ITIL and related practices) it refers to the management of a database (CMDB) of software, hardware and components in a system, including their relationships within an organization. This is a much higher level view of 'configuration' than is taken in Unix management.

10.1. *Revision control*

One of the main recommendations about change management is the use of revision control to document changes. We want to track changes in order to:

- Be able to reverse changes that had a negative impact.
- Analyze the consequences of changes either successful or unsuccessful.

Some form of documentation is therefore necessary. This is normally done with the aid of a revision control system.

Revision control is most common in software release management, but it can also be used to process documentation or specifications for any kind of change (Figure 14). A revision control system is only a versioning tool for the actual change specification; the actual decisions should also be based on a process. There are many process models for change, including spiral development models, agile development, extreme change, followed by testing and perhaps and regret and reversal, etc. In mission critical systems more care in needed than 'try and see'. See for instance the discussions on change management in ITIL [25,26].

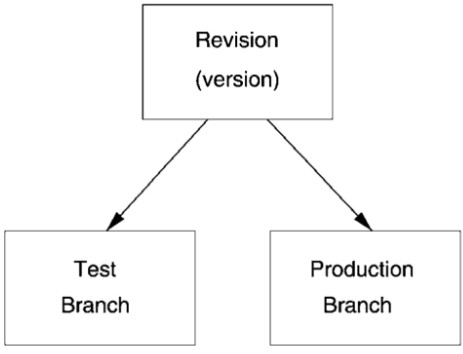


Fig. 14. Revision control – changes are tested and only later committed to production.

10.2. *'Rollback'*

In software engineering one has the notion of going back to an earlier version in the revision control system is a change is evaluated to have negative consequences. Many service managers and data centre engineers would like to have a similar view of live production systems, however there are problems associated with this.

The difference between software code and a live system is that code does not have *run-time operational state*. The behavior of a system depends in general on both system configuration and learned state that has been acquired through the running of the system. Even if one undoes changes to the configuration of a system, one cannot easily undo operational state (e.g. user sessions in an extended transaction). Thus simply undoing a configuration change will not necessarily return the behavior of a system to its previous condition. An obvious example of this is: suppose one removes all access control from a system so that it becomes overrun by hackers and viruses etc, simply restoring the access control will not remove these users or viruses from the system.

11. Human resources

In spite of the density of technology in a data centre, humans are key pieces of the service delivery puzzle. Services are increasingly about user-experience. Humans cannot therefore be removed from the system as a whole – one deals with human–computer systems [6]. Several issues can be mentioned:

- *Redundancy of workforce* is an important security in an organization. A single expert who has knowledge about an organization is a *single point of failure* just as much as any piece of equipment. Sickness or even vacation time places the organization at risk if the human services are not available to the organization.
- *Time management* is something that few people do well in any organization. One must resist the temptation to spend all day feeling busy without clearing time to work on something uninterrupted. The state of being busy is a mental state rather than a real affliction. Coping with the demands on a person's time is always a challenge. One has to find the right balance between fire-fighting and building.
- *Incident response procedures* are about making a formal response to a system event that is normally viewed to be detrimental to the system. This is a human issue because it usually involves human judgment. Some responses can be automated, e.g. using a tool like cfengine, but in general more cognition is required. This might involve a fault, a security breach or simply a failure to meet a specification, e.g. a Service Level Agreement (SLA).
- *Procedural documentation* of experience-informed procedures is an essential asset to an organization, because:
  - Past experience is not necessarily available to every data centre engineer.
  - People do not easily remember agreed procedures in a stressful situation or emergency.
  - Consistent behavior is a comfort to customers and management.

These matters are all a part of managing system predictability.

12. Concluding remarks

Data centres are complex systems that are essential components in a holistic view of IT service delivery. They require both intelligence and experience to perfect. There are many issues in data centres that system administrators have to deal with that are not covered in this overview. An excellent introduction to heuristic methods and advice can be found in ref. [23].

The main slogan for this review is simple: well designed systems fail rarely to meet their design targets and when they do so, they do so predictably. They are the result of exceptional planning and long hard experience.

Acknowledgements

I have benefited from the knowledge and work of several former students and colleagues in writing this chapter: Claudia Eriksen, Gard Undheim and Sven Ingebrigt Ulland (all now of Opera Software) and Tore Møller Jonassen. This work was supported in part by the EC IST-EMANICS Network of Excellence (#26854).

References

[1] G.M. Amdahl, *Validity of a the single processor approach to achieving large scale computer capabilities*, Proceedings of the AFTPS Spring Joint Computer Conference (1967).

[2] J.H. Bjørnstad and M. Burgess, *On the reliability of service level estimators in the data centre*, Proc. 17th IFIP/IEEE Integrated Management, volume submitted, Springer-Verlag (2006).

[3] British Standard/International Standard Organization, BS/ISO 17799 Information Technology – Code of Practice for Information Security Management (2000).

[4] British Standards Institute, *BS15000 IT Service Management* (2002).

[5] M. Burgess, *Principles of Network and System Administration*, Wiley, Chichester (2000).

[6] M. Burgess, *Analytical Network and System Administration – Managing Human–Computer Systems*, Wiley, Chichester (2004).

[7] M. Burgess and G. Canright, *Scalability of peer configuration management in partially reliable and ad hoc networks*, Proceedings of the VIII IFIP/IEEE IM Conference on Network Management (2003), 293.

[8] M. Burgess and G. Canright, *Scaling behavior of peer configuration in logically ad hoc networks*, IEEE eTransactions on Network and Service Management **1** (2004), 1.

[9] M. Burgess and G. Undheim, *Predictable scaling behavior in the data centre with multiple application servers*, Proc. 17th IFIP/IEEE Distributed Systems: Operations and Management (DSOM 2006), volume submitted, Springer (2006).

[10] J. Cao, W.S. Cleveland, D. Lin and D.X. Sun, *On the nonstationarity of Internet traffic*, SIGMETRICS '01: Proceedings of the 2001 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, ACM Press (2001), 102–112.

[11] P.M. Chen, E.K. Lee, G.A. Gibson, R.H. Katz and D.A. Patterson, *RAID: High-performance, reliable secondary storage*, ACM Comput. Surv. **26** (2) (1994), 145–185.

[12] H.-K. Choi and J.O. Limb, *A behavioral model of web traffic*, ICNP'99: Proceedings of the Seventh Annual International Conference on Network Protocols, IEEE Computer Society, Washington, DC (1999), 327.

[13] R.B. Cooper, *Stochastic Models*, Handbooks in Operations Research and Management Science, Vol. 2, Elsevier (1990), Chapter: Queueing Theory.

[14] M.E. Crovella and A. Bestavros, *Self-similarity in world wide web traffic: Evidence and possible causes*, IEEE/ACM Trans. Netw. **5** (6) (1997), 835–846.

[15] G.R. Ganger, B.L. Worthington, R.Y. Hou and Y.N. Patt, *Disk subsystem load balancing: Disk striping vs. conventional data placement*, Proceeding of the Twenty-Sixth Hawaii International Conference on System Sciences, Vol. 1 (1993), 40–49.
[16] G.R. Ganger, B.L. Worthington, R.Y. Hou and Y.N. Patt, *Disk arrays: High-performance, high-reliability storage subsystems*, Computer **27** (3) (1994), 30–36.
[17] K.I. Hopcroft, E. Jakeman and J.O. Matthews, *Discrete scale-free distributions and associated limit theorems*, J. Math. Phys. **A37** (2004), L635–L642.
[18] A. Høyland and M. Rausand, *System Reliability Theory: Models and Statistical Methods*, Wiley, New York (1994).
[19] R. Jain, *The Art of Computer Systems Performance Analysis*, Wiley Interscience, New York (1991).
[20] T. Karagiannis, M. Molle and M. Faloutsos, *Long-range dependence: Ten years of Internet traffic modeling*, IEEE Internet Computing **8** (5) (2004), 57–64.
[21] A.H. Karp and H.P. Flatt, *Measuring parallel processor performance*, Comm. ACM **33** (5) (1990), 539–543.
[22] W.E. Leland, M.S. Taqqu, W. Willinger and D.V. Wilson, *On the self-similar nature of ethernet traffic (extended version)*, IEEE/ACM Trans. Netw. **2** (1) (1994), 1–15.
[23] T. Limoncelli and C. Hogan, *The Practice of System and Network Administration*, Addison–Wesley (2003).
[24] R. Menuet and W.P. Turner, *Continuous cooling is required for continuous availability*, Technical report, Uptime Institute (2006).
[25] Office of Government Commerce, ed., *Best Practice for Service Delivery*, ITIL: The Key to Managing IT Services, The Stationary Office, London (2000).
[26] Office of Government Commerce, ed., *Best Practice for Service Support*, ITIL: The Key to Managing IT Services, The Stationary Office, London (2000).
[27] K. Park, G. Kim and M. Crovella, *On the relationship between file sizes, transport protocols, and self-similar network traffic*, ICNP'96: Proceedings of the 1996 International Conference on Network Protocols (ICNP'96), IEEE Computer Society, Washington, DC (1996), 171.
[28] V. Paxson and S. Floyd, *Wide area traffic: The failure of Poisson modeling*, IEEE/ACM Trans. on Netw. **3** (3) (1995), 226–244.
[29] K. Raatikainen, *Symptoms of self-similarity in measured arrival process of ethernet packets to a file server*, Preprint, University of Helsinki (1994).
[30] G.B. Rodosek, *Quality aspects in it service management*, IFIP/IEEE 13th International Workshop on Distributed Systems: Operations and Management (DSOM 2002) (2002), 82.
[31] H. Simitci and D.A. Reed, *Adaptive disk striping for parallel input/output*, 16th IEEE Symposium on Mass Storage Systems (1999), 88–102.
[32] L.P. Slothouber, *A model of web server performance*, The 5th International World Wide Web Conference, Paris, France (1996).
[33] W.P. Turner, J.H. Seader and K.G. Brill, *Tier classifications define siet infrastructure performance*, Technical report, Uptime Institute (1996), 2001–2006.
[34] R.D. van der Mei, R. Hariharan and P.K. Reeser, *Web server performance modeling*, Telecommunication Systems **16** (March 2001), 361–378.
[35] J. Walrand, *Stochastic Models*, Handbooks in Operations Research and Management Science, Vol. 2, Elsevier (1990), Chapter: Queueing Networks.

– 1.3 –

Automating System Administration: Landscape, Approaches and Costs

Aaron B. Brown[1], Joseph L. Hellerstein[2], Alexander Keller[3]

[1]*IBM Software Group, Route 100, Somers, NY 10589, USA*
E-mail: abbrown@us.ibm.com

[2]*Microsoft Corporation, One Microsoft Way, Redmond, WA 98052, USA*
E-mail: Joehe@microsoft.com

[3]*IBM Global Technology Services, 11 Madison Avenue, New York, NY 10010, USA*
E-mail: alexk@us.ibm.com

1. Introduction

The cost of systems administration in Information Technology (IT) systems often exceeds the cost of hardware and software. Our belief is that automating system administration can reduce these costs and increase the business value provided by IT.

Making progress with automating system administration requires addressing three questions. What to automate? How should this automation be done? When does the automation provide business value?

What to automate is often answered in a simplistic way – everything! The problem here is that automation requires investment and inevitably causes some disruption in the 'as-is' IT environment. As a result, it is important to target aspects of System Administration where automation provides the most value. We believe that a process-based perspective provides the kind of broad context in which such judgments can be made.

How to automate system administration can be approached in many ways. Among these are rule-based techniques, control theory and automated workflow construction. These approaches provide different kinds of benefits, and, in many ways, address different aspects of the automation challenge.

HANDBOOK OF NETWORK AND SYSTEM ADMINISTRATION
Edited by Jan Bergstra and Mark Burgess

When to automate is ultimately a business decision that should be based on a full understanding of the costs and benefits. The traditional perspective has been that automation is always advantageous. However, it is important to look at the full costs imposed by automation. For example, automating software distribution requires that: (a) the distribution infrastructure be installed and maintained; (b) software packages be prepared in the format required by the distribution infrastructure; and (c) additional tools be provided to handle problems with packages that are deployed because of the large scale of the impact of these problems. While automation often provides a net benefit despite these costs, we have observed cases in which these costs exceed the benefits.

There is a variety of existing literature on systems administration. One of the earliest efforts in automating systems administration is an expert systems for managing mainframe systems [39]. More recently, [9] addresses distributed resource administration using cfengine. Historically, authors have addressed various aspects of systems administration [21,23,24,47,49] and [36]. There has been theoretical work as well such as [46], who discusses how to structure actions for systems administration in configuration management, and [50], who addresses automated responses to anomalies using immunological algorithms.

The remainder of this chapter provides more detail on the questions of what, how and when to automate.

2. What system administrators do

We start our consideration of automating system administration by examining what system administrators do, as these tasks are the starting points for building automation.

System administrators are a central part of IT operations. Anyone who has owned a personal computer knows some elements of system administration. Software must be installed, patched and upgraded. Important data must be backed up, and occasionally restored. And, when a problem occurs, time must be spent on problem isolation, identification, and resolution.

In many ways, system administrators perform the same activities as individuals. There are, however, some important differences. The first is scale. Data centers consist of hundreds to tens of thousands of servers, and administrators are typically responsible for tens to hundreds of machines. This scale means that management tools are essential to deal with repetitive tasks such as software installation.

A second important difference is that system administrators are usually responsible for mission critical machines and applications. Seconds of downtime at a financial institution can mean millions of dollars of lost revenue. Failures in control systems for trains and aviation can cost lives. These considerations place tremendous emphasis on the speed and accuracy with which administrators perform their jobs.

Large data centers structure system administration by technology.

Examples of these technology areas are servers, databases, storage and networks. Administrators will typically train and develop specialized expertise in one technology area –

for example a database administrator (DBA) becomes an expert in the details of database installation, tuning, configuration, diagnosis and maintenance. However, the different technology areas often contain similar tasks, such as diagnosis and capacity planning, so the techniques developed in one area will often translate relatively easily to others. Even so, the tools used to accomplish these tasks in the separate areas remain different and specialized.

The remainder of this section is divided into four parts. The first two parts describe system administration in the words of experts. We focus on two technology areas – database administration and network administration. The third part of the section is a broader look at system administration based on a USENIX/SAGE survey. The section concludes with a discussion of best practices.

2.1. *Database administration*

We begin our look at expert system administration in the area of database administration. The material in this section is based on a database administration course [10].

The job of a database administrator (DBA) is largely driven by the needs of the organization. But at the heart of this job is a focus on managing data integrity, access, performance and security/privacy. Typical tasks performed by a DBA might include:

- designing and creating new databases;
- configuring existing databases to optimize performance, for example by manipulating index structures;
- managing database storage to ensure enough room for stored data;
- diagnosing problems such as 'stale' database data and deadlocks;
- ensuring data is replicated to on-line or off-line backup nodes;
- federating multiple independent database instances into a single virtual database;
- interfacing with other administrators to troubleshoot problems that extend beyond the database (e.g., storage or networking issues);
- generating reports based on database contents and rolling up those reports across multiple database systems.

The tools that a DBA uses to perform tasks like these vary greatly depending on the scale and needs of the organization. Small organizations and departments may use simple tools such as Paradox, Visual FoxPro, or Microsoft Access to maintain data. Larger organizations and governments with corporate-wide data requirements demand industrial-strength database tools such as IBM DB2, Oracle database, Microsoft SQL Server, Ingres, etc. And for organizations with multiple databases (often a situation that occurs after acquisitions, or when independent departments are transitioned to a centralized system), additional tools, often called 'middleware', will be necessary to federate and integrate the existing databases across the corporation. Middleware poses its own administration considerations for installation, configuration, optimization, and maintenance.

In addition to the core tools – the database engine(s) and middleware – DBAs use many specialized administration tools. These may be scripts developed by the DBA for specific situations, or they may be vendor-supplied tools for monitoring, administration and reporting.

As we look toward automation of a DBAs system administration responsibilities, we must recognize the challenges faced by a DBA. First is the integration of large numbers of tools. It should be clear from the discussion above that anything larger than a small department environment will involve multiple database engines, possibly middleware and a mix of home-grown and vendor-supplied tools. Second is the need to 'roll-up' data in large environments, e.g., integrating department reports into corporate-wide reports. Part of the challenge of the roll-up is the need for processes to scrub data to ensure correctness and consistency. The challenge of providing roll-ups creates a tension between the small scale systems with ease of entry and the large scale systems that provide robustness and scalability.

Finally, DBAs do not operate in a vacuum. There is considerable interaction with other administration 'towers', such as network and storage management. The following from [10] provides an illustrative example:

> Unfortunately, one situation that can occur more often than planned, or more accurately, more than it should, is when the database does not update despite the careful steps taken or the time involved in trying to accomplish a successful update [...]
>
> The first step is to investigate the problem to learn the true 'age' of the data. Next, the DBA would attempt to update a recent set of data to determine what may have occurred. It could have been a malfunction that day, or the evening the update was attempted. Assuming the situation does not improve by this upload, the next step is to contact the network administrator about possible server malfunctions, changes in standard record settings, or other changes that might affect the upload of data.
>
> Occasionally, the network server record lock settings were changed to 'disallow' any upload to a network server over a certain limit [...] If the DBA is lucky, or has positive karma due for collection, all that may be required is for the network administrator to reset the record lock setting at a higher level. Updates can then be repeated and will result in current data within the business unit database for purposes of management reporting.
>
> However, on a bad day, the DBA may learn from the network administrator that the server was or is malfunctioning, or worse, crashed at the precise time of the attempted update [...] In this situation the investigation must retrace the steps of data accumulation to determine the validity of the dataset for backup and experimental upload (network administrators at ringside) to watch for any type of malfunction.

In environments such as the foregoing, automation usually proceeds in an incremental manner, starting with special-purpose scripts that automate specific processes around the sets of existing tools, to generalizations of those scripts into new management tools, to rich automation frameworks that integrate tools to close the loop from detecting problems to reacting to them automatically.

2.2. *Network administration*

Next, we look at another technology area for system administration: network administration. The material in this section is based on a course on network administration [54]. We have included quotes as appropriate.

Network administrators are responsible for the performance, reliability, and scalability of corporate networks. Achieving these goals requires a substantial understanding of the business for which these services are being delivered as well as the nature and trends

in network technologies. In particular, in today's network-connected, web-facing world, network administrators have gone from supporting the back-office to enabling the online front-office, ensuring the network can deliver the performance and reliability to provide customer information, sales, support, and even B2B commerce online via the Internet. The network administrator's job has become a critical function in the revenue stream for many businesses.

Typical tasks performed by a network administrator might include:

- designing, deploying, and redesigning new networks and network interconnections;
- deploying new devices onto a network, which requires a good understanding of the network configuration, the device requirements, and considerations for loads imposed by network traffic;
- setting and enforcing security policies for network-connected elements;
- monitoring network performance and reconfiguring network elements to improve performance;
- managing network-sourced events;
- tracking the configuration of a network and the inventory of devices attached to it;
- detecting failures, diagnosing their root causes, and taking corrective actions such as reconfiguring the network around a failed component.

Network administrators make use of a variety of tools to perform these tasks. Some are as simple as the TCP `ping`, `traceroute` and `netstat` commands, which provide simple diagnostics. Administrators responsible for large corporate networks frequently make use of network management systems with sophisticated capabilities for filtering, eventing, and visualization. Examples of such systems are Hewlett–Packard's OpenView product and IBM's NetView product. Administrators of all networks will occasionally have to use low-level debugging tools such as packet sniffers and protocol decoders to solve compatibility and performance problems. No matter the scale of the network, tools are critical given the sizes of modern networks and the volumes of data that move across them. To quote from the advice for network administrators in [54], 'A thorough inventory and knowledge of the tools at your disposal will make the difference between being productive and wasting time. A good network administrator will constantly seek out ways to perform daily tasks in a more productive way'.

As we look toward automation of a network administrator's activities, there are two key aspects to consider. First is to provide more productive tools and to integrate existing tools. Here, automation must help administrators become more productive by absorbing the burden of monitoring, event management, and network performance optimization. Again this kind of automation progresses incrementally, starting with better scripts to integrate existing tools into situation-specific problem solvers, moving to integrated tools that, for example, merge monitoring with policy-driven response for specific situations, and culminating in integrated frameworks that close the loop and take over management of entire portions of a corporate network from the human administrator's hands.

The second key aspect of automating a network administrator's activities is to improve the task of problem diagnosis. Here, we refer to both proactive maintenance and reactive problem-solving. In the words of [54]:

> Resolving network, computer, and user related issues require the bulk of an administrator's time. Eventually this burden can be lessened through finding permanent resolutions to common prob-

> lems and by taking opportunities to educate the end user. Some reoccurring tasks may be automated to provide the administrator with more time [...] However, insuring the proper operation of the network will preempt many problems before the users notice them.

Finally, it is important to point out that for any system administration role, automation can only go so far. There always is a human aspect of system administration that automation will not replace; in the case of network administration, again in the words of [54], this comes in many forms.

> Apart from the software and hardware, often the most difficult challenge for a network administrator is juggling the human aspects of the job. The desired result is always a productive network user, not necessarily just a working network. To the extent possible, the administrator should attempt to address each user's individual needs and preferences. This also includes dealing with the issues that arise out of supporting user skill levels ranging from beginner to knowledgeable. People can be the hardest and yet most rewarding part of the job.
>
> As problems are addressed, the solutions will be documented and the users updated. Keeping an open dialogue between the administrator and users is critical to efficiently resolving issues.

2.3. *The broader view*

Now that we have looked at a few examples of system administration, next we move up from the details of individual perspectives to a survey of administrators conducted by SAGE, the Special Interest Group of the USENIX Association focusing on system administration. Other studies support these broad conclusions, such as [4].

The SAGE study [20] covered 128 respondents from 20 countries, with average experience of 7 years, 77% with college degrees, and another 20% having some college. The survey focused on server administration. Within this group, each administrator supported 500 users, 10–20 servers, and works in a group of 2 to 5 people. As reported in the survey, administrators have an 'atypical day' at least once a week.

With this background, we use the survey results to characterize how administrators spend their time, with an eye to assessing the opportunity for automation. This is done along two different dimensions. The first is *what* is being administered:

- 20% miscellaneous;
- 12% application software;
- 12% email;
- 9% operating systems;
- 7% hardware;
- 6% utilities;
- 5% user environment.

There is a large spread of administrative targets, and the reason for the large fraction of miscellaneous administration may well be due to the need to deal with several problems at once.

> Broad automation is needed to improve system administration.

Automation that targets a single domain such as hardware will be useful, but will not solve the end-to-end problems that system administrators typically face. Instead, we should

consider automation that takes an end-to-end approach, cutting across domains as needed to solve the kinds of problems that occur together. We will address this topic in Section 4 in our discussion of process-based automation.

Another way to view administrators' time is to divide it by the kind of activity. The survey results report:

- 11% meetings;
- 11% communicating;
- 9% configuring;
- 8% installing;
- 8% 'doing';
- 7% answering questions;
- 7% debugging.

We see that at least 29% of the administrator's job involves working with others. Sometimes, this is working with colleagues to solve a problem. But there is also substantial time in reporting to management on progress in resolving an urgent matter and dealing with dissatisfied users. About 32% of the administrator's job involves direct operations on the IT environment. This is the most obvious target for automation, although it is clear that automation will need to address some of the communication issues as well (e.g., via better reporting).

The survey goes on to report some other facts of interest. For example, fire fighting only takes 5% of the time. And, there is little time spent on scheduling, planning and designing. The latter, it is noted, may well be reflected in the time spent on meetings and communicating. We can learn from these facts that automation must address the entire lifecycle of administration, not just the (admittedly intense) periods of high-pressure problem-solving.

The high pressure and broad demands of system administration may raise serious questions about job satisfaction. Yet, 99% of those surveyed said they would do it again.

2.4. *The promise of best practices and automation*

One insight from the foregoing discussion is that today systems administration is more of a craft than a well-disciplined profession. Part of the reason for this is that rapid changes in IT make it difficult to have re-usable best practices, at least for many technical details.

There is a body of best practices for service delivery that is gaining increasing acceptance, especially in Europe. Referred to as the Information Technology Infrastructure Library (ITIL), these best practices encompass configuration, incident, problem management, change management, and many other processes that are central to systems administration [27]. Unfortunately, ITIL provides only high level guidance. For example, the ITIL process for change management has activities for 'authorizing change', 'assign priority' and 'schedule change'. There are few specifics about the criteria used for making decisions, the information needed to apply these criteria, or the tools required to collect this information.

> The key element of the ITIL perspective is its focus on *process*.

ITIL describes end-to-end activities that cut across the traditional system administra-

tion disciplines, and suggests how different 'towers' like network, storage, and application/database management come together to design infrastructure, optimize behavior, or solve cross-cutting problems faced by users. ITIL thus provides a philosophy that can guide us to the ultimate goals of automation, where end-to-end, closed-loop activities are subsumed entirely by automation, the system administrators can step out, and thus the costs of delivering IT services are reduced significantly.

We believe that installations will move through several phases in their quest to reduce the burden on systems administrators and hence the cost of delivering IT services. In our view, these steps are:

1. *Environment-independent automation*: execution of repetitive tasks without system-dependent data within one tower, for example static scripts or response-file-driven installations.
2. *Environment-dependent automation*: taking actions based on configuration, events, and other factors that require collecting data from systems. This level of automation is often called 'closed-loop' automation, though here it is still restricted to one discipline or tower.
3. *Process-based automation*: automation of interrelated activities in best practices for IT service delivery. Initially, this tends to be open-loop automation that mimics activities done by administrators such as the steps taken in a software install. Later on, automation is extended to closed-loop control of systems such as problem detection and resolution.
4. *Business level automation*: automation of IT systems based on business policies, priorities, and processes. This is an extension of process-based automation where the automation is aware of the business-level impact of various IT activities and how those IT activities fit into higher-level business processes (for example, insurance claim processing). It extends the closed-loop automation described in (3) to incorporate business insights.

Business-level automation is a lofty goal, but the state of the art in 2006 is still far from reaching it outside very narrowly-constrained domains. And, at the other extreme, environment-independent automation is already well-established through ad-hoc mechanisms like scripts and response files. Thus in the remainder of our discussion we will focus on how to achieve environment-dependent and process-based automation. Section 3 describes a strategy for reaching process-based automation, and follows that with a description of some automation techniques that allow for environment-dependent automation and provide a stepping stone to higher levels of automation.

3. How to automate

This section addresses the question of how to automate system administration tasks.

3.1. *Automation strategies*

We start our discussion of automating IT service delivery by outlining a best-practice approach to process-based automation (cf. [7]). We have developed this approach based on

our experiences with automating change management, along with insight we have distilled from interactions and engagements with service delivery personnel. It provides a roadmap for achieving process-level automation.

Our approach comprises six steps for transforming existing IT service delivery processes with automation or for introducing new automated process into an IT environment. The first step is to

(1) identify best practice processes for the domain to be automated.

To identify these best practices we turn to the IT Infrastructure Library (ITIL), a widely used process-based approach to IT service management. ITIL comprises several disciplines such as infrastructure management, application management, service support and delivery. The ITIL Service Support discipline [27] defines the Service Desk as the focal point for interactions between a service provider and its customers. To be effective, the Service Desk needs to be closely tied into roughly a dozen IT support processes that address the lifecycle of a service. Some examples of IT service support processes for which ITIL provides best practices are: Configuration Management, Change Management, Release Management, Incident Management, Problem Management, Service Level Management and Capacity Management.

ITIL provides a set of process domains and best practices within each domain, but it does not provide guidance as to which domain should be an organization's initial target for automation. Typically this choice will be governed by the cost of existing activities.

After identifying the best practices, the next step is to

(2) establish the scope of applicability for the automation.

Most ITIL best practices cover a broad range of activities. As such, they are difficult to automate completely. To bound the scope of automation, it is best to target a particular subdomain (e.g., technology area). For example, Change Management applies changes in database systems, storage systems and operating systems.

We expect that Change Management will be an early focus of automation. It is our further expectation that initial efforts to automate Change Management will concentrate on high frequency requests, such as security patches. Success here will provide a proof point and template for automating other areas of Change Management. And, once Change Management is automated to an acceptable degree, then other best practices can be automated as well, such as Configuration Management. As with Change Management, we expect that the automation of Configuration Management will be approached incrementally. This might be structured in terms of configuration of servers, software licenses, documents, networks, storage and various other components.

The next step in our roadmap follows the ITIL-based approach:

(3) identify delegation opportunities.

Each ITIL best practice defines (explicitly or implicitly) a process flow consisting of multiple activities linked in a workflow. Some of these activities will be amenable to automation, such as deploying a change in Change Management. These activities can be *delegated* to an automated system or tool. Other activities will not be easy to automate, such as obtaining change approvals from Change Advisory Boards. Thus, an analysis is needed to determine what can be automated and at what cost. Sometimes, automation drives changes in IT processes. For example, if an automated Change Management system is trusted enough, change approvals might be handled by the Change Management System automatically.

The benefit of explicitly considering delegation is that it brings the rigor of the best-practice process framework to the task of scoping the automation effort. The best practice defines the set of needed functionality, and the delegation analysis explicitly surfaces the decision of whether each piece of functionality is better handled manually or by automation. Using this framework helps prevent situations like the one discussed later in Section 4 of this chapter, where the cost of an automation process outweighs its benefits in certain situations.

With the delegated activities identified, the fourth step in our automation approach is to

(4) identify links between delegated activities and external activities, processes and data sources.

These links define the control and data interfaces to the automation. They may also induce new requirements on data types/formats and APIs for external tools. An example in Change Management is the use of configuration data. If Change Management is automated but Configuration Management remains manual, the Configuration Management Database (CMDB) may need to be enhanced with additional APIs to allow for programmatic access from the automated Change Management activities. Moreover, automation often creates *induced processes*, additional activities that are included in support of the automation. Examples of induced processes in automated software distribution are the processes for maintaining the software distribution infrastructure, error recovery and preparation of the software packages.

The latter point motivates the following step:

(5) Identify, design, and document induced processes needed to interface with or maintain the automation.

This step surfaces many implications and costs of automation and provides a way to do cost/benefit tradeoffs for proposed automation.

The last step in our automation approach is to

(6) implement automation for the process flow and the delegated activities.

Implementing the process flow is best done using a workflow system to automatically coordinate the best-practice process' activities and the flow of information between them. Using a workflow system brings the additional advantage that it can easily integrate automated and manual activities.

For the delegated activities, additional automation implementation considerations are required. This is a non-trivial task that draws on the information gleaned in the earlier steps. It uses the best practice identified in (1) and the scoping in (2) to define the activities' functionality. The delegation choices in (3) scope the implementation work, while the interfaces and links to induced process identified in (4) and (5) define needed APIs, connections with external tools and data sources, and user interfaces. Finally, the actual work of the activity needs to be implemented directly, either in new code or by using existing tools.

In some cases, step (6) may also involve a recursive application of the entire methodology described here. This is typically the case when a delegated ITIL activity involves a complex flow of work that amounts to a process in its own right. In these cases, that activity sub-process may need to be decomposed into a set of subactivities; scoped as in steps (2) and (3), linked with external entities as in (4), and may induce extra sub-process as in (5). The sub-process may in turn also need to be implemented in a workflow engine, albeit at a lower level than the top-level best practice process flow.

3.2. *Automation technologies*

This section introduces the most common technologies for automation. Considered here are rules, feedback control techniques, and workflows technology.

3.2.1. *Rule-based techniques*

Rules provide a condition–action representation for automation.

An example of a rule is

- Rule: If there is a `SlowResponse` event from system ?S1 at location ?L1 within 1 minute of another `SlowResponse` event from system ?S2 $\neq$?S1 at location ?L1 and there is no `SlowResponse` event from system S3 at location ?L2 $\neq$?L1, then alert the Network Manager for location ?L1.

In general, rules are if-then statements. The if-portion, or left-hand side, describes a situation. The then-portion, or right-hand side, specifies actions to take when the situation arises.

As noted in [52], there is a great deal of work in using rule-based techniques for root cause analysis (RCA). [39] describes a rule-based expert system for automating the operation of an IBM mainframe, including diagnosing various kinds of problems. [22] describes algorithms for identifying causes of event storms. [28] describe the application of an expert system shell for telecommunication network alarm correlation.

The simplicity of rule-based systems offers an excellent starting point for automation. However, there are many shortcomings. Among these are:

1. A pure rule-based system requires detailed descriptions of many situations. These descriptions are time-consuming to construct and expensive to maintain.
2. Rule-based systems scale poorly because of potentially complex interactions between rules. Such interactions make it difficult to debug large scale rule-based systems, and create great challenges when adding new capabilities to such systems.
3. Not all automation is easily represented as rules. Indeed, step-by-step procedures are more naturally expressed as workflows.

There are many approaches to circumventing these shortcomings. One of the most prominent examples is [32], which describes a code-book-based algorithm for RCA. The practical application of this approach relies on explicit representations of device behaviors and the use of configuration information. This has been quite effective for certain classes of problems.

3.2.2. *Control theoretic approaches*

Control theory provides a formal framework for optimizing systems.

Over the last sixty years, control theory has developed a fairly simple reference architecture. This architecture is about manipulating a target system to achieve a desired objective. The component that manipulates the target system is the controller.

As discussed in [15] and depicted in Figure 1, the essential elements of feedback control system are:

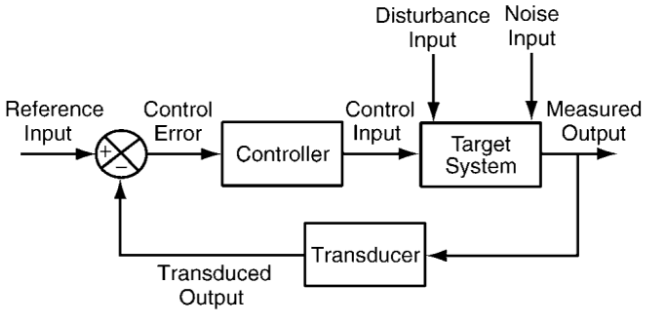


Fig. 1. Block diagram of a feedback control system. The reference input is the desired value of the system's measured output. The controller adjusts the setting of control input to the target system so that its measured output is equal to the reference input. The transducer represents effects such as units conversions and delays.

- target system, which is the computing system to be controlled;
- control input, which is a parameter that affects the behavior of the target system and can be adjusted dynamically (such as the `MaxClients` parameter in the Apache HTTP Server);
- measured output, which is a measurable characteristic of the target system such as CPU utilization and response time;
- disturbance input, which is any change that affects the way in which the control input influences the measured output of the target system (e.g., running a virus scan or a backup);
- noise input, which is any effect that changes the measured output produced by the target system. This is also called sensor noise or measurement noise;
- reference input, which is the desired value of the measured output (or transformations of them), such as CPU utilization should be 66%. Sometimes, the reference input is referred to as desired output or the setpoint;
- transducer, which transforms the measured output so that it can be compared with the reference input (e.g., smoothing stochastics of the output);
- control error, which is the difference between the reference input and the measured output (which may include noise and/or may pass through a transducer);
- controller, which determines the setting of the control input needed to achieve the reference input. The controller computes values of the control input based on current and past values of control error.

Reference inputs specify policies.

For example, a service level agreement may specify constraints on response times for classes of services such as "requests to browse the product catalogue should be satisfied within two seconds". In this case, the measured output is response time for a browse request, and the reference input is two seconds. More details can be found in [8].

The foregoing is best understood in the context of a specific system. Consider a cluster of three Apache Web Servers. The Administrator may want these systems to run at no greater than 66% utilization so that if any one of them fails, the other two can immediately absorb

the entire load. Here, the measured output is CPU utilization. The control input is the maximum number of connections that the server permits as specified by the `MaxClients` parameter. This parameter can be manipulated to adjust CPU utilization. Examples of disturbances are changes in arrival rates and shifts in the type of requests (e.g., from static to dynamic pages).

To illustrate how control theory can be applied to computing systems, consider the IBM Lotus Domino Serveras described in [41]. To ensure efficient and reliable operation, Administrators of this system often regulate the number of remote procedure calls (RPCs) in the server, a quantity that we denote by *RIS*. *RIS* roughly corresponds to the number of *active users* (those with requests outstanding at the server). Regulation is accomplished by using the `MaxUsers` tuning parameter that controls the number of *connected users*. The correspondence between `MaxUsers` and *RIS* changes over time, which means that `MaxUsers` must be updated almost continuously to achieve the control objective. Clearly, it is desirable to have a controller that automatically determines the value of `MaxUsers` based on the objective for *RIS*.

Our starting point is to model how `MaxUsers` affects *RIS*. The input to this model is `MaxUsers`, and the output is *RIS*. We use $u(k)$ to denote the kth value of the former and $y(k)$ to denote the kth value of the latter. (Actually, $u(k)$ and $y(k)$ are offsets from a desired operating point.) A standard workload was applied to a IBM Lotus Domino Server running product level software in order to obtain training and test data. In all cases, values are averaged over a one minute interval. Based on these experiments, we constructed an empirical model (using least squares regression) that relates `MaxUsers` to *RIS*:

$$y(k+1) = 0.43y(k) + 0.47u(k). \tag{1}$$

To better facilitate control analysis, Equation (1) is put into the form of a transfer function, which is a Z-transform representation of how `MaxUsers` affects *RIS*. Z-transforms provide a compact representation for time varying functions, where z represents a time shift operation. The transfer function of Equation (1) is

$$\frac{0.47}{z - 0.43}.$$

The poles of a transfer function are the values of z for which the denominator is 0. It turns out that the poles determine the stability of the system represented by the transfer function, and they largely determine its settling time. This can be seen in Equation (1). Here, there is one pole, which is 0.43. The effect of this pole on settling time is clear if we solve the recurrence in Equation (1). The result has the factors $0.43^{k+1}, 0.43^k, \ldots$. Thus, if the absolute value of the pole is greater than one, the system is unstable. And the closer the pole is to 0, the shorter the settling time. A pole that is negative (or imaginary) indicates an oscillatory response.

Many researchers have applied control theory to computing systems. In data networks, there has been considerable interest in applying control theory to problems of flow control, such as [31] which develops the concept of a Rate Allocating Server that regulates the flow of packets through queues. Others have applied control theory to short-term rate variations

in TCP (e.g., [35]) and some have considered stochastic control [2]. More recently, there have been detailed models of TCP developed in continuous time (using fluid flow approximations) that have produced interesting insights into the operation of buffer management schemes in routers (see [25,26]). Control theory has also been applied to middleware to provide service differentiation and regulation of resource utilizations as well as optimization of service level objectives. Examples of service differentiation include enforcing relative delays [1], preferential caching of data [37], and limiting the impact of administrative utilities on production work [40]. Examples of regulating resource utilizations include a mixture of queueing and control theory used to regulate the Apache HTTP Server [48], regulation of the IBM Lotus Domino Server [41], and multiple-input, multiple-output control of the Apache HTTP Server (e.g., simultaneous regulation of CPU and memory resources) [14]. Examples of optimizing service level objectives include minimizing response times of the Apache Web Server [16] and balancing the load to optimize database memory management [17].

All of these examples illustrate situations where control theory-based automation was able to replace or augment manual system administration activities, particularly in the performance and resource management domains.

3.2.3. *Automated workflow construction* In Section 2.4, we have observed that each ITIL best practice defines (explicitly or implicitly) a process flow consisting of multiple activities linked in a workflow. Some of these activities will be amenable to automation – such as deploying a change in Change Management – whereas others will not (such as obtaining change approvals from Change Advisory Boards). On a technical level, recent efforts aim at introducing extensions for people facing activities into workflow languages [33]. The goal is to facilitate the seamless interaction between the automated activities of an IT service management process and the ones that are carried out by humans. Here, we summarize work in [5] and [29] on automating system administration using workflow.

In order to provide a foundation for process-based automation, it is important to identify the control and data interfaces between delegated activities and external activities, processes, and data sources. The automation is provided by lower-level automation workflows, which consist of atomic administration activities, such as installing a database management system, or configuring a data source in a Web Application Server. The key question is to what extent the automation workflows can be automatically generated from domain-specific knowledge, instead of being manually created and maintained. For example, many automation workflows consist of activities whose execution depends on the current state of a distributed system. These activities are often not specified in advance. Rather, they are merely implied. For example, applications must be recompiled if they use a database table whose schema is to change. Such implicit changes are a result of various kinds of relationships, such as service dependencies and the sharing of the service provider's resources among different customers. Dependencies express compatibility requirements between the various components of which a distributed system is composed. Such requirements typically comprise software pre-requisites (components that must be present on the system for an installation to succeed), co-requisites (components that must be jointly installed) as well as ex-requisites (components that must be removed prior to installing a new component). In addition, version compatibility constraints and memory/disk

space requirements need to be taken into account. All of this information is typically captured during the development and build stages of components, either by the developer, or by appropriate tooling. Dependency models, which can be specified e.g., as *Installable Unit Deployment Descriptors* [53] or *System Description Models* [38], are a key mechanism to capture this knowledge and make it available to the tools that manage the lifecycle of a distributed system.

An example of a consumer of dependency information is the *Change Management System*, whose task is to orchestrate and coordinate the deployment, installation and configuration of a distributed system. Upon receiving a request for change (RFC) from an administrator, the change management system generates a *Change Plan*. A change plan describes the partial order in which tasks need to be carried out to transition a system from a workable state into another workable state. In order to achieve this, it contains information about:

- The type of change to be carried out, e.g., install, update, configure, uninstall;
- the roles and names of the components that are subject to a change (either directly specified in the RFC, or determined by the change management system);
- the precedence and location constraints that may exist between tasks, based on component dependency information;
- an estimate of how long every task is likely to take, based on the results of previous deployments. This is needed to estimate the impact of a change in terms of downtime and monetary losses.

The change management system exploits dependency information. The dependency information is used to determine whether tasks required for a change must be carried out sequentially, or whether some of them can be parallelized. The existence of a dependency between two components – each representing a managed resource – in a dependency graph indicates that a precedence/location constraint must be addressed. If a precedence constraint exists between two tasks in a workflow (e.g., X must be installed before Y), they need to be carried out sequentially. This is typically indicated by the presence of a link; any task may have zero or more incoming and outgoing links. If two tasks share the same predecessor and no precedence constraints exist between them, they can be executed concurrently. Typically, tasks and their constraints are grouped on a per-host basis. Grouping on a per-host basis is important because the actual deployment could happen either push-based (triggered by the provisioning system) or pull-based (deployment is initiated by the target systems). An additional advantage of grouping activities on a per-host basis is that one can carry out changes in parallel (by observing the presence of cross-system links) if they happen on different systems. Note that parallelism on a single host system is difficult to exploit with current operating systems as few multithreaded installers exist today. In addition, some operating systems require exclusive access to shared libraries during installation.

Different types of changes require different traversals through the dependency models: If a request for a new installation of a component is received, one needs to determine which components must already be present before a new component can be installed. On the other hand, a request for an update, or an uninstall of an component leads to a query to determine the components that will be impacted by the change.

Precedence constraints represent the order in which provisioning activities need to be carried out. Some of these constraints are implicit (e.g., by means of a containment hier-

archy expressed as 'HasComponent' relationships between components), whereas others (typically resulting from communication dependencies such as 'uses') require an explicit representation (e.g., the fact that a database client needs to be present on a system whenever a database server located on a remote host needs to be accessed).

3.2.3.1. *Example for software Change Management* The example is based on the scenario of installing and configuring a multi-machine deployment of a J2EE based enterprise application and its supporting middleware software (including IBM's HTTP Server, WebSphere Application Server, WebSphere MQ embedded messaging, DB2 UDB database and DB2 runtime client). The specific application we use is taken from the SPECjAppServer2004 enterprise application performance benchmark [51]. It is a complex, multi-tiered on-line e-Commerce application that emulates an automobile manufacturing company and its associated dealerships. SPECjAppServer2004 comprises typical manufacturing, supply chain and inventory applications that are implemented with web, EJB, messaging, and database tiers. We jointly refer to the SPECjAppServer2004 enterprise application, its data, and the underlying middleware as the SPECjAppServer2004 *solution*. The SPECjAppServer2004 solution spans an environment consisting of two systems: one system hosts the application server along with the SPECjAppServer2004 J2EE application, whereas the second system runs the DBMS that hosts the various types of SPECjAppServer2004 data (catalog, orders, pricing, user data, etc.). One of the many challenges in provisioning such a solution consists in determining the proper order in which its components need to be deployed, installed, started and configured. For example, 'HostedBy' dependencies between the components 'SPECjAppServer2004 J2EE Application' and 'WebSphere Application Server v5.1' (WAS) state that the latter acts as a hosting environment for the former. This indicates that all the WAS components need to be operating before one can deploy the J2EE application into them.

A provisioning system supports the administrator in deploying, installing and configuring systems and applications. As mentioned above, a change plan is a procedural description of activities, each of which maps to an operation that the provisioning system exposes, preferably by means of a set of interfaces specified using the Web Services Description Language (WSDL). As these interfaces are known well in advance before a change plan is created, they can be referenced by a change plan. Every operation has a set of input parameters, for example, an operation to install a given software component on a target system requires references to the software and to the target system as input parameters. Some of the parameters may be input by a user when the change plan is executed (e.g., the hostname of the target system that will become a database server) and need to be forwarded to several activities in the change plan, whereas others are produced by prior activities.

3.2.3.2. *Executing the generated change plan* Upon receiving a newly submitted change request, the change management system needs to determine on which resources and at what time the change will be carried out. The change management system first inspects the resource pools of the provisioning system to determine which target systems are best assigned to the change by taking into account which operating system they run, what their system architecture is, and what the cost of assigning them to a change request is. Based on this information, the change management system creates a change plan, which may be

composed of already existing change plans that reside in a change plan repository. Once the change plan is created, it is submitted to the workflow engine of the provisioning system. The provisioning system maps the actions defined in the change plan to operations that are understood by the target systems. Its object-oriented data model is a hierarchy of *logical devices* that correspond to the various types of managed resources (e.g., software, storage, servers, clusters, routers or switches). The methods of these types correspond to *Logical Device Operations (LDOs)* that are exposed as WSDL interfaces, which allows their inclusion in the change plan. *Automation packages* are product-specific implementations of logical devices: e.g., an automation package for the DB2 DBMS would provide scripts that implement the software.install, software.start, software.stop, etc. LDOs. An automation package consists of a set of Jython scripts, each of which implements an LDO. Every script can further embed a combination of PERL, Expect, Windows scripting host or bash shell scripts that are executed on the remote target systems. We note that the composition pattern applies not only to workflows, but occurs in various places within the provisioning system itself to decouple a change plan from the specifics of the target systems.

The workflow engine inputs the change plan and starts each provisioning operation by directly invoking the LDOs of the provisioning system. These invocations are performed either in parallel or sequentially, according to the instructions in the change plan. A major advantage of using an embedded workflow engine is the fact that it automatically performs state-checking, i.e., it determines whether all conditions are met to move from one activity in a workflow to the next. Consequently, there is no need for additional program logic in the change plan to perform such checks. Status information is used by the workflow engine to check if the workflow constraints defined in the plan (such as deadlines) are met and to inform the change management system whether the roll-out of changes runs according to the schedule defined in the change plan.

3.2.3.3. *Recursive application of the automation pattern* The automation described in this section essentially makes use of three levels of workflow engines: (1) top-level coordination of the Change Management workflow; (2) implementation of the generated change plan; and (3) the provisioning system where the LDOs are implemented by miniature-workflows within their defined scripts. This use of multiple levels of workflow engine illustrates a particular pattern of composition that we expect to be common in automation of best-practice IT service management processes, and recalls the discussion earlier in Section 3.1 of recursive application of the automation approach.

In particular, the delegated Change Management activity of Distribute and Install Changes involves a complex flow of work in its own right – documented in the change plan. We can see that the approach to automating the construction of the change plan follows the same pattern we used to automate change management itself, albeit at a lower level. For example, the creation of the change workflow is a lower-level analogue to using ITIL best practices to identify the Change Management process activities. The execution of change plan tasks by the provisioning system represents delegation of those tasks to that provisioning system. The provisioning system uses external interfaces and structured inputs and APIs to automate those tasks – drawing on information from the CMDB to determine available resources and invoking lower-level operations (automation packages) to effect changes in the actual IT environment. In this latter step, we again see the need to

provide such resource information and control APIs in the structured, machine-readable formats needed to enable automation. The entire pattern repeats again at a lower level within the provisioning system itself, where the automation packages for detailed software and hardware products represent best practice operations with delegated functionality and external interfaces for data and control.

One of the key benefits of this type of recursive composition of our automation approach is that it generates reusable automation assets. Namely, at each level of automation, a set of automated delegated activities is created: automated ITIL activities at the top (such as Assess Change), automated change management activities in the middle (such as Install the DB2 Database), and automated software lifecycle activities at the bottom (such as Start DB2 Control Process). While created in the context of change management, it is possible that many of these activities (particularly lower-level ones) could be reused in other automation contexts. For example, many of the same lower-level activities created here could be used for performance management in an on-demand environment to enable creating, activating, and deactivating additional database or middleware instances. It is our hope that application of this automation pattern at multiple levels will reduce the long-term costs of creating system management automation, as repeated application will build up a library of reusable automation components that can be composed together to simplify future automation efforts.

3.2.3.4. *Challenges in workflow construction* An important question addresses the problem whether the change management system or the provisioning system should perform error recovery for the complete change plan in case an activity fails during workflow execution or runs behind schedule. One possible strategy is not to perform error recovery or dealing with schedule overruns within the change plan itself and instead delegate this decision instead to the change management system. The reason for doing so is that in service provider environments, resources are often shared among different customers, and a change of a customer's hosted application may affect the quality of service another customer receives. Before rolling out a change for a given customer, a service provider needs to trade off the additional benefit he receives from this customer against a potential monetary loss due to the fact that an SLA with another customer may be violated because of the change. The scheduling of change plans, a core change management system function, is the result of solving an optimization problem that carefully balances the benefits it receives from servicing one customer's change request against the losses it incurs from not being able to service other customers. In an on-demand environment, the cost/profit situation may change very rapidly as many change plans are concurrently executed at any given point in time. In some cases, it may be more advantageous to carry on with a change despite its delay, whereas in other cases, aborting a change and instead servicing another newly submitted request that is more profitable may lead to a better global optimum. This big picture, however, is only available to the change management system.

Another important requirement for provisioning composed applications is the dynamic aggregation of already existing and tested change plans. For example, in the case of SPECjAppServer2004 and its underlying middleware, change plans for provisioning some of its components (such as WebSphere Application Server or the DB2 DBMS) may already exist. Those workflows need to be retrieved from a workflow repository and aggregated in a

new workflow for which the activities for the remaining components have been generated. The challenge consists in automatically annotating the generated workflows with metadata so that they can be uniquely identified and evaluated with respect to their suitability for the specific request for change. The design of efficient query and retrieval mechanisms for workflows is an important prerequisite for the reuse of change plans.

4. When to automate

Now that we have identified the opportunities for automation and discussed various approaches, we now step back and ask: when is it appropriate to deploy systems management automation? The answer is, surprisingly, not as straightforward as one might expect.

Automation does not always reduce the cost of operations.

This is not a new conclusion, though it is rarely discussed in the context of automating system administration. Indeed, in other fields practitioners have long recognized that automation can be a double-edged sword. For example, early work with aircraft autopilots illustrated the dangers of imperfect automation, where pilots would lose situational awareness and fail to respond correctly when the primitive autopilot reached the edge of its envelope and disengaged [44], causing more dangerous situations than when the autopilot was not present. There are many other well-documented cases where either failures or unexpected behavior of industrial automation caused significant problems, including in such major incidents as Three Mile Island [43,45].

How are we to avoid, or at least minimize, these problems in automation of system administration? One solution is to make sure that the full consequences of automation are understood before deploying it, changing the decision of 'when to automate?' from a simple answer of 'always!' to a more considered analytic process.

In the next several subsections, we use techniques developed in [6] to consider what this analysis might look like, not to dissuade practitioners from designing and deploying automation, but to provide the full context needed to ensure that automation lives up to its promises.

4.1. *Cost-benefit analysis of automation*

We begin by exploring the potential hidden costs of automation. While automation obviously can reduce cost by removing the need for manual systems management, it can also induce additional costs that may offset or even negate the savings that arise from the transfer of manual work to automated mechanisms. If we take a process-based view, as we have throughout this paper, we see that automation transforms a manual process, largely to reduce manual work, but also to introduce new process steps and roles needed to maintain, monitor, and augment the automation itself.

Figure 2 illustrates the impact of automation on an example system management process, drawn from a real production data center environment. The process here is software distribution to server machines, a critical part of operating a data center. Software

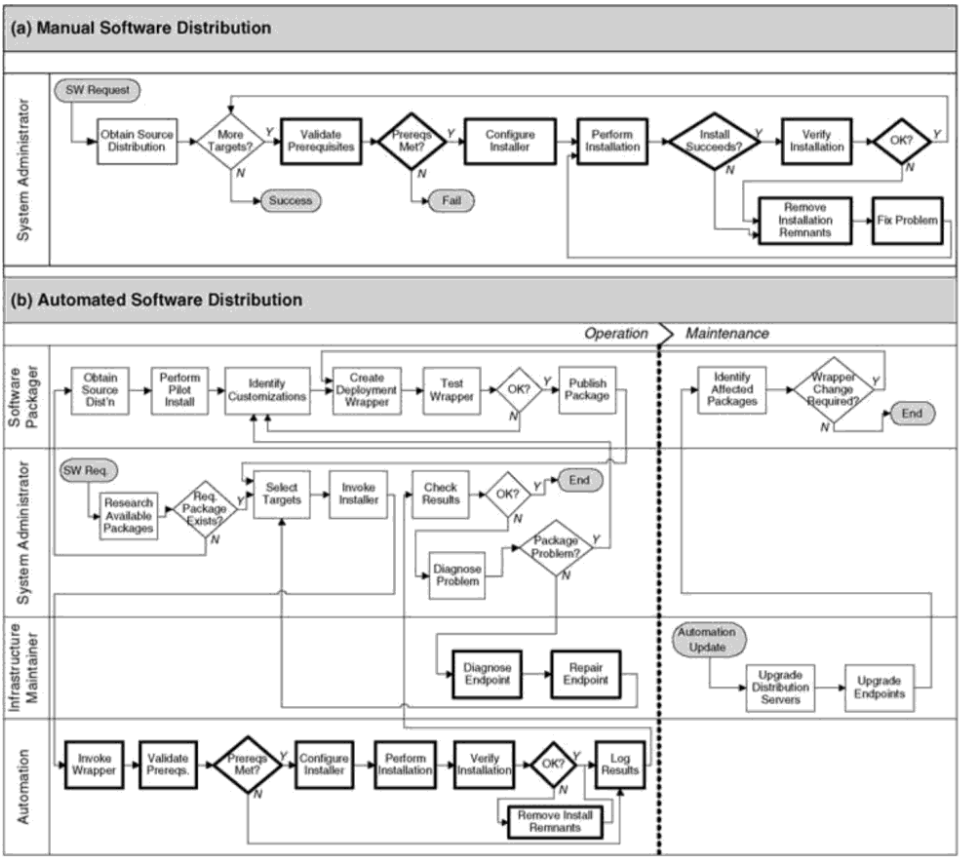


Fig. 2. Manual and automated processes for software distribution. Boxes with heavy lines indicate process steps that contribute to variable (per-target) costs, as described in Section 4.2.

distribution involves the selection of software components and their installation on target machines. We use the term 'package' to refer to the collection of software resources to install and the step-by-step procedure (process) by which this is done. We represent the process via 'swim-lane' diagrams – annotated flowcharts that allocate process activities across *roles* (represented as rows) and *phases* (represented as columns). Roles are typically performed by people (and can be shared or consolidated); we include automation as its own role to reflect activities that have been handed over to an automated system.

Figure 2(a) shows the 'swim-lane' representation for the manual version of our example software distribution process, and Figure 2(b) shows the same process once automated software distribution has been introduced.

Notice that, while key parts of the process have moved from a manual role to an automated role, there are additional implications. For one thing, the automation infrastructure is another software system that must itself be installed and maintained, creating initial transition costs as well as periodic costs for update and maintenance. (For simplicity, in the

figure we have assumed that the automation infrastructure has already been installed, but we do consider the need for periodic updates and maintenance.) Next, using the automated infrastructure requires that information be provided in a structured form. We use the term *software package* to refer to these structured inputs. These inputs are typically expressed in a formal structure, which means that their creation requires extra effort for package design, implementation, and testing. Last, when errors occur in the automated case, they happen on a much larger scale than for a manual approach, and hence additional processes and tools are required to recover from them. These other impacts manifest as additional process changes, namely extra roles and extra operational processes to handle the additional tasks and activities induced by the automation.

We have to recognize as well that the effects of induced processes – for error recovery, maintenance, and input creation/structuring – are potentially mitigated by the probability and frequency of their execution. For example, if automation failures are extremely rare, then having complex and costly recovery procedures may not be a problem. Furthermore, the entire automation framework has a certain lifetime dictated by its flexibility and generalization capability. If this lifetime is long, the costs to create and transition to the automated infrastructure may not be an issue. But even taking this into account, it is apparent from inspection that the collection of processes in Figure 2(b) is much more complicated than the single process in Figure 2(a). Clearly, such additional complexity is unjustified if we are installing a single package on a single server. This raises the following question – at what point does automation stop adding cost and instead start reducing cost?

The answer comes through a cost-benefit analysis, where the costs of induced process are weighted by their frequency and balanced against the benefits of automation. Our argument in this section is that such analyzes are an essential part of a considered automation decision. That is, when considering a manual task for automation, the benefits of automating that task (reduced manual effort, skill level, and error probability) need to be weighed against the costs identified above (extra work from induced process, new roles and skills needed to maintain the automation, and potential consequences from automation failure).

In particular, this analysis needs to consider both the changes in tasks (tasks replaced by automation and tasks induced by it) and the changes in roles and required human skills. The latter consideration is important because automation tends to create induced tasks requiring more skill than the tasks it eliminates. For example, the tasks induced by automation failures tend to require considerable problem-solving ability as well as a deep understanding of the architecture and operation of the automation technology. If an IT installation deploys automation without having an experienced administrator who can perform these failure-recovery tasks, the installation is at risk if a failure occurs. The fact that automation often increases the skill level required for operation is an example of an *irony of automation* [3]. The irony is that while automation is intended to reduce costs overall, there are some kinds of cost that increase. Many ironies of automation have been identified in industrial contexts such as plant automation and infrastructure monitoring [45]. Clearly, the ironies of automation must be taken into account when doing a cost-benefit analysis.

4.2. *Example analysis: software distribution for a datacenter*

To illustrate the issues associated with automation, we return to the example of software distribution. To perform our cost-benefit analysis, we start by identifying the fixed and variable costs for the process activities in Figure 2. The activities represented as boxes with heavy outlines represent variable-cost activities, as they are performed once for each machine in the data center. The other activities are fixed-cost in that they are performed once per software package.

A key concept used in our analysis is that of the *lifetime of a software package*. By the lifetime of a package, we mean the time from when the package is created to when it is retired or made obsolete by a new package. We measure this in terms of the number of target machines to which the package is distributed.

For the benefits of automation to outweigh the cost of automation, the variable costs of automation must be lower than the variable costs of the manual process, *and* the fixed cost of building a package must be amortized across the number of targets to which it is distributed over its lifetime. Using data from a several computer installation, Figure 3 plots the cumulative fraction of packages in terms of the lifetimes (in units of number of targets to which the package is distributed). We see that a large fraction of the packages are distributed to a small number of targets, with 25% of the packages going to fewer than 15 targets over their lifetimes.

Next, we look at the possibility of automation failure. By considering the complete view of the automated processes in Figure 2(b), we see that more sophistication and people are required to address error recovery for automated software distribution than for the manual process. Using the same data from which Figure 3 is extracted, we determined that 19% of the requested installs result in failure. Furthermore, at least 7% of the installs fail due to issues related to configuration of the automation infrastructure, a consideration that does not exist if a manual process is used. This back-of-the envelope analysis underscores the

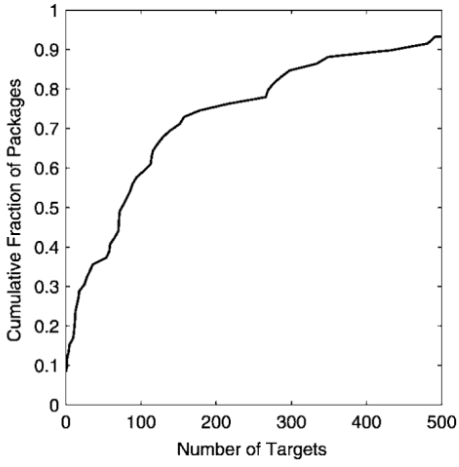


Fig. 3. Cumulative distribution of the number of targets (servers) on which a software package is installed over its lifetime in several data centers. A larger number of packages are installed on only a small number of targets.

importance of considering the entire set of process changes that occur when automation is deployed, particularly the extra operational processes created to handle automation failures. Drawing on additional data from IBM internal studies of software distribution and netting out the analysis (the details can be found in [6]), we find that for complex software packages, there should be approximately 5 to 20 targets for automated software distribution to be cost effective. In terms of the data in Figure 3, these numbers mean that from 15% to 30% of the installs in the data centers we examined should not have been automated from a cost perspective.

Automation can be a double-edged sword.

While in these datacenters automation of software distribution provided a net cost benefit for 70–85% of installs, it increased costs for the remaining 15–30%. In this case the choice was made to deploy the automation, with the assessment that the benefit to the large fraction of installs outweighed the cost to the smaller fraction. The insight of a cost-benefit analysis allows such decisions to be made with eyes open, and full awareness of the tradeoffs being made.

We note in passing that cost models for system administration have been developed based on other frameworks. For example, [12] models direct costs (e.g., salaries of system administrators) and indirect costs (e.g., diminished productivity due to downtime). Clearly, different cost models can yield different results in terms of when to employ automation.

4.3. *Additional concerns: adoption and trust*

Automation is only useful if it is used. Even the best automation – carefully designed to maximize benefit and minimize induced process and cost – can lack traction in the field. Often these situations occur when the automation fails to gain the trust of the system management staff tasked with deploying and maintaining it.

So what can automation designers do to help their automation gain trust? First, they must recognize that automation is a disruptive force for IT system managers. Automation changes the way system administrators do their jobs. It also creates uncertainty in terms of the future of the job itself. And since no automation is perfect, there is concern about the extent to which automation can be trusted to operate correctly.

Thus adoption of automation is unlikely without a history of successful use by administrators. This observation has been proven in practice many times. Useful scripts, languages, and new open-source administration tools tend to gain adoption via grass-roots movements where a few brave early adopters build credibility for the automation technology. But, not all new automation can afford to generate a community-wide grass-roots movement behind it. And in these cases we are left with a kind of circular logic – we cannot gain adoption without successful adoptions!

In these situations, the process-based view can help provide a transition path. From a process perspective, the transition to automation can be seen as an incremental delegation of manual process steps to an automated system. And by casting the automation into the same terms as the previously-manual process steps (for example, by reusing the same work

products and tracking metrics), the automation can provide visibility into its inner workings that assuages a system administrator's distrust. Thus the same work that it takes to do the cost/benefit analysis for the first cut at an automation decision can be leveraged to plan out an automation deployment strategy that will realize the automation's full benefits.

We believe that the process-based perspective described above can provide the basis for making a business case for automation. With a full understanding of the benefits of automation as well as all the cost factors described above, we finally have a framework to answer the question of when to automate. To determine the answer we compute or estimate the benefits of the proposed automation as well as the full spectrum of costs, and compute the net benefit. In essence, we boil the automation decision down to the bottom line, just as in a traditional business case.

The process for building an automation business case consists of 5 steps:

1. Identify the benefit of the automation in terms of reduction in 'core' manual activity, lower frequency of error (quantified in terms of the expected cost of errors, e.g., in terms of business lost during downtime), and increased accuracy of operations.
2. Identify and elucidate the induced processes that result from delegation of the manual activity to automation, as in the software distribution example above. 'Swim-lane' diagrams provide a useful construct for structuring this analysis.
3. Identify or estimate the probability or frequency of the induced processes. For example, in software distribution we estimate how often new packages will be deployed or new versions released, as those events require running the packaging process. Often estimates can be made based on existing runtime data.
4. Assess the possible impact of the 'automation irony'. Will the induced processes require additional skilled resources, or will they change the existing human roles enough that new training or hiring will be needed?
5. Collate the collected data to sum up the potential benefits and costs, and determine whether the automation results in net benefit or net cost.

These steps are not necessarily easy to follow, and in fact in most cases a rough, order-of-magnitude analysis will have to suffice. Alternately, a sensitivity analysis can be performed. For example, if the probability of automation failure is unknown, a sensitivity analysis can be used to determine the failure rate where costs break even. Say this is determined to be one failure per week. If the automation is thought to be stable enough to survive multiple weeks between failures, then the go-ahead decision can be given.

Furthermore, even a rough business-case analysis can provide significant insight into the impact of automation, and going through the exercise will reveal a lot about the ultimate utility of the automation. The level of needed precision will also be determined by the magnitude of the automation decision: a complete analysis is almost certainly unnecessary when considering automation in the form of a few scripts, but it is mandatory when deploying automation to a datacenter with thousands of machines.

4.4. *Measuring complexity*

In this chapter, we have provided a variety of reasons why IT departments of enterprises and service providers are increasingly looking to automation and IT process transformation

as a means of containing and even reducing the labor costs of IT service management. However, Section 4.2 provides evidence of a *reduction* in efficiencies and productivity when automation is introduced into business operations. This raises a critical question from the point of view of both the business owners and CIOs as well as service and technology providers: *how can one quantify, measure and (ultimately) predict whether and how the introduction of a given technology can deliver promised efficiencies?*

Many automation strategies focus on the creation of standardized, reusable components that replace today's custom-built solutions and processes – sometimes at a system level, sometimes at a process level. While both service providers and IT management software vendors heavily rely on qualitative statements to justify investment in new technologies, a qualitative analysis does not provide direct guidance in terms of *quantitative* business-level performance metrics, such as labor cost, time, productivity and quality.

Our approach to measuring complexity of IT management tasks is inspired by the widely successful system performance benchmark suites defined by the Transaction Processing Council (TPC) and the Standard Performance Evaluation Corporation (SPEC). In addition, we have borrowed concepts from Robert C. Camp's pioneering work in the area of business benchmarking [11]. Furthermore, the system administration discipline has started to establish cost models: The most relevant work is [13], which generalizes an initial model for estimating the cost of downtime [42], based on the previously established System Administration Maturity Model [34].

In [5], we have introduced a framework for measuring IT system management complexity, which has been subsequently extended to address the specifics of IT service management processes [18]. The framework relies on categorical classification of individual complexities, which are briefly summarized as follows:

Execution complexity refers to the complexity involved in performing the tasks that make up the configuration process, typically characterized by the number of tasks, the context switches between tasks, the number of roles involved in an action and their degree of automation. *Decision complexity*, a sub-category of execution complexity, quantifies decision making according to: (1) the number of branches in the decision, (2) the degree of guidance, (3) the impact of the decision, and (4) the visibility of the impact.

Business item complexity addresses the complexity involved in passing data between the various tasks within a process. Business items can be nested; they range from simple scalar parameters to complex data items, such as build sheets and run books.

Memory complexity takes into account the number of business items that must be remembered, the length of time they must be retained in memory, and how many intervening items were stored in memory between uses of a remembered business item.

Coordination complexity represents the complexity resulting from coordinating between multiple roles. The per-task metrics for coordination complexity are computed based on the roles involved and whether or not business items are transferred.

While we have found it useful to analyze and compare a variety of configuration procedures and IT service management processes according to the aforementioned four complexity dimensions, a vector of complexity scores reflecting such a categorization does not directly reveal the actual labor reduction or cost savings. For both purchasers and vendors of IT service management software, it is this cost savings that is of eminent concern.

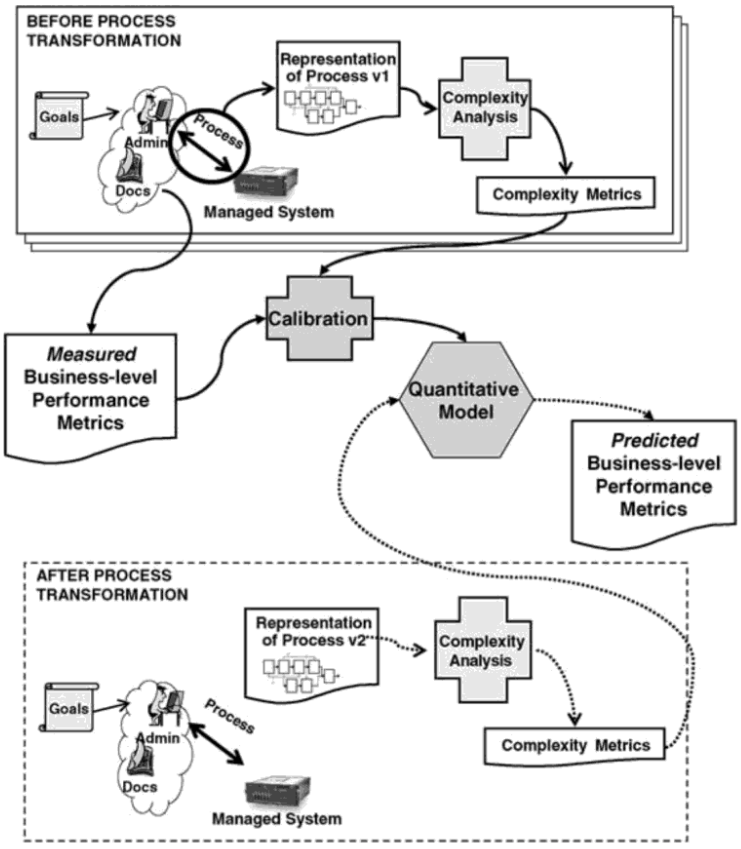


Fig. 4. Approach to constructing and applying a quantitative model.

In order to address this issue, one needs to relate the metrics of our IT management complexity framework to key business-level performance metrics (such as labor cost, effort and duration; cf. Section 4.4.2). In [19], we have developed a quantitative model based on a qualitative description of IT complexity parameters together with quantitative calibration data of the IT process. We found that such a hybrid approach is more practical than a pure qualitative or quantitative approach, since extensive quantitative IT process measurements are notoriously difficult to obtain in the field.

Our approach to predicting labor cost through IT management complexity metrics is depicted in Figure 4. It consists of four distinct steps, which can be summarized as follows:

4.4.1. *Step 1: collecting complexity metrics* As depicted in the upper part of Figure 4, an administrator configures one or more managed systems, according to the goals he wants to accomplish. To do so, he follows a process that may either be described in authoritative documents, or as a workflow in an IT process modeling tool.

This process (called 'Process version 1' in the figure) reflects the current ('as-is') state of the process, which serves as a baseline for our measurements. While carrying out the configuration process, an administrator logs – by interacting with a web-based graphical user interface [18] – the various steps, context switches and parameters that need to be input or produced by the process. In addition, the administrator assesses the complexity for each of the parameters, as perceived by him. The complexity data that an administrator records by means of the web-based graphical user interface is recorded by our tooling on a step-by-step basis. The key concept of the process representation is that it is 'action-centric', i.e., the representation decomposes the overall process into individual, atomic actions that represent the configurations steps an administrator goes through.

Once the process has been carried out, the process capture is input to a complexity analysis tool that we implemented, which contains the scoring algorithms for a variety of complexity measures from the four complexity dimensions mentioned above. The tool also outputs aggregated complexity metrics, such as the maximum number of parameters that need to be kept in memory during the whole procedure, the overall number of actions in the process, or the context switches an administrator needs to perform. Both the individual and the aggregated complexity metrics are used to construct the quantitative model.

We note that by assigning a complexity score to each configuration action, it is possible to identify those actions in the process (or combinations of actions) that have the greatest contribution to complexity (i.e., they are complexity hotspots). As a result, the methodology facilitates the task of process designers to identify targets for process transformation and optimization: If, for example, a given step has a very high complexity relative to other steps in the process, a designer obtains valuable guidance on which action(s) his improvement efforts need to focus first in order to be most effective.

4.4.2. *Step 2: measuring business-level performance metrics* The second step consists in measuring the business-level performance metrics. The most common example of a business-level metric, which we use for our analysis, is the time it takes to complete each action in the process, and the aggregated, overall process execution time. This is also commonly referred to as the *Full Time Equivalent (FTE)*. Labor Cost, another key business-level performance metric, is derived in a straightforward way by multiplying the measured FTEs with the administrator's billable hourly rate.

When assessing the time it takes to complete a task or the overall process, great care must be taken to distinguish between the *effort* (how long it actually takes to carry out a task) and the *duration*, the agreed upon time period for completing a task, which is often specified in a Service Level Agreement. While, for example, the effort for the installation of a patch for a database server may be only 30 minutes, the overall duration may be 2 days. Often, the difference is the time period that the request sits in the work queue of the administrator.

While our tooling allows an administrator to record the time while he captures the actions in a configuration process, the time measurements are kept separate from the complexity metrics.

4.4.3. *Step 3: model construction through calibration* The step of constructing a quantitative model by means of calibration, depicted in the middle of Figure 4, is at the heart

of the approach. Calibration relates the complexity metrics we computed in step 1 to the FTE measurements we performed in step 2. To do so, we have adapted techniques we initially developed in [30] (where we were able to predict the user-perceived response time of a web-based Internet storefront by examining the performance metrics obtained from the storefront's database server) to the problem domain of IT management complexity. The purpose of calibration is to set the time it takes to execute a configuration process in relation to the recorded complexity metrics, i.e., the former is being explained by the latter.

4.4.4. *Step 4: predict business-level performance metrics* Once a quantitative model has been built based on the 'as-is' state of a process, it can be used to predict the FTEs and therefore the labor cost for an improved process that has been obtained through process transformation based on analyzing and mitigating the complexity hotspots that were identified in step 1. This 'to-be' process is referred to as 'Process version 2' in the bottom part of the figure as it accomplishes the very same goal(s) as the 'as-is' version 1 of the process. It is therefore possible to not only apply the quantitative model that has been developed for the 'as-is' state in order to estimate the FTEs from the complexity metrics of the 'to-be' state, but also to directly compare both the complexity metrics and the FTEs of the before/after transformation versions of the process. The difference between the 'as-is' and the 'to-be' FTEs and, thus, labor cost yields the savings that are obtained by process transformation. For a detailed example of applying the approach to an install scenario, the reader is referred to [19].

We envision a future scenario where our validated quantitative model is built directly into an IT process modeling tool, so that a process designer can simulate the execution of an IT management process during the design phase. The designer is then able to obtain the predicted FTEs by running the complexity metrics that were automatically collected from the process model – by means of a plugin – through the quantitative model. Such a quantitative model has many benefits:

First, the model can be used to assist in return-on-investment (ROI) determination, either a-priori or post-facto. Across multiple products (for example, IT offerings from different providers), a customer can use models involving each product for a comparison in terms of the impact on the bottom line. Conversely, IT sales personnel can make a substantiated, quantitative argument during a bidding process for a contract.

Second, using the cost prediction based on the model, the customer can further use the model output for budgeting purposes.

Third, a calibrated model for a particular process can reveal which are the important factors that contribute to the overall complexity of the process, along with a measure of their relative contributions. IT providers can use this data to improve their products and offerings, focusing on areas which yield the largest customer impact.

Fourth, a particular process can also be studied in terms of its sensitivity to skill levels of individual roles. Because labor cost of different skill levels varies, this sensitivity analysis can be used for hiring and employee scheduling purposes.

Finally, process transformation involves cost/benefit analysis. These decisions can be guided by the quantitative predictions from the model.

5. Conclusions

This chapter addresses the automation of system administration, a problem that is addressed as a set of three interrelated questions: what to automate, how to automate, and when to automate.

We address the 'what' question by studying what system administrators do and therefore where automation provides value. An observation here is that tasks such as configuration and installation consume a substantial fraction of the time of systems administrators, approximately 20%. This is fortunate since it seems likely that it is technically feasible to increase the degree of automation of these activities. Unfortunately, meetings and other forms of communication consume almost of third of the time of systems administrators. Here, there seems to be much less opportunity to realize benefits from automation.

The 'how' question is about technologies for automation. We discuss three approaches – rule-based systems, control theoretic approaches, and automated workflow construction. All three have been used in practice. Rules provide great flexibility in building automation, but the complexity of this approach becomes problematic as the scope of automation increases. Control theory provides a strong theoretical foundation for certain classes of automation, but it is not a universal solution. Workflow has appeal because its procedural structure is a natural way for humans to translate their activities into automation.

The 'when' question is ultimately a business question that should be based on a full understanding of the costs and benefits. The traditional perspective has been that automation is always advantageous. However, it is important to look at the full costs imposed by automation. For example, automating software distribution requires that: (a) the distribution infrastructure be installed and maintained; (b) software packages be prepared in the format required by the distribution infrastructure; and (c) additional tools be provided to handle problems with packages that are deployed because of the large scale of the impact of these problems. While automation often provides a net benefit despite these costs, we have observed cases in which these costs exceed the benefits.

As the scale of systems administration increases and new technologies for automation are developed, systems administrators will have even greater challenges in automating their activities.

References

[1] T.F. Abdelzaher and N. Bhatti, *Adaptive content delivery for Web server QoS*, International Workshop on Quality of Service, London, UK (1999), 1563–1577.

[2] E. Altman, T. Basar and R. Srikant, *Congestion control as a stochastic control problem with action delays*, Automatica **35** (1999), 1936–1950.

[3] L. Bainbridge, *The ironies of automation*, New Technology and Human Error, J. Rasmussen, K. Duncan and J. Leplat, eds, Wiley (1987).

[4] R. Barrett, P.P. Maglio, E. Kandogan and J. Bailey, *Usable autonomic computing systems: The system administrators' perspective*, Advanced Engineering Informatics **19** (2006), 213–221.

[5] A. Brown, A. Keller and J.L. Hellerstein, *A model of configuration complexity and its application to a change management system*, 9th International IFIP/IEEE Symposium on Integrated Management (IM 2005), IEEE Press (2005), 531–644.

[6] A.B. Brown and J.L. Hellerstein, *Reducing the cost of IT operations – Is automation always the answer?*, Tenth Workshop on Hot Topics in Operating Systems (HotOS-X), Santa Fe, NM (2005).

[7] A.B. Brown and A. Keller, *A best practice approach for automating IT management processes*, 2006 IEEE/IFIP Network Operations and Management Symposium (NOMS 2006), Vancouver, BC, Canada (2006).

[8] M. Burgess, *On the theory of system administration*, Science of Computer Programming **49** (2003).

[9] M. Burgess and R. Ralston, *Distributed resource administration using cfengine*, Software Practice and Experience **27** (1997), 1083.

[10] P. Byers, *Database Administrator: Day in the Life*, Thomson Course Technology, http://www.course.com/careers/dayinthelife/dba_jobdesc.cfm.

[11] R.C. Camp, *Benchmarking – The Search for Industry Best Practices that Lead to Superior Performance*, ASQC Quality Press (1989).

[12] A. Couch, N. Wu and H. Susanto, *Toward a cost model for system administration*, 19th Large Installation System Administration Conference (2005).

[13] A.L. Couch, N. Wu and H. Susanto, *Toward a cost model for system administration*, Proc. 19th Large Installation System Administration Conference (LISA'05), D.N. Blank-Edelman, ed., USENIX Association, San Diego, CA, USA (2005), 125–141.

[14] Y. Diao, N. Gandhi, J.L. Hellerstein, S. Parekh and D. Tilbury, *Using MIMO feedback control to enforce policies for interrelated metrics with application to the Apache Web server*, IEEE/IFIP Network Operations and Management Symposium (NOMS 2002) (2002), 219–234.

[15] Y. Diao, R. Griffith, J.L. Hellerstein, G. Kaiser, S. Parekh and D. Phung, *A control theory foundation for self-managing systems*, Journal on Selected Areas of Communications **23** (2005).

[16] Y. Diao, J.L. Hellerstein and S. Parekh, *Optimizing quality of service using fuzzy control*, IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM 2002) (2002), 42–53.

[17] Y. Diao, J.L. Hellerstein, A. Storm, M. Surendra, S. Lightstone, S. Parekh and C. Garcia-Arellano, *Using MIMO linear control for load balancing in computing systems*, American Control Conference (2004), 2045–2050.

[18] Y. Diao and A. Keller, *Quantifying the complexity of IT service management processes*, Proceedings of 17th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM'06), Springer-Verlag, Dublin, Ireland (2006).

[19] Y. Diao, A. Keller, V.V. Marinov and S. Parekh, *Predicting labor cost through IT management complexity metrics*, 10th International IFIP/IEEE Symposium on Integrated Management (IM 2007), IEEE Press (2007).

[20] B. Dijker, *A day in the life of systems administrators*, http://www.sage.org/field/ditl.pdf.

[21] J. Finke, *Automation of site configuration management*, Proceedings of the Eleventh Systems Administration Conference (LISA XI), USENIX Association, Berkeley, CA (1997), 155.

[22] A. Finkel, K. Houck and A. Bouloutas, *An alarm correlation system for heterogeneous networks*, Network Management and Control **2** (1995).

[23] M. Fisk, *Automating the administration of heterogeneous LANS*, Proceedings of the Tenth Systems Administration Conference (LISA X), USENIX Association, Berkeley, CA (1996), 181.

[24] S.E. Hansen and E.T. Atkins, *Automated system monitoring and notification with swatch*, Proceedings of the Seventh Systems Administration Conference (LISA VII), USENIX Association, Berkeley, CA (1993), 145.

[25] C.V. Hollot, V. Misra, D. Towsley and W.B. Gong, *A control theoretic analysis of RED*, Proceedings of IEEE INFOCOM'01, Anchorage, Alaska (2001), 1510–1519.

[26] C.V. Hollot, V. Misra, D. Towsley and W.B. Gong, *On designing improved controllers for AQM routers supporting TCP flows*, Proceedings of IEEE INFOCOM'01, Anchorage, Alaska (2001), 1726–1734.

[27] IT Infrastructure Library, *ITIL service support, version 2.3*, Office of Government Commerce (2000).

[28] G. Jakobson, R. Weihmayer and M. Weissman, *A domain-oriented expert system shell for telecommunications network alarm correlation*, Network Management and Control, Plennum Press (1993), 277–288.

[29] A. Keller and R. Badonnel, *Automating the provisioning of application services with the BPEL4WS workflow language*, 15th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM 2004), Springer-Verlag, Davis, CA, USA (2004), 15–27.

[30] A. Keller, Y. Diao, F.N. Eskesen, S.E. Froehlich, J.L. Hellerstein, L. Spainhower and M. Surendra, *Generic on-line discovery of quantitative models*, IEEE Eight Symposium on Network Management (2003), 157–170.

[31] S. Keshav, *A control-theoretic approach to flow control*, Proceedings of ACM SIGCOMM'91 (1991), 3–15.

[32] S. Kliger, S. Yemini, Y. Yemini, D. Ohlse and S. Stolfo, *A coding approach to event correlation*, Fourth International Symposium on Integrated Network Management, Santa Barbara, CA, USA (1995).

[33] M. Kloppmann, D. Koenig, F. Leymann, G. Pfau and D. Roller, *Business process choreography in WebSphere: Combining the power of BPEL and J2EE*, IBM Systems Journal **43** (2004).

[34] C. Kubicki, *The system administration maturity model – SAMM*, Proc. 7th Large Installation System Administration Conference (LISA'93), USENIX Association, Monterey, CA, USA (1993), 213–225.

[35] K. Li, M.H. Shor, J. Walpole, C. Pu and D.C. Steere, *Modeling the effect of short-term rate variations on tcp-friendly congestion control behavior*, Proceedings of the American Control Conference (2001), 3006–3012.

[36] D. Libes, *Using expect to automate system administration tasks*, Proceedings of the Fourth Large Installation System Administrator's Conference (LISA IV), USENIX Association, Berkeley, CA (1990), 107.

[37] Y. Lu, A. Saxena and T.F. Abdelzaher, *Differentiated caching services: A control-theoretic approach*, International Conference on Distributed Computing Systems (2001), 615–624.

[38] Microsoft, *Overview of the system description model*, http://msdn2.microsoft.com/en-us/library/ms181772.aspx.

[39] K.R. Milliken, A.V. Cruise, R.L. Ennis, A.J. Finkel, J.L. Hellerstein, D.J. Loeb, D.A. Klein, M.J. Masullo, H.M. Van Woerkom and N.B. Waite, *YES/MVS and the automation of operations for large computer complexes*, IBM Systems Journal **25** (1986), 159–180.

[40] S. Parekh, K. Rose, J.L. Hellerstein, S. Lightstone, M. Huras and V. Chang, *Managing the performance impact of administrative utilities*, 14th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM 2003) (2003), 130–142.

[41] S. Parekh, N. Gandhi, J. Hellerstein, D. Tilbury, J. Bigus and T.S. Jayram, *Using control theory to achieve service level objectives in performance management*, Real-time Systems Journal **23** (2002), 127–141.

[42] D. Patterson, *A simple way to estimate the cost of downtime*, Proc. 16th Large Installation System Administration Conference (LISA'05), A.L. Couch, ed., USENIX Association, Philadelphia, PA, USA (2002), 185–188.

[43] C. Perrow, *Normal Accidents: Living with High-Risk Technologies*, Perseus Books (1990).

[44] J. Rasmussen and W. Rouse, eds, *Human Detection and Diagnosis of System Failures: Proceedings of the NATO Symposium on Human Detection and Diagnosis of System Failures*, Plenum Press, New York (1981).

[45] J. Reason, *Human Error*, Cambridge University Press (1990).

[46] F.E. Sandnes, *Scheduling partially ordered events in a randomized framework – empirical results and implications for automatic configuration management*, Proceedings of the Fifteenth Systems Administration Conference (LISA XV), USENIX Association, Berkeley, CA (2001), 47.

[47] P. Scott, *Automating* 24×7 *support response to telephone requests*, Proceedings of the Eleventh Systems Administration Conference (LISA XI), USENIX Association, Berkeley, CA (1997), 27.

[48] L. Sha, X. Liu, Y. Lu and T. Abdelzaher, *Queueing model based network server performance control*, *IEEE RealTime Systems Symposium* (2002), 81–90.

[49] E. Solana, V. Baggiolini, M. Ramlucken and J. Harms, *Automatic and reliable elimination of e-mail loops based on statistical analysis*, Proceedings of the Tenth Systems Administration Conference (LISA X), USENIX Association, Berkeley, CA (1996), 139.

[50] A. Somayaji and S. Forrest, *Automated response using system-call delays*, Proceedings of the 9th USENIX Security Symposium, USENIX Association, Berkeley, CA (2000), 185.

[51] SPEC Consortium, *SPECjAppServer2004 design document, version 1.01* (2005), http://www.specbench.org/osg/jAppServer2004/docs/DesignDocument.html.

[52] D. Thoenen, J. Riosa and J.L. Hellerstein, *Event relationship networks: A framework for action oriented analysis in event management*, International Symposium on Integrated Network Management (2001).

[53] M. Vitaletti, ed., *Installable unit deployment descriptor specification, version 1.0*, W3C Member Submission, IBM Corp., ZeroG Software, InstallShield Corp., Novell (2004), http://www.w3.org/Submission/2004/SUBM-InstallableUnit-DD-20040712/.

[54] T. Woodall, *Network Administrator: Day in the Life*, Thomson Course Technology, http://www.course.com/careers/dayinthelife/networkadmin_jobdesc.cfm.

– 1.4 –

System Configuration Management

Alva L. Couch

Computer Science, Tufts University, 161 College Avenue, Medford, MA 02155, USA
E-mail: couch@cs.tufts.edu

1. Introduction

System configuration management is the process of maintaining the function of computer networks as holistic entities, in alignment with some previously determined policy [28]. A *policy* is a list of high-level goals for system or network behavior. This policy – that describes how systems should behave – is translated into a low-level *configuration*, informally defined as the contents of a number of specific files contained within each computer system. This policy translation typically involves installation of predefined hardware and software components, reference to component documentation and experience, and some mechanism for controlling the contents of remotely located machines (Figure 1).

For example, a high-level policy might describe classes of service to be provided; a 'web server' is a different kind of machine than a 'mail server' or a 'login server'. Another example of a high-level goal is that certain software must be either available or unavailable to interactive users (e.g., programming tools and compilers) [23].

The origin of the phrase 'system configuration management' is unclear, but the practice of modern system configuration management was, to our knowledge, first documented in Paul Anderson's paper [2] and implemented in the configuration tool *LCFG* [2,4]. This paper contains several ideas that shaped the discipline, including the idea that the configuration of a system occurs at multiple levels, including a high-level policy, intermediate code and physical configuration.

HANDBOOK OF NETWORK AND SYSTEM ADMINISTRATION
Edited by Jan Bergstra and Mark Burgess

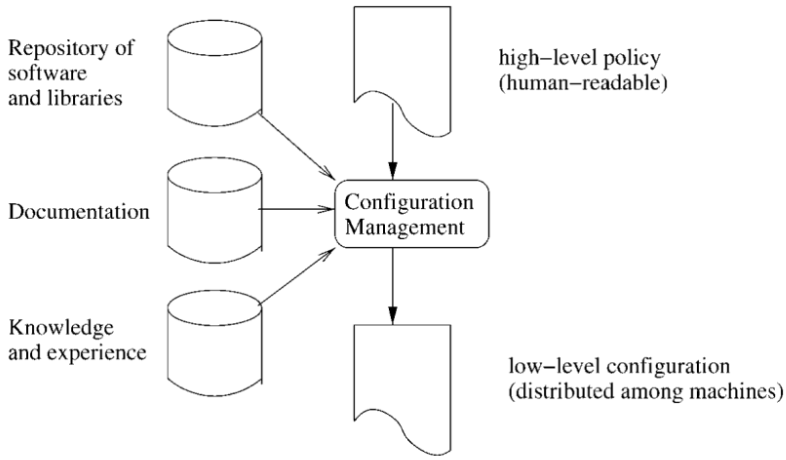


Fig. 1. An overview of the configuration management process.

1.1. *Overview of configuration management*

A typical system to be managed consists of a central-processing unit, hard disk and associated peripherals, including network card, video card, etc. Depending upon the system, many of these components may actually be contained on a single computer board. A system typically starts its life with no operating system installed, often called 'bare metal'. As software components are installed on the hard disk, the system obtains an operating system, system modules that extend the operating system (e.g., I/O drivers), and software packages that provide application-level functions such as web service or compilation. During this process, each operating system, module, or package is assigned a *default configuration* specifying how it will behave. This is best thought of as a set of variables whose values control behavior. For example, one variable determines the number of threads utilized by a web server, while another variable determines whether certain kinds of network tunneling are allowed. The configuration of each module or package may be contained in text files (for Unix and Linux) or in a system database (e.g., the Windows Registry). The configuration of a component may be a file of key-value associations (e.g., for the secure shell), an XML file (e.g., for the Apache web server), or a registry key-value hierarchy in Windows. Several crucial configuration files are shared by most Linux applications and reside in the special directory `/etc`. Configuration management is the process of defining and maintaining consistent configurations for each component so that they all work together to accomplish desired tasks and objectives.

1.2. *Configuration management tasks*

There are several tasks involved in typical configuration management (Figure 2). First, *planning* involves deciding what constitutes desirable behavior and describing an overall operating *policy*. The next step is to *code* this policy into machine-readable form. This step

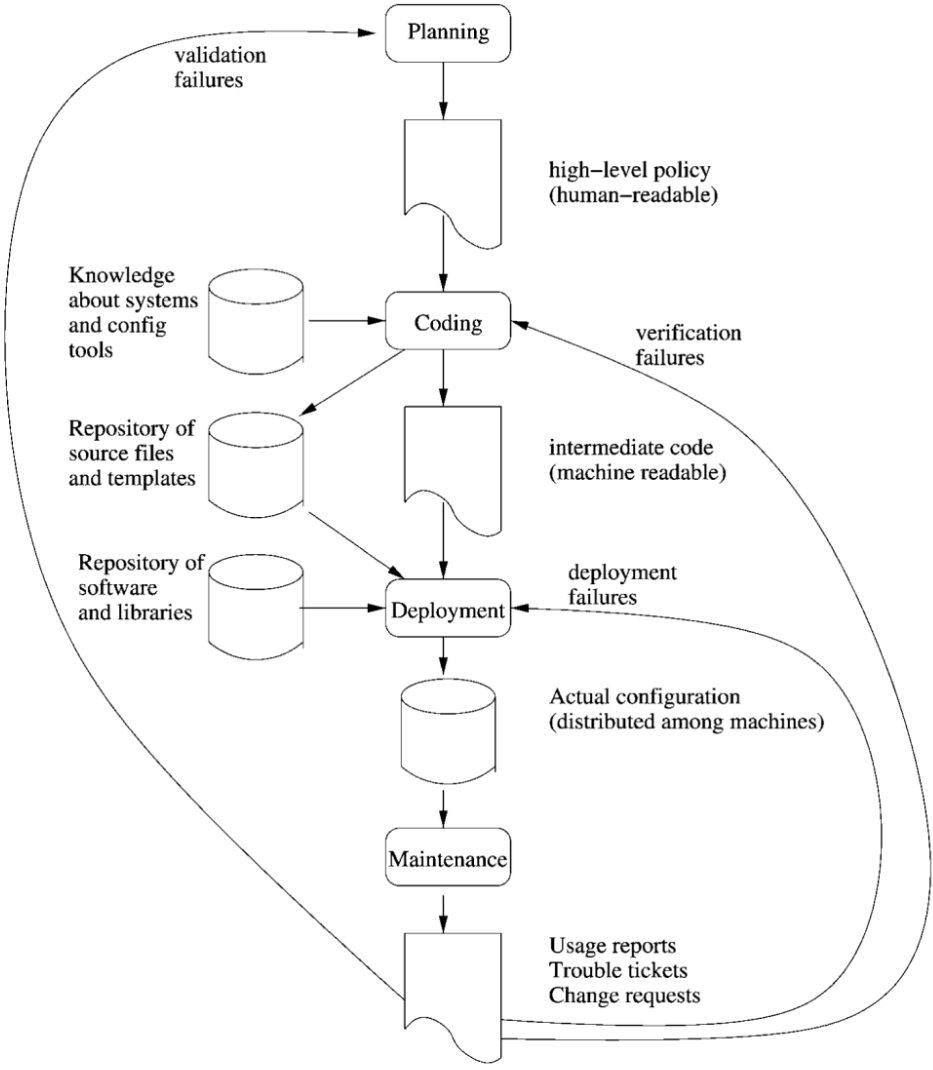


Fig. 2. The configuration management process includes creating high-level policies, translating them into machine-readable form, and deploying the configuration throughout a network.

utilizes knowledge about machines and configurations to create both the intermediate code and a repository of *source and template files* that can be used to create individual machine configurations. A source file is a verbatim copy of some configuration file, while an template file is a configuration file with some details left out, to be filled in later. The next step is to *deploy* the configuration on target machines, which involves changing the contents of each machine's disks. After deployment, the configuration must be *maintained* through an

ongoing process of quality control, including data sources such as trouble tickets and new user requests.

At each step, there can be failures. A *verification failure* is a failure to create a system with a proper configuration, while a *validation failure* occurs when a correctly deployed configuration fails to meet user requirements.[1] For example, if a file fails to be copied to a machine to be configured, this is a verification failure, while a validation failure results if a machine that is intended to become a web server does not provide web services after being configured to do so. The former is a mistake in coding; the latter is a behavioral problem. A system fails to verify if the information that should be in configuration files on each machine does not ever get copied there; a system fails to validate if the configuration files are precisely as intended but the behavior fails to meet specifications.

1.3. *Objectives of configuration management*

It may seem to the reader at this point that configuration management is just about assuring the behavior of machines and nothing else. If this were the only goal of configuration management, it would be a simple and straightforward task. Configuration management is easy if one has access to unlimited hardware, and can encapsulate each distinct service on a separate machine. The complexities of configuration management arise from the need to combine or compose distinct and changing needs while utilizing limited amounts of hardware. For more details on complexity, see Section 9.5 and the chapter by Sun and Couch on Complexity of System Configuration Management.

In fact, the true goal of configuration management is to *minimize the lifecycle cost* and *total cost of ownership* for a computing system, while *maximizing value* of the computing system to the enterprise [36]. The cost of management has several components, including the cost of planning, deployment and maintenance. As in software engineering, the cost of maintenance seems to dominate the others for any lifecycle of reasonable length.

The relationships between configuration management and cost are many and varied. There is an intimate relationship between the methods utilized for configuration management and the ability to react to contingencies such as system outages quickly and effectively. One characterization of an effective configuration management strategy is that one should be able to pick a random server, unplug it and toss it out of a second story window, and *without backup*, be up and running again in an hour [73].

2. Relationship with other disciplines

2.1. *System administration tasks*

We define:

[1] The words ‘verification’ and ‘validation’ are used here in the precise software engineering sense, though in system administration, the word ‘validation’ has often been used to mean ‘verification’. For more details, see [8].

1. *Configuration management* is the process of insuring that the operating system and software components work properly and according to externally specified guidelines.
2. *Package management* is the process of insuring that each system contains software components appropriate to its assigned task, and that components interact properly when installed.
3. *Resource management* is the process of insuring that appropriate resources (including memory, disk and network bandwidth) are available to accomplish each computing task.
4. *User management* is the process of assuring that users have access to computing resources, through appropriate authentication and authorization mechanisms.
5. *Monitoring* is the process of measuring whether computing systems are correctly performing their assigned missions.

These activities are not easily separable: one must install software through package management before one can configure it via configuration management, and monitor it via monitoring mechanisms. Users must be able to access the configured software, and have appropriate resources available from which to complete their tasks.

This is an inevitable convergence; part of monitoring is to ensure that configuration management is working properly, and an ideal configuration management tool would employ both static analysis and 'dynamic feedback' to understand whether the configuration is working properly and whether corrective action is needed.

2.2. *Configuration management and system administration*

System administration and system configuration management appear on the surface to be very similar activities. In both cases, the principal goal is to assure proper behavior of computing systems. The exact boundary between 'system configuration management' and regular 'system administration' is unclear, but there seems to be a solid distinction between the two at the point at which the overall configuration of a network of systems is being managed as an entity, rather than managing individual machines that contain distinct and unrelated configurations. At this point, we say that one is engaging in 'system configuration management' rather than merely acting as a system administrator for the same network.

3. Configuration management and systems architecture

There is an implicit assumption that configuration management begins with an overall 'architectural plan' of how services will be mapped to servers, and that changes in overall architecture are not allowed after that architecture has been mapped out. This is an unfortunate distinction for several reasons:

1. The choice of architecture often affects the cost of management. A classical example of this is that cramming many services onto a single server seems to save money but in fact, the cost of configuration management might be dramatically higher due to conflicts and constraints between services.

2. The exact impact of architectural choices may not be known until one considers how the architecture will be managed. For example, it might be unclear to someone not involved in system administration that certain pairs of web server modules cannot be loaded at the same time without behavior problems. Thus there could be a need for segmentation of web services that is only known to someone in configuration management, and not known to people who might be designing an architecture.
3. Several hard problems of configuration management can be mitigated or even eliminated via architectural changes. For example, one can eliminate subsystem dependencies by locating potentially conflicting subsystems on disjoint or virtual operating systems.

Separating architectural design from configuration management strategy can be a costly and significant mistake, especially because system administrators are often the only persons with enough knowledge to design an efficiently manageable solution architecture.

3.1. *Other kinds of configuration management*

The term *configuration management* has several other meanings with the potential to overlap with the meaning of that term in this chapter.

1. *Software configuration management* [52] (in software engineering) is the activity of insuring that the components of a software product interact correctly to accomplish the objectives of the product. The 'configuration' – in this case – is a set of software module revisions that are 'composed' to form a particular software product.
2. *Network configuration management* is the activity of insuring that routers, switches, load balancers, firewalls, and other networking hardware accomplish the objectives set by management.

System configuration management combines the problems inherent in software configuration management and network configuration management into a single problem. There is the problem of *component composition* [39] that arises in software configuration management, i.e., ensuring that a particular system contains sufficient components to accomplish a task, along with the problem of *component configuration*, of limiting the behavior of components according to some established norms.

Software configuration management in itself is only weakly related to system configuration management. It concentrates upon keeping versions of software components that comprise a specific software product synchronized, so that the resulting product works properly. The key difference between software and system configuration management is that a software product is not usually configured 'differently' for different customers. The goal of software configuration management is to produce a single product to which multiple individuals contribute, but the issue of providing distinct behavior in different environments does not arise except in very advanced cases where vendor customization is requested. In a way, system configuration management takes over where software configuration management leaves off, at the point where a product must be customized by choices for control parameters that are not made within the structure of the software package itself. Making these choices is typically distinct from software compilation.

Network configuration management is closely related to system configuration management, with almost the same objectives. The principal difference is that network devices – unlike computing systems – are seldom extensible in function by adding new programs to the ones they already execute. There is a single program running on each networking device that does not change. The key issues in network configuration management include:

1. Choosing an appropriate version for the single program executing on each device, so that this program interoperates well with programs running on other devices.
2. Determining values for parameters that affect this program and its ability to interoperate with programs on other devices.

Choices for these two parts of configuration are often determined more by device limitations rather than human desires; devices have limited interoperability and not all combinations of software versions will work. Once the component composition has been determined (usually by either reading documentation or by experience), programs that enforce parameter choices usually suffice to manage network configuration.

There are a few issues that arise in system configuration management and nowhere else. First, we must compose and configure software components to accomplish a large variety of tasks, where the success of composition is sometimes unsure. Second, the systems being configured are also mostly active participants in the configuration, and must be given the capability to configure themselves via a series of bootstrapping steps. This leads to problems with sequencing, and requirements that changes occur in a particular order, in ways that are less than intuitive.

4. Cost models

The basic goal of configuration management is to minimize cost and to maximize value (see chapter by Burgess on System Administration and Business), but many sources of cost have been intangible and difficult to quantify. A typical administrative staff has a fixed size and budget, so all that varies in terms of cost is how fast changes are made and how much time is spent waiting for problem resolution. Patterson [61] quantifies the 'cost of downtime' as

$$C_{\text{downtime}} = C_{\text{revenue lost}} + C_{\text{work lost}} \tag{1}$$

$$\approx (C_{\text{avg revenue lost}} + C_{\text{avg work lost}}) \times T_{\text{downtime}}, \tag{2}$$

where

- $C_{\text{revenue lost}}$ represents the total revenue lost due to downtime,
- $C_{\text{work lost}}$ represents the total revenue lost due to downtime,
- $C_{\text{avg revenue lost}}$ represents the average revenue lost per unit of downtime,
- $C_{\text{avg work lost}}$ represents the average work lost per unit of downtime,
- T_{downtime} represents the downtime actually experienced.

Of course, this can be estimated at an even finer grain by noting that the revenue lost is a function of the number of customers inconvenienced, while the total work lost is a function of the number of workers left without services.

Patterson's simple idea sparked other work to understand the nature of cost and value. Configuration management was – at the time – completely concerned with choosing tools and strategies that make management 'easier'. Patterson defined the problem instead as minimizing cost and maximizing value. This led (in turn) to several observations about configuration management that may well transform the discipline in the coming years [36]:

1. Configuration management should minimize *lifecycle cost*, which is the integral of unit costs over the lifecycle of the equipment:

$$C_{\text{lifecycle}} = \int_{t_{\text{start}}}^{t_{\text{end}}} c_r(t)\,\mathrm{d}t, \tag{3}$$

   where $c_r(t)$ is the cost of downtime at time t, and t_{start} and t_{end} represent the lifespan of the system being configured.
2. The instantaneous cost of downtime in the above equation is a sum of two factors:

$$c_r(t) = c_{rm}(t) + c_{ri}(t), \tag{4}$$

   where
   - $c_{rm}(t)$ represents constant costs, such as workers left idle (as in Patterson's model),
   - $c_{ri}(t)$ represents intangible costs, such as contingencies that occur *during* an outage.
3. We can safely approximate contingencies during an outage as multi-class with Poisson arrival rates, so that

$$c_r(t) = c_{rm}(t) + \sum_{d \in D_r} \lambda_d c_d, \tag{5}$$

   where D_r is a set of classes of contingencies, and d represents a contingency class with arrival rate λ_d and potential average cost c_d.
4. To a first approximation, the cost of downtime is proportional to the time spent troubleshooting.
5. Troubleshooting time is controlled by the complexity of the troubleshooting process, as well as the sequence of choices one makes in understanding the problem.
6. The best sequence of troubleshooting tasks is site-specific, and varies with the likelihood of specific outages.

The key discovery in the above discussion is that *good practice is site-relative*.[2] The practices that succeed in minimizing downtime and increasing value depend upon the likelihood of each contingency.

One way to study potential costs is through simulation of the system administration process [36]. We can easily simulate contingencies requiring attention using classical queueing theory; contingencies are nothing more than multi-class queues with different average service times for each class. We can observe arrival frequencies of classes of tasks

[2]The term 'best practice' is often used in this context, albeit speculatively.

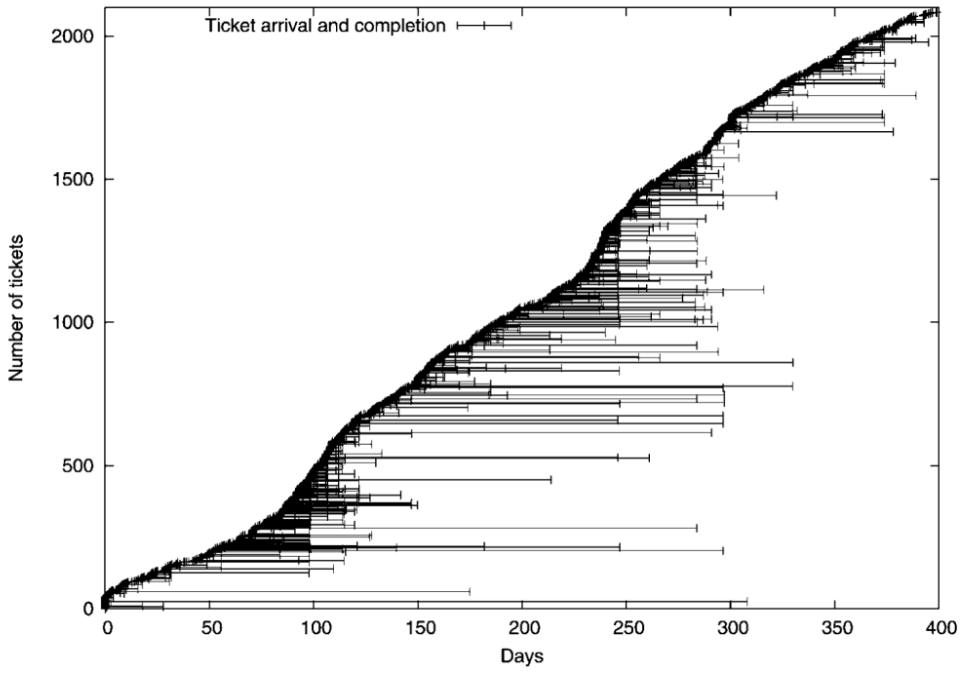


Fig. 3. Data from trouble ticket systems indicates arrival rates and service rates for common tasks. Reprinted from [36] by permission of the authors.

in practice by using real data from trouble ticketing systems (Figure 3), and can utilize this to form an approximate arrival and service models for requests.

Simple simulations show just how sensitive downtime costs are to the number of staff available. In an understaffed situation, downtime costs increase rapidly when request queues expand without bound; the same scenario with more system administrators available shows no such behavior (Figure 4).

Configuration management must balance the cost of management against the cost of downtime. Since sites differ in how much downtime costs, there are several distinct strategies for configuration management that depend upon site goals and mission.

4.1. *Configuration management lifecycles*

To understand the lifecycle cost of configuration management, it helps to first consider the lifecycle of the systems to be configured. Configuration management is an integral and inseparable part of the lifecycle of the computing systems being managed. Each system goes through a series of configuration states in progressing from ‘bare metal’ (with no information on disk) to ‘configured’, and finally to being ‘retired’ (Figure 5). We define a *bare*

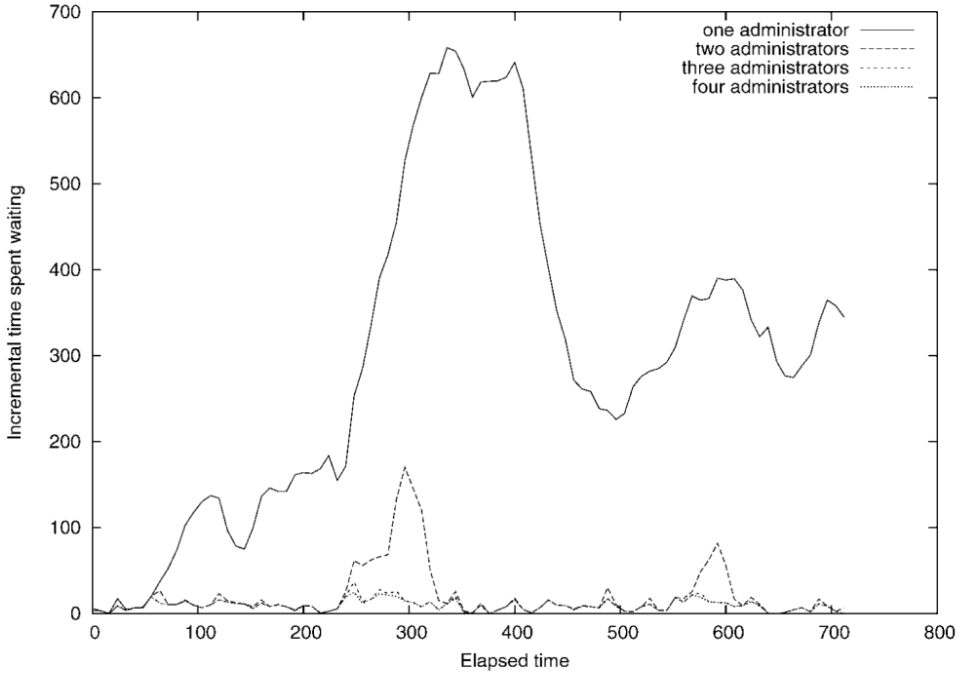


Fig. 4. Chaotic interactions ensue when there are too few system administrators to handle a trouble-ticket queue; the hidden cost is that of downtime due to insufficient staffing. Reprinted from [36] by permission of the authors.

metal system is one in which the operating system has not yet been installed; a *bare-metal rebuild* involves starting from scratch, with a clean disk, and re-installing the operating system, erasing all data on disk in the process. After system building, *initial configuration* gives the system desirable behavioral attributes, after which there is an ongoing *maintenance* phase in which configuration changes are made on an ongoing basis. Finally, the system is *retired* from service in some way.

In almost precise correspondence with the system lifecycle, configuration management itself has its own lifecycle, consisting of planning, deployment, maintenance, and retirement states. *Planning* consists of the work needed in order to begin managing the network as an entity. *Deployment* consists of making tools for configuration management available on every host to be managed. *Maintenance* consists of reacting to changes in policy and contingencies on an ongoing basis, and *retirement* is the process of uninstalling or de-commissioning configuration management software.

Part of what makes configuration management difficult is that the cost of beginning to utilize a particular method for configuration management can be substantial. To decide whether to adopt a method, one must analyze one's needs carefully to see whether the cost of a method is justified.

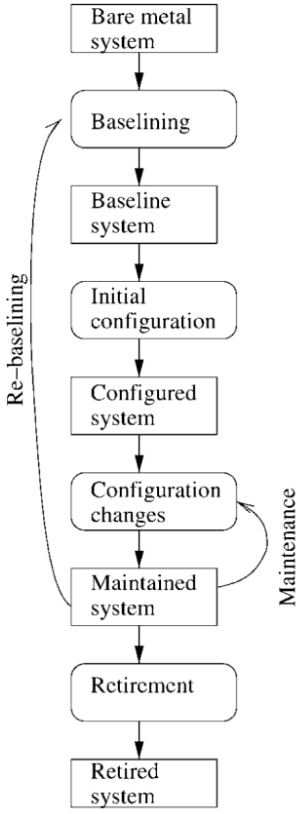


Fig. 5. Managed systems go through a series of states in their lifecycle, from 'bare metal' (unbuilt) to 'retired'.

5. Configuration management strategies

To minimize cost and maximize value, the configuration manager adopts an overall *configuration management strategy* to control the configuration of a network. The management strategy details and documents how configuration changes are to be made, tracked and tested. This strategy is best considered to be independent of the software tools used to enforce the changes. Key components of a typical configuration management strategy include:

1. A mechanism for making changes in the configuration, through tools or practices.
2. A mechanism for tracking, documenting and auditing configuration changes.
3. A mechanism for verifying and validating configurations.
4. A procedure for staging deployment of new configurations, including prototyping and propagating changes.

For example, in an academic research environment, staging may often be skipped; the prototype environment and the production environment may be the same and downtime may not be extremely important.

One should not confuse a choice of configuration management strategy with a particular choice of tools to perform configuration management. Tools can support and ease implementation of particular strategies, but are no substitute for a coherent strategic plan.

There are three main kinds of strategies for configuration management, that we shall call 'scripted', 'convergent' and 'generative' [35]. A *scripted* strategy uses some form of custom scripts to make and manage changes in configuration; the scripts mimic the way an administrator would make the same changes manually [7]. A *convergent* strategy utilizes a special kind of scripts that can be applied multiple times without risk of damage. These scripts are usually safer to employ than unconstrained scripts, but are somewhat more complex in character and more complex than the manual procedures they replace. A *generative* strategy involves generating all of the configuration from some form of database of requirements. Note that all convergent strategies are in some sense scripted, and all generative strategies are (trivially) convergent, in the sense that generating the whole configuration twice yields the same answer as generating it once.

These approaches differ in scope, capabilities and complexity; scripting has a low startup cost, convergent scripting is safer than unrestricted scripting but requires recasting manual procedures in convergent terms, and generative mechanisms require specification of a comprehensive site policy, and thus incur a high startup cost. One crucial choice the practicioner must make in beginning configuration management is to weigh cost and value, and choose between one of these three base strategies.

There are several reasons for adopting a comprehensive management strategy rather than administering machines as independent and isolated entities:

1. To reduce the cost of management by adopting standards and eliminating unnecessary variation and heterogeneity.
2. To reduce downtime due to troubleshooting.
3. To assure reproducibility of systems during a contingency.
4. To increase the interchangeability of administrative staff in dealing with changes in configuration.
5. To increase the organizational maturity of the system administration organization.

The last one of these deserves careful consideration. The basic idea of the Capabilities Maturity Model (CMM) of software engineering [20,21] is that the quality of a software product is affected by the 'maturity' of the organization that produces it. Maturity measures include size of programming staff, consistency and documentability of engineering process in creating software, interchangeability of function among programming staff, and ability to deal with contingencies. Applying this principle to system administration [51], we find that important factors in the maturity of a system administration organization include documentation of process, reproducibility of results, and interchangeability of staff. All of these maturity factors are improved by the choice of a reasonable configuration management strategy. The strategy defines how results are to be documented, provides tools for reproducing results and allows staff to easily take over configuration management when system administrators leave and enter the organization.

5.1. *The effect of site policy*

Configuration management is a discipline in which the nature of one's goals determines the best possible management strategy. Thus the choice of tools is dependent upon the choice of operating policies.

Let us consider two possible extremes for site policy, that we shall call 'loose' and 'tight'. At a 'tight' site, there is a need for operations to be auditable and verifiable, and for absolute compliance between policy and systems. Examples of 'tight' organizations include banks and hospitals. In these situations, mistakes in configuration cost real money, both in terms of legal liability and lost work. In some cases a configuration mistake could threaten the life of a person or the organization as a whole.

By contrast, at a 'loose' site, the consequences of misconfiguration are less severe. In a development lab, e.g., the risks due to a misconfiguration are much smaller. The same is true for many academic labs.

Of course, many sites choose to function between the two extremes. For example, within a startup firm, it is common practice to skimp on system administration and cope with lack of reliability, in order to save money and encourage a liquid development process. However, even in such an organization, *certain* systems – including accounts payable and receivable – must be managed to a 'tight' set of standards. This has led many startups to 'outsource' critical business functions rather than running them in-house. The choice of 'loose' or 'tight', however, comes with a number of other choices that greatly affect one's choice of management strategy.

First, we consider the 'tight' site. In this kind of site:

1. Security risks are high, leading to a lack of trust in binary packages not compiled from source code.
2. Thus many packages are compiled from source code rather than being installed from vendor-supplied binary packages.
3. Configuration is considered as a one-time act that occurs during system building. Changes are not routinely propagated.
4. Almost any deviation from desired configuration is handled by rebuilding the system from scratch. Thus 'immunizing' methods are not considered important, and may even obstruct forensic analysis of security break-ins.

Thus the ability to rebuild systems is most important. Day-to-day changes are not as important to handle.

The administrator of a 'tight' site is not expected to react to many changes. Instead, the configuration is specified as a series of steps toward rebuilding a system. 'Convergent' methods are considered irrelevant, and 'generative' methods are simply unnecessary. All that is needed for a 'tight' site is a script of commands to run in order to assure the function of the site, and no more. Thus 'managed scripting' is the mechanism of choice.

By contrast, consider the 'loose' site, e.g., a development laboratory or academic research site. In this kind of site:

1. Security risks are low.
2. It might be important to have up-to-date tools and software.

3. Manpower is often extremely short.
4. Changes are extremely frequent, thus, it is impractical to rebuild systems from scratch for every change.

Here some form of 'generative' or 'convergent' strategy could be advisable.

5.2. *Making changes*

With an understanding of the nature of one's site in hand, the task of the configuration manager is to choose a strategy that best fits that site's mission. We now turn to describing the components of an effective configuration management strategy in detail.

The most crucial component in any configuration management strategy is the ability to make changes in configuration. Change mechanisms usually involve a *configuration description* that specifies desired qualities of the resulting (possibly distributed) system in an intermediate language, as well as some *software tool or infrastructure* that interprets this and propagates changes to the network.

The process of actually making changes to configuration is so central to system configuration management that there is a tendency to ignore the other components of a management strategy and concentrate solely upon the change propagation mechanism. In a simple configuration management strategy, the configuration file suffices as documentation and as input for auditing and verification of the results, while a software tool provides mechanisms for propagating those changes. As of this writing, an overwhelming majority of sites use an extremely simple configuration management strategy of this kind, first noted by [43] and apparently still true today.

5.3. *Tracking and auditing changes*

A related problem in configuration management is that of tracking and auditing changes to machines after they are made. Today, many sites use only a simple form of tracking, by keeping a history of versions of the configuration description via file revision control [70]. A small number of sites, motivated by legal requirements, must use stricter change control mechanisms in which one documents a *sequence of changes* rather than a *history of configurations*.

Today, few sites track *why* changes are made in a formal or structured way, although one could argue that the changes themselves are meaningless without some kind of motivation or context. A change can often be made for rather subtle reasons that are soon forgotten, for example, to address a hidden dependency between two software packages. In the absence of any documentation, the system administrator is left to wonder why changes were made, and must often re-discover the constraint that motivated the change, all over again, the next time the issue arises. Thus, effective documentation – both of the systems being configured and the motives behind configuration features – is a cost-saving activity if done well.

5.4. *Verification and validation*

Verification and validation are technical terms in software engineering whose meaning in system configuration management is often identical.

1. *Verification* is the process of determining whether the software tools or procedures accomplish changes in accordance with the configuration policy.
2. *Validation* is the process of determining whether the intermediate policy code actually produces a system that satisfies human goals.

In the words of Fredrick Brooks [8], *verification* is the process of determining "whether one is making the product right", while *validation* is the process of determining "whether one is making the right product".

Failures of validation and verification bring 'loops' into the configuration management process (see Figure 2). A failure of verification requires recoding the intermediate representation of configuration and propagating changes again, while a failure of validation requires modifying the original operating policy.

Currently, many software tools for configuration management have strong mechanisms for verification: this is the root of 'convergent' management strategies as discussed in Section 13. In verification, the agreement between source descriptions and target files on each computer is constantly checked, on an ongoing basis, and lack of agreement is corrected.

Currently, the author is not aware of any tools for *validation* of system configurations.[3] While validation has been a central part of software engineering practice for the last 30 years, it has been almost ignored as part of configuration management. It is almost entirely performed by a human administrator, either alone or by interacting with users. Traditionally, validation has been mislabeled as other activities, including 'troubleshooting', 'help desk', etc. User feedback is often utilized as a substitute for quality assurance processes. For more details on structured testing and its role in configuration management, see [75].

Why do configurations fail to function properly? There are several causes, including:

(1) Human errors in coding the intermediate representation of configuration;
(2) Errors in propagating changes to configured nodes;
(3) Changes made to configured nodes without the administrator's knowledge;
(4) Errors or omissions in documentation for the systems being configured.

Human error plays a predominant role in configuration failures.

5.5. *Deployment*

Deployment is the process of enforcing configuration changes in a network of computing devices. It can take many forms, from rebuilding all workstations from scratch to invoking an intelligent agent on each host to make small changes.

Staging is the process of implementing a configuration change in small steps to minimize the effect of configuration errors upon the site (Figure 6). A typical staging strategy involves two phases, *prototyping* and *propagation*. In the prototyping phase, a small testbed of representative computers is assembled to test a new configuration.

[3] Note that some tools that perform verification are commonly referred to as 'validation' tools.

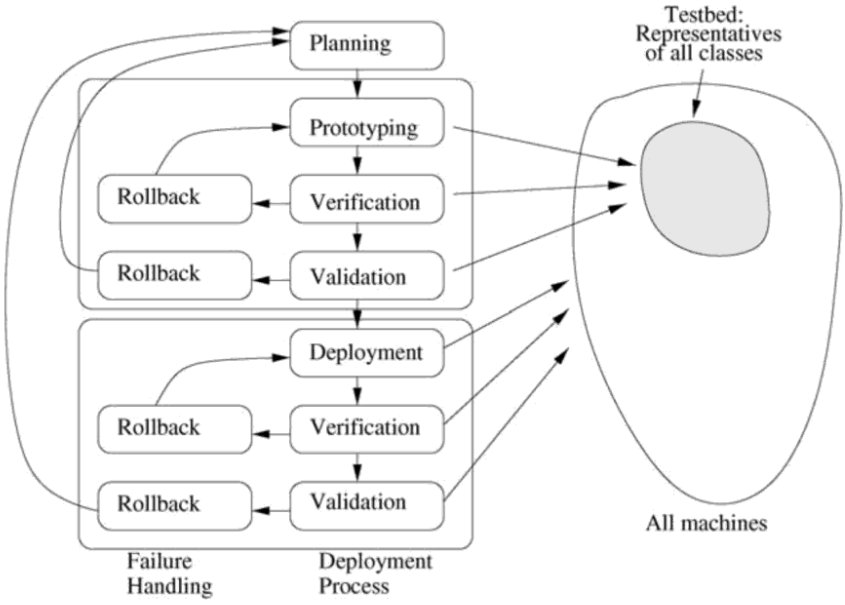


Fig. 6. A staging strategy involves prototyping, testing, actual deployment and potential rollback in case of difficulties.

A good testbed contains at least one computer of each type available in the network. The configuration is first tried on the prototype network, and debugging and troubleshooting irons out the problems. Once that is complete, propagation ensues and the configuration is enforced on all hosts. If this does not work properly, *rollback* may be utilized: this is the process of restoring a previous configuration that was known to function.

Problems due to deployment can cost money in terms of work and revenue lost; see [61] for some statistics. Testing is just one way to assure that deployment does not cause downtime. Another way is to wait a short time before installing a security patch, rather than installing it immediately [37].

6. Challenges of configuration management

Before discussing the traditions that led to current configuration management practices, it is appropriate to understand why configuration management is difficult and deserves further research. These brief remarks are intended to provide a non-practicioner with a background on the subtle challenges and rewards of a good configuration management strategy.

Configuration management would be more straightforward if all machines were identical; then a method for configuring one machine would be reusable for all the others. It would be more straightforward if there were no large-scale networks and no requirement for uptime and disaster recovery; one could simply configure all machines by hand. It would be more straightforward if policy never changed. But none of these is true.

The principal drivers of configuration management as a practice include changes in policy, scale of application, heterogeneity of purpose, and need for timely response to unforeseen contingencies. These have influenced current approaches to configuration management, both by defining the nature of desirable behavior and by identifying problematic conditions that should be avoided.

6.1. *Change*

Perhaps the most significant factor in configuration management cost is that the high-level policy that defines network function is not constant, but rather changes over time. One must be able to react effectively, efficiently, unintrusively and quickly to accomplish desired changes, without unacceptable downtime. Mechanisms for configuration management vary in how quickly changes are accomplished, as well as how efficiently one can avoid and correct configuration mistakes. One recurrent problem is that it is often not obvious whether it is less expensive to change an existing configuration, or to start over and configure the machine from scratch: what practicioners call a 'bare-metal rebuild'.

6.2. *Scale*

Another significant driver of configuration management cost and strategy is the scale of modern enterprise networks. Tasks that were formerly accomplished by hand for small numbers of stations become impractical when one must produce thousands of identical stations, e.g., for a bank or trading company. Scale also poses several unique problems for the configuration manager, no matter which scheme is utilized. In a large-enough network, it is not possible to assure that changes are actually made on hosts when requested; the host could be powered down, or external effects could counteract the change. This leads to strategies for insuring that changes are committed properly and are not changed inappropriately by other external means, e.g., by hand-editing a managed configuration file.

6.3. *Heterogeneity*

Heterogeneity refers to populations of multiple types of machines with differing hardware, software, pre-existing configuration, or mission constraints. Hardware heterogeneity often occurs naturally as part of the maintenance lifecycle; stations persist in the network as new stations are purchased and deployed. A key question in the choice of a configuration management strategy is how this heterogeneity will be managed; via a centrally maintained database of differences [2,4,40,41,44,45], or via a distributed strategy with little or no centralized auditing [9–11,18].

Another kind of heterogeneity is that imposed by utilizing differing solutions to the same problem; e.g., a group of administrators might all choose differing locations in which to install new software, for no particularly good reason. This 'unintentional heterogeneity'

[31] arises from lack of coordination among human managers (or management tools) and not from the site itself.

Unintentional heterogeneity can become a major stumbling block when one desires to transition from one configuration management scheme to another. It creates situations in which it is extremely difficult to construct a script or specification of what to do to accomplish a specific change, because the content of that script would potentially differ on each station to be changed.

For example, consider a network in which the active configuration file for a service is located at a different path on each host. To change this file, one must remember which file is actually being used on a host. One thus has to *remember* where the file is on all hosts, forever. This is common in networks managed by groups of system administrators, where each administrator is free to adopt personal standards. To manage such a network, one must remember *who* set up each host, a daunting and expensive task.

6.4. *Contingency*

A final driver of configuration management cost is contingency. A configuration can be modified by many sources, including package installation, manual overrides, or security breaches. These contingencies often violate assumptions required in order for a particular strategy to produce proper results. This leads to misconfigurations and potential downtime.

Contingencies and conflicts arise even between management strategies. If two tools (or administrators) try to manage the same network, conflicts can occur. Each management strategy or tool presumes – perhaps correctly – that it is the sole manager of the network. As the sole arbiter, each management strategy encourages making arbitrary configuration decisions in particular ways. Conflicts arise when arbiters disagree on how to make decisions. This also means that when an administrator considers adopting a new management strategy, he or she must evaluate the cost of changing strategies, which very much depends upon the prior strategy being used. More often than not, adopting a new strategy requires rebuilding the whole network from scratch.

7. High-level configuration concepts

The first task of the system administrator involved in configuration management is to describe how the configured network should behave. There are two widely used mechanisms for describing behavior at a high level. Machines are categorized into *classes* of machines that have differing behaviors, while *services* describe the relationships between machines in a network.

7.1. *Classes*

The need to specify similarities and differences between sets of hosts led to the concept of *classes*. Classes were introduced in *rdist* [24] as static lists of hosts that should agree

on contents of particular files. Each host in the class receives the same exact file from a server; classes are defined and only meaningful on the *server* that stores master copies of configuration files.

The idea of classes comes into full fruition in *Cfengine* [9–11,18], where a class instead represents a set of hosts that *share a particular dynamic property*, as in some forms of object-oriented programming. In *Cfengine*, class membership is a *client* property, determined when *Cfengine* runs as a result of dynamic tests made during startup. There is little need to think of a class as a list of hosts; rather, a class is something that a client *is* or *is not*.[4]

In either case, a *class* of machines is a set of machines that should share some concept of state or behavior, and many modern tools employ a mix of *rdist*-like and *Cfengine*-like classes to specify those shared behaviors in succinct form. For example, the class 'web server' is often disjoint from the class 'workstation'; these are two *kinds* of hosts with different configurations based upon kind.

Cfengine also introduces two different kinds of classes:

(1) *hard* classes define attributes of machines that cannot change during the software configuration process, e.g., their hardware configurations;
(2) *soft* classes define attributes of machines that are determined by human administrators and the configuration process.

For example, a 'hard class' might consist of those machines having more than a 2.0 GB hard drive, while a 'soft class' might consist of those machines intended to be web servers. It is possible to characterize the difference between hard classes and soft classes by saying that the hard classes are part of the *inputs* to the configuration management process, while the soft classes describe *outputs* of that process.

Figure 7 shows the class diagram for a small computer network with both hard and soft classes of machines. In the figure there are two hard classes, 'x386' and 'PowerPC', that refer to machine architectures. A number of soft classes differentiate machines by

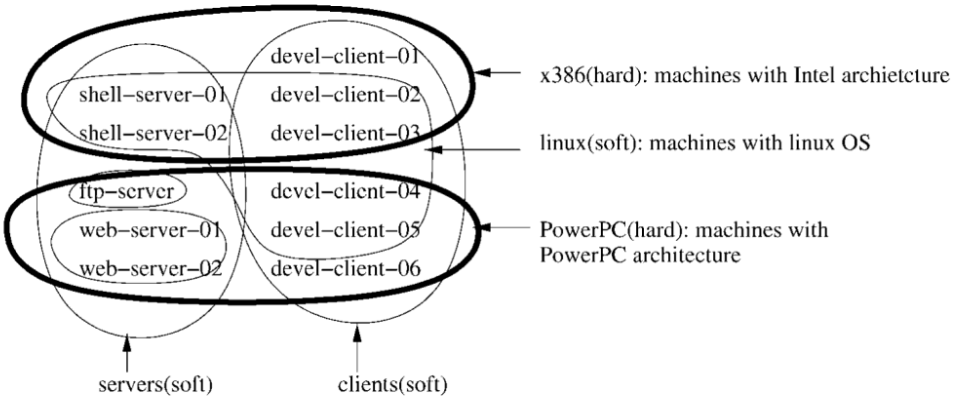


Fig. 7. Classes of machines include hard classes (bold boundaries) and soft classes (regular weight boundaries).

[4]This is the basis of 'client pull' configuration management, described in Section 13.1.

function: some machines are clients, while others are servers; among servers, there are different kinds of services provided.

Note that the concepts of 'hard' and 'soft' depend upon the capabilities of the tools being used and a particular machine's state within a configuration process. During system bootstrapping from bare metal, the operating system of the machine is a 'soft' class. If one is maintaining the configuration of a live system, the same class becomes 'hard', because the live maintenance process has no capacity to change the operating system itself without bringing the system down.

7.2. *Services*

Classes provide a way to describe the function of machines in isolation: two machines should exhibit common features or behavior if they are members of the same class. To describe how machines interoperate at the network level we use the concept of 'services'.

In informal terms, a *service* is a pattern of interaction between two hosts, a *client* requesting the service and a *server* that provides it. We say that the client *binds* to the server for particular services.

Traditionally, a *service* has been defined as a behavior provided by a single server for the benefit of perhaps multiple clients. In modern networks, this is too limiting a definition, and it is better to define a service as a set of tasks undertaken by a distinguished *set* of hosts for some client consumer.

Common services include:

1. Dynamic Host Configuration Protocol: MAC address to IP address mapping.
2. Domain Name Service: hostname to IP address mapping.
3. Directory Service: user identity.
4. File service: persistent remote storage for files.
5. Mail service: access to (IMAP, POP) and delivery of (SMTP) electronic mail.
6. Backup service: the ability to make offline copies of local disks.
7. Print service: the ability to make hard copies of files.
8. Login service: the ability to get shell access.
9. Window (or 'remote desktop') service: the ability to utilize a remote windowing environment such as X11 or Windows XP.
10. Web service: availability of web pages (HTTP, HTTPS),

etc.

A 'service architecture' is a map of where services are provided and consumed in an enterprise network (Figure 8). In typical practice, this architecture is labeled with names of specific hosts providing specific services.

8. Low-level configuration concepts

Our next step is to develop a model of the configuration itself. We describe 'what' system configuration should be without discussing 'how' that configuration will be assured or managed. This is a *declarative* model of system configuration, in the same way that a

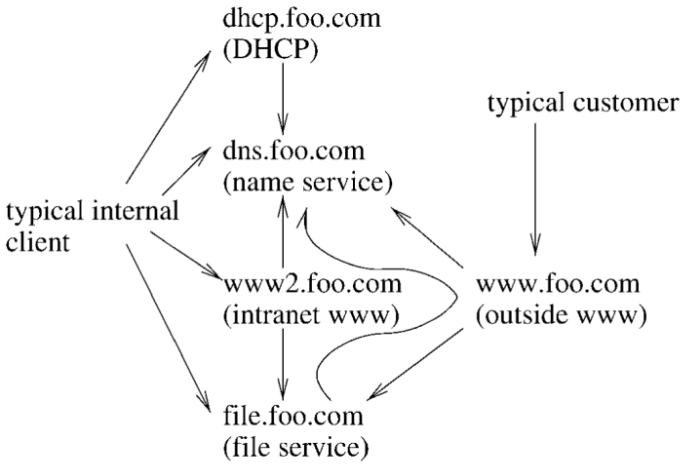


Fig. 8. A service architecture includes bindings between hosts and their servers. In the diagram $A \rightarrow B$ means that 'A is a client of B' or 'B serves A'.

'declarative language' in the theory of programming languages specifies 'what' goals to accomplish but not 'how' to accomplish them.

8.1. *Components*

A *component* of a computer network is any device that can be considered as an entity separate from other entities in the network. Likewise, a *component* of an individual computing system is a subsystem within the system that is in some sense independent from other subsystems. Hardware components are devices that may or may not be present, while software components are programs, libraries, drivers, kernel modules, or other code that are optional and/or independently controlled.

8.2. *Parameters*

Likewise, each *component* of a system may have *configuration parameters* that must be set in order for it to operate properly. Parameters can take many forms, from complex hierarchical files describing desired behaviors (e.g., the Apache web server's `httpd.conf`), to simple key/value pairs (such as kernel parameters). A parameter value describes or limits the *behavior* of a component.

8.3. *Configuration*

A configuration of a system usually has two parts:

1. A set of (hardware and software) components that should be present.

2. A collection of parameter values describing appropriate behavior for each component.

The way that components are installed, and the way their configuration is specified, varies from component to component. There is in all cases some form of persistent storage containing the parameter values for each component, such as a file on disk (though certain configuration parameters may be stored in unusual places, including flash memory).

In many cases, parameter values for a single component are spread out among several files within a filesystem. For example, in implementing an FTP service, we might specify:

(1) the root directory for the FTP service;
(2) the security/access policy for the FTP service;
(3) the list of users allowed to access the FTP service

in *different* files. Files that specify parameters for components are called *configuration files* and are considered to be *part of system configuration*.

8.4. *Behavior*

The configuration management problem is made more difficult by the lack of any efficient mechanism for mapping from low-level choices to the system behaviors that results from these choices. Usually, the behaviors that arise from configuration choices are 'documented in the manual'. A manual is a map from behaviors to choices, and not the reverse. This means that in practice, the meaning of a particular choice may be virtually impossible to reverse-engineer from the list of choices.

8.5. *Constraints*

Configuration management is really a 'constraint satisfaction problem'. The input is a set of constraints that arise from two sources:

1. The *hard constraints* that specify which configurations will work properly. Failing to satisfy a hard constraint leads to a network that does not behave properly.
2. The *soft constraints* (e.g., policies) that specify desirable options among the space of all possible options. Failing to satisfy a soft constraint does not meet the needs of users or enterprise.

The problem of configuration management is to choose one configuration that satisfies both hard and soft constraints.

Constraints arise and can be described in several ways, including considering required consistencies, dependencies, conflicts and aspects of the network to be configured.

8.6. *Consistency*

First, there are many *consistency* requirements for parameters, both within one machine and within a group of cooperating machines. Parameters of cooperating components must agree in value in order for the components to cooperate. As a typical example, machines

wishing to share a file service must agree upon a server that indeed contains the files they wish to share. But many simpler forms of coordination are required, e.g., a directory declared as containing web content must exist and contain web content, and must be protected so that the web server can access that content. But there are many other ways to express the same constraints.

8.7. *Dependencies*

A *dependency* refers to a situation in which one parameter setting or component depends upon the existence or prior installation of another component or a particular parameter setting in that component [33,74]. A program can depend upon the prior existence of another program, a library, or even a kernel module. For example, many internet services cannot be provided unless an internet service daemon `inetd` or `xinetd` is present.

8.8. *Conflicts*

It is sometimes impossible to select parameter settings that make a set of components work properly together, because components can make conflicting requirements upon parameters and/or other components. The classic example is that of two components that require different versions of a third in order to function properly. For example, two programs might require different versions of the same dynamic library; one might only work properly if the old version is installed, and the other might only work properly if a newer version is instead installed. We say in that case that there is a *conflict* between the two sets of component requirements. Equivalently, we could say that the two sets of requirements are *inconsistent*.

8.9. *Aspects*

Aspects provide a conceptual framework with which to understand and critically discuss configuration management tools and strategies [3,17]. One key problem in configuration management is that there are groups of parameters that only make sense when set together, as a unit, so that there is agreement between the parts. We informally call such groups *aspects*, in line with the current trends in 'aspect-oriented programming'.

Informally, an aspect is a set of parameters, together with a set of constraints that define the set of reasonable values for those parameters. The *value of an aspect* is a set of values, one per parameter in the aspect, where the individual parameter values conform to the constraints of the aspect.

The notion of aspects here is somewhat different than the notion of aspects in 'aspect-oriented programming' but there are some similarities. In aspect-oriented programming, an aspect is a snippet of code that changes the function of an existing procedure, perhaps by modifying its arguments and when it is invoked. It is most commonly utilized to describe side-effects of setting a variable to a particular value, or side-effects of reading that value. Here, an aspect is a set of parameters whose values must conform to some constraints.

8.10. *Local and distributed aspects*

There are many kinds of aspects. For example, in configuring Domain Name Service and Dynamic Host Configuration Protocol, the address of a hostname in DHCP should correspond with the same address in DNS. It makes no sense to set this address in one service and not another, so the host identity information (name, IP address, MAC address) form a single aspect even though the (name, IP address) pair is encoded in DNS and the (IP address, MAC address) pair is encoded in DHCP. Further, it is possible that the DNS and DHCP servers execute on different hosts, making this a *distributed aspect* that affects more than one machine at a time.

By contrast, a *local aspect* is a set of parameters that must agree in value for a single machine. For example, the host name of a specific host appears in multiple files in `/etc`, so that the parameters whose value should reflect that host name form a single aspect.

There are many cases in which configuration information must be replicated. For example, in the case of an apache web server, there are many places where parameters must agree in value (Figure 9). Each required agreement is an aspect.

Another example of a distributed aspect is that in order for web service to work properly, there must be a record for each virtual server in DNS. The parameter representing each virtual name in the server configuration file must appear in DNS and map to the server's address in DNS.

Another example of a distributed aspect is that for clients to utilize a file server, the clients must be configured to mount the appropriate directories. For the set of all clients and the server, the identity of the directory to be mounted is an aspect of the network; it makes no sense to configure a server without clients or a client without a corresponding server. The constraint in this aspect is that one choice – that of the identity of a server –

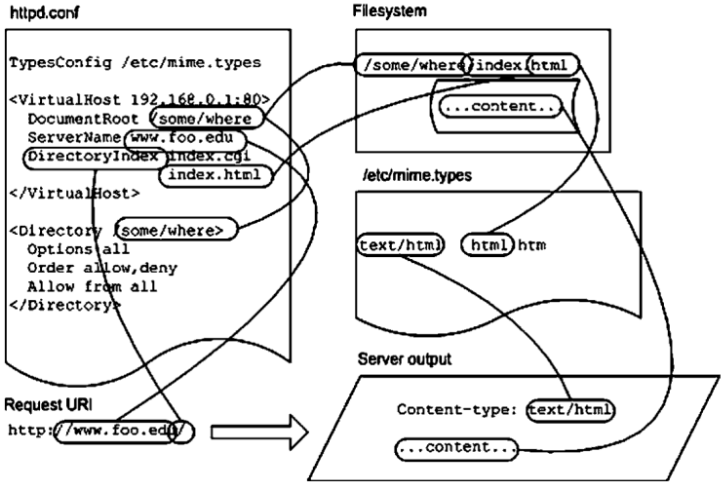


Fig. 9. Aspects of an apache web server include many local configuration parameters whose values must be coordinated. Reprinted from [68] with permission of the authors.

determines the choices for where clients will receive the service. A *binding* is a simple kind of aspect relationship between a client and a server, in which a single client takes service from a server known to provide that service.

8.11. *Aspect consistency*

Aspects can overlap and set differing constraints on the same parameters. There might be an aspect of web service that says that we need at least 2 GB of space in the web root directory, and another aspect that says the web root directory must be on a local disk. The complete configuration must satisfy the constraints of both aspects. As another example, the individual aspects for each host DHCP+DNS record overlap with any aspect determining the whole configuration of the DHCP or DNS server.

A set of several (potentially overlapping) aspects is *consistent* if there are no inherent conflicts between the constraints of the aspects. A set of (potentially overlapping) aspects is *inconsistent* if not. A set of inconsistent aspects represents a system that is not likely to function properly.

As another example, a particular web server could be described by specifying several aspects of the service:

(1) where files for each site to be served are stored;
(2) how much space must be available in that directory;
(3) the URL that is used to access each site;
(4) the way that users are authorized to view site materials,

etc.

Aspects need not be limited to a single machine, but can be properties of an entire network. The *scope* of an aspect is the number of components to which it applies. For example,

- the identity of the mail server,
- the identity of the server containing home directories,
- the identity of the gateway to the internet

are all aspects whose scope might be that of an entire network.

Anderson [3] refers to aspects as units of configuration managed by different administrators. This is very similar to our definition, because the administrators are typically responsible for different service layers requiring coordination of the kind we describe. We utilize a lower-level, mathematical definition for the purpose of separating the problem of configuration from that of mapping humans to configuration tasks. Our aspects exist as mathematical boundaries, without reference to human administrative structure. Thus,

PRINCIPLE 1. A configuration is a consistent composition of aspects.

This view of configuration management, as compositional in nature, motivates the chapter by Sun and Couch on the complexity of configuration management.

9. An operational model of configuration management

So far, we have considered configuration management as an abstract process of declaring specific state for a set of machines. In practice, however, this is not enough; we must also consider how to implement and manage that state on an on-going basis. Burgess formalized this discussion by introducing the notion of configuration operators with *Cfengine*, whose behavior can then be discussed in terms of certain properties [10,12,14], based on whether a system is in a known state or an unknown state. An operation would then bring the systems into conformance with specified requirements or a desired final state.

9.1. *Baselining*

The first step in an operational model of configuration management is to define a *baseline state* of each machine to which changes will be applied (see Figure 5). This might be a machine state, a state created by a differing management method, or a 'clean' system state created by a fresh install of the operating system onto a blank or erased disk. The choice of baseline state is one of many factors that influences which configuration management strategies may be least costly.

9.2. *Pre-conditions and post-conditions*

Operations utilized to configure a system are often dependent upon its current state; their outcome thus depends upon the order in which they are performed. This is particularly true during bootstrapping scripts, in which there is often only one order in which to complete the bootstrapping steps. Technically, we say that each change to a system has *pre-conditions* that must be true before the change can be accomplished properly, and *post-conditions* that will be true after the change, if the pre-conditions were true before the change.

PRINCIPLE 2. Every configuration operation can be associated with a set of pre-conditions that must be true for proper function, and a set of post-conditions that will be true after the operation if pre-conditions are met beforehand.

Preconditions and post-conditions of operations are often ignored but have an overall effect upon management. For example, many operations must be applied to previously configured systems; and almost all configuration management scripts presume that the system being configured has already been built by some other means.

Alternatively, we say that each operation *depends* upon the execution of its predecessors for success. For example, one must install and configure the network card driver and configuration before software can be downloaded from the network.

There are two kinds of dependencies: known and hidden (or latent). A known dependency is something that is documented as being required. A hidden dependency or 'latent precondition' [35,48] is a requirement not known to the system administrator. Many configuration changes can create latent pre-conditions. A typical example would be to set

security limits on a service and forget that these were set. As long as the service is never turned on, the latent precondition of security policy is never observed. But if the service is later turned on, mysteriously, it will obey the hidden security policy even though the system administrator might have no intent of limiting the service in that way.

Looking more carefully at this example, we see that there is another very helpful way to look at latent pre-conditions. Rather than simply being hidden state, they involve asserting behaviors that were not known to be asserted. Accordingly,

PRINCIPLE 3. Every latent precondition is a result of violating a (perhaps forgotten) aspect constraint.

In other words, since an aspect is (by definition) the functional unit of behavior, producing unexpected behavior is (by definition) equivalent with violating the atomicity of an aspect.

9.3. *Properties of configuration operations*

Suppose that we define a configuration operation to be a program that, when executed on a system or on the network, effects changes in configuration. There is some controversy amongst designers about how one should choose and define configuration operations.

One school of thought, introduced in the tool *Cfengine* [9–11,18], is that 'all operations should be declarative', which is to say that every operation should assert a particular system state or states, and leave systems alone if those states are already present. The more traditional school of thought claims that 'imperative operations are necessary' in order to deal with legacy code and systems, such as software that is installed and configured through use of `make` and similar tools.

The solution to this quandary is to look carefully at the definitions of what it means to be declarative or imperative. A set of declarative operations is simply a set in which the order of operation application does not matter [68]. Likewise, an imperative set of operations may have a different effect for each chosen order in which operations are applied. A set of operations where each one operates on a distinct subset parameters is trivially declarative, as the order of changing parameters does not affect the total result. Likewise, a set of operations each of which assigns a different value to one shared parameter is trivially imperative.

The impact of certain properties of configuration operations upon configuration management has been studied in some detail in [34]. Properties of configuration operations include:

1. *Idempotence:* an operation p is *idempotent* if its repetition has the same effect as doing the operation once. If p is an idempotent operation, then doing pp in sequence has the same result as doing p.
2. *Sequence idempotence:* a set of operations P is *sequence idempotent* if for any sequence Q taken from P, applying the *sequence* Q twice (QQ) has the same effect as applying it once. I.e., if one has a sequence of operations pqr, then the effect of $pqrpqr$ is the same as the effect of pqr applied once. This is a generalization of individual operation idempotence.

9.5. *Configuration operators*

The above arguments show that aspect consistency is an important property of configuration operations. There has been much study of how to make operations consistent, which has led to a theory of configuration operators. There is an important distinction to be made here:

- *Configuration operations* include any and all means of effecting changes in configuration parameters.
- *Configuration operators* (in the sense that *Cfengine* attempts to approximate) are operations that are fixed-point convergent in a strict mathematical sense, as defined below.

An alternative and more general view of convergence and configuration management turns the whole concept of classical configuration management upside down [12]. We view the system to be configured as a dynamical system, subject to the influences of multiple external operators that determine configuration. We specify configuration by specifying *operators to apply* under various circumstances, and cease to consider any parameters at all, except in the context of defining operators.

DEFINITION 1. A set of operators $\mathcal{P}$ is *fixed-point convergent* if there is a set of fixed points of all operators that are achieved when the operators are applied in random order with repeats, regardless of any outside influences that might be affecting the system. These fixed points are *equilibrium points* of the dynamical system comprising hosts and operators.

Whereas in the classical formulation, a policy is translated into 'parameter settings', in the dynamical systems characterization, a policy is translated into and embodied as a *set of operators to apply under various conditions*. We employ the equivalence between operation structure and parameters characterized in Section 9.3 *in reverse*, and substitute operators for parameters in designing a configuration management strategy.

The differences between classical configuration management and the dynamical systems characterization are profound. While most configuration management strategies assume that the configuration management problem lives in a closed world devoid of outside influences, the dynamical systems theory of configuration management views the system as open to multiple unpredictable influences, including 'configuration operators' that attempt to control the system and other operators and influences such as hacking, misbehaving users, etc. While the goal of traditional configuration management is to manage the configuration as a relatively static entity, the dynamical systems characterization views the configuration of each target system as dynamic, ever-changing, and open to unpredictable outside influences.

Traditional configuration management considers policy as a way of *setting parameters*, and the dynamical systems theory views the role of policy as *specifying bindings between events and operators to apply*. While the goal of traditional configuration management is deterministic *stability*, in which the configuration does not change except through changes in policy, the goal of dynamical system configuration management is *equilibrium*; a state in

which the system is approximately behaving in desirable ways, with counter-measures to each force that might affect its behavior adversely. This is the key to computer immunology [10,14].

The ramifications of this idea are subtle and far-reaching. If one constitutes an appropriate set of (non-conflicting) operators as a policy, and applies them in any order and schedule whatsoever, the result is still equilibrium [14,66]. This has been exploited in several ways to create immunological configuration management systems that implement self-management functions without the feedback loops found in typical autonomic computing implementations. Because these solutions are feedback-free and self-equilibrating *without* a need for centralized planning, they are typically easier to implement than feedback-based self-management solutions, and inherently more scalable to large networks.

There are several applications of this theory already, with many potential applications as yet unexplored. By expressing the problem of managing disk space as a two-player game between users and administrator, one can solve the problem through random application of convergent 'tidying' operators that clean up after users who attempt to utilize more than their share of disk space [11]. By defining operators that react to abnormal conditions, a system can be 'nudged' toward a reasonable state in a statespace diagram. These applications go far beyond what is traditionally considered configuration management, and it is possible to model the entire task of system administration, including backup and recovery, via dynamical systems and operators [19].

9.6. *Fixed points of configuration operators*

The base result for configuration operators concerns the relationship between configuration operations and constraints. A common misconception is that we must predefine constraints before we utilize configuration management. If we consider the configuration operations as *operators upon a space*, we can formulate convergent operators as applying parameter changes only if parameters are different, by use of the Heaviside step function. This characterizes configuration operations as linear operators upon configuration space.

This relatively straightforward and obvious characterization has far-reaching consequences when one considers what happens when a *set* of such operators is applied to a system:

THEOREM 3. *Let $\mathcal{P}$ be a set of convergent operators represented as linear operators on parameter space, and let $\tilde{p}$ represent a randomly chosen sequence of operators from $\mathcal{P}$. As the length of $\tilde{p}$ increases, the system configured by $\tilde{p}$ approaches a set of states S in which all operators in $\mathcal{P}$ are idempotent, provided that S exists as a set of fixed points for the operators in $\mathcal{P}$.*

In other words, for any $\tilde{p}$ long enough, and for any $q \in \mathcal{P}$, $q\,\tilde{p} \equiv \tilde{p}$. Paraphrasing the theorem, provided that one selects one's operators upon configuration so that they are convergent and that there is a consistent state that is a fixed point, this state will be reached through any sufficiently long path of random applications of the operators. Thus the con-

sistent state need not be planned in advance; *it is an emergent property of the system of operations being applied*.[5]

This is our second example of an emergent property theorem. Recall that previously, we observed that the concept of a parameter emerges from the structure of operations. Here we observe that the result of a set of operators over time is a fixed point that need not be known in advance.

This result challenges the concept that the configuration itself is a static aggregate state that does not change between configuration operations. Within the operator theory, a configuration can and does change due to 'outside influences' and the only way to bound it is via application of particular operators. Security violations are seen in traditional configuration management as being somewhat outside the scope of the management process; when formulating management in terms of operators, they are simply contingencies to be repaired. In the operator theory, security violations are just another operator process that occurs asynchronously with the operator process of ongoing management. A properly designed set of operators can automatically repair security problems and one can consider security violations as just one more aspect of *typical* operation.

The purport of this theorem is subtle and far-reaching. As network components increase in complexity, the act of planning their entire function becomes impractical and one must be content with planning parts of their function and leaving other parts to the components themselves. It says that if we control only parts of a thing, and the controls on parts are not conflicting, then the thing will eventually be configured in a consistent and compliant state, even though the exact nature of that state is *not known in advance*.

In our view, the controversy over operators is a problem with the definition of the problem, and not the operators. If configuration management consists just of controlling the contents of particular files, then operators are a difficult way of granting that kind of control, but if configuration management is instead a process instead of assuring particular *behaviors*, then operators can help with that process. The operators are behavioral controls that – instead of controlling the literal configuration – control the behavior that arise from that configuration.

The belief that one must control specific files seems to arise from a lack of trust in the software base being managed [72]. If minor undocumented differences in the configuration files can express latent effects, then one must indeed manage the fine structure of configuration files in order to assure particular behaviors. In that environment, operators upon the physical configuration do not seem useful, because there are parts of configuration that must be crafted in specific and complex ways so that applications will function properly.

This is again a problem with how we define configuration management as a practice. If we define the practice as controlling the entire configuration so that errors cannot occur, we then must do that and any operators that establish only partial control are unsuitable. If, conversely, we define the practice as assuring behaviors, then operators can be crafted that assure particular behaviors by direct feedback, and whose composition converges to a consistent network state in which all behaviors are assured, regardless of exactly how that state is accomplished, or even its nature.

[5]This is a similar result to that of the Maelstrom theorem for troubleshooting, which demonstrates that the optimal order for troubleshooting steps is an emergent property of the results of the steps [29].

9.7. *Observability*

In the previous section, we discussed the intimate relationship between configuration operators and behaviors, and how the operators – by assuring behaviors – both give rise to an implicit notion of parameter, as well as an implicit notion of consistency. But what is meant by behavior?

In [35], the concept of 'observability' is used to build a model of the effects of operations, using the idea that systems that behave equivalently are effectively indistinguishable. The configuration process is represented as a state machine, where operations cause transitions between behavioral states, represented as sets of tests that succeed. The configuration operators discussed above are then simply ways to accomplish state transitions, and the fixed points of Theorem 3 are simply a set of states achieved by repeated application of the operators.

The main result of [35] is that:

THEOREM 4. *One can determine whether the configuration state machine is deterministic by static analysis of the structure of its operations.*

The meaning of this result is that the state machine is deterministic if an operation applied to one observed state always leads to another observed state, which in turn means that there is no hidden state information that is (i) not observed and (ii) utilized during the application of an operation. This is a property of configuration operations that can be statically verified. Thus the theorem can be summarized:

PRINCIPLE 6. It is possible to assure freedom from latent configuration effects by careful design of convergent operators.

In other words, a good set of operators are all that one needs to accomplish configuration management.

9.8. *Promises*

Previous sections defined the idea of configuration operators and demonstrated how those operators can be constructed. But what are appropriate configuration operators? One issue of interest is whether it is possible to manage the configuration of a network without centralized authority or control. Can a network self-organize into a functional community when no component of the network has authority over another? Recently, Burgess et al. have suggested that this is possible, through a mathematical idea known as 'promise theory' [15,16].

The basic idea of promise theory is that stations in a network are autonomous competitors for resources who only collaborate for mutual gain, like animals in a food chain. Autonomous agents in the network are free to 'promise' facts or commitments to others. Thus there is a concept of 'service' from one station to another, in the form of 'promises

kept'. This involves a kind of reciprocal trust agreement between computers who otherwise have no reason to trust one another. Preliminary results show that networks managed in this way do indeed converge to a stable state, even in the presence of harmful influences, simply because promises can be viewed as convergent operators in the sense of Theorem 3.

Many practicioners of configuration management are in opposition to the idea that computers will broker their own services; the definition of the job of configuration manager is to insure that there is no disruption of services. Politically, many people who practice configuration management do not yet accept the idea that systems could manage their own configurations and services. But as networks become larger, more heterogeneous and more complex, something like promise theory will be necessary to manage them.

9.9. *Myths of configuration management operations*

The above section lays to rest several ideas that have plagued the practice for years. One cannot sidestep the complexities of configuration management via a simple choice of language. Nor is the process of defining network state actually necessary in order to achieve appropriate behaviors. But there are other myths that also misdirect practicioners in creating effective and cost-effective configuration management strategies.

Let us consider these:

PRINCIPLE 7. Configuration management is not undecidable.

The conjecture that it is undecidable, proposed in [72], is easily proven false by a simple proof that no finite system can be undecidable, only intractable. The intractability of configuration management was partly refuted in [35], by showing that most configuration operations take a simple form whose completion is statically verifiable.

There has been some controversy about whether it is possible to 'roll back' systems from a new state into an older state. This stems partly from disagreement about what rollback means, i.e. whether it means a complete return to previous behavior. The argument has been that since it is impossible to roll back state using certain tools (including *ISConf* and *Cfengine*) that rollback is impossible in general. We assert that, as long as this refers to the managed state only, this is a limitation of the tools, not a theoretical limit:

PRINCIPLE 8. Rollback of configuration operations is in fact possible.

Any state that is not managed, like runtime or operational data, that affects the operation of a computer will, in general, prevent a computer from behaving identically after rollback.

10. Traditions of configuration management

In the above sections, we have discussed many theoretical issues that control the practice of configuration management. We now turn to the practice itself, and describe how practicioners actually accomplish the task of configuration management. Our first step is to study and analyze the traditions from which current practices arose.

11.1. *File distribution*

The difficulties inherent in writing unconstrained scripts led to several alternative approaches that attempt to accomplish the same end results as scripts, while improving maintainability of the final result. Alternatives to scripting arose opportunistically, automating parts of the operations traditionally accomplished by scripts and leaving other operations to be done by scripts at a later time.

File distribution is the practice of configuring systems solely or primarily by copying files from a master repository to the computers being configured. This has its roots in the practice of using file copying commands in scripts. File distribution in its simplest form is accomplished by a script that copies files from a locally mounted template directory into system locations, using a local file copying command. More sophisticated versions of file distribution include use of remote copying commands ('*rcp*', '*scp*', '*rsync*', etc.), which naturally evolved into file distribution subsystems such as '*rdist*' [24], and related tools [26].

Rdist is a simple file distribution system that functions by executing on a machine with administrative privilege over managed machines. An input file, usually called `Distfile`, describes files to copy, the target hosts to which to copy them, and the locations for which they are destined. The input file can also direct *rdist* to execute appropriate remote commands on the hosts to be configured, after a file is distributed.

For example, the `Distfile` entries in Figure 11 direct *rdist* to ensure that `/usr/lib/sendmail` and all header files in `/usr/include` are the same for the source host as for the target hosts `host01` and `host02`. Additionally, whenever `/usr/lib/sendmail` has to be copied, *rdist* will run a command to recompile the sendmail ruleset (`/usr/lib/sendmail-bz`).

Rdist improved greatly upon the robustness of custom scripts, and exhibits several features that have become standard in modern configuration management tools. A *file repository* contains master copies of configuration files. This is copied to a target host via a *push strategy* in which the master copy of *rdist* (running on an *rdist* master server) invokes a *local agent* to modify the file on each client host. A *class mechanism* allows one to categorize hosts into equivalent classes and target specific file content at classes of hosts. Files are not copied unless the timestamp of the target is older than the timestamp of the source. This minimizes intrusiveness of the distribution process; 'gratuitous' copying that serves no purpose is avoided. Finally, after a file is copied, 'special' *post-copying commands* can be issued, e.g., to email a notification to a system administrator or signal a daemon to re-read its configuration.

```
HOSTS = ( host01 host02 )
FILES = ( /usr/lib/sendmail /usr/include/{*.h} )
${FILES} -> ${HOSTS}
        install ;
        special /usr/lib/sendmail "/usr/lib/sendmail -bz"
```

Fig. 11. A simple Distfile instructs *rdist* to copy some files and execute one remote command if appropriate.

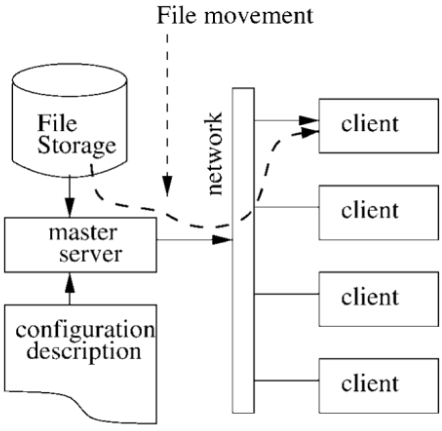


Fig. 12. A push strategy for file distribution stores all configuration information on a master server, along with files to be distributed.

Rdist implements what is often called a *server-push strategy* (also sometimes called 'master-slave') for configuration management. A push strategy has several advantages and disadvantages (Figure 12). First, all information about what is to be configured on clients must be contained on the master server in a configuration file. This means that the master server must somehow be informed of all deviations and differences that distinguish hosts on the network. This includes not only soft classes that designate behaviors, but also hard classes that indicate hardware limitations. One advantage of push strategies is that they are relatively easy to set up for small networks.

File distribution of this kind was useful on moderate-size networks but had many drawbacks on networks of large scale or high heterogeneity. First, in a highly heterogeneous network, the file repository can grow exponentially in size as new copies of configuration files must be stored; there is a *combinatorial explosion* as file repository size is an exponential function of heterogeneity. A more subtle limit of *rdist*'s push strategy is that (originally) only one host could be configured at a time; running *rdist* for a large network can take a very long time. Attempts at utilizing parallelism [24] address the problem somewhat, but utilizing push-based file distribution on large networks remains relatively time-consuming.

Server-push configuration management becomes impractical for large and complex networks for other reasons. In a *push* strategy, each host must receive content from a master host that has been configured specifically to contact the target. This requires configuring the master with a *list* of hosts to be contacted. If there are a very large number of hosts, keeping this list up to date can be expensive. For example, every new host to be managed must be 'registered' and data must be kept on its hardware configuration, etc.

File distribution used alone is impractical for large networks in two other ways:

1. Distribution to a large number of clients is a slow process, involving maintenance of client databases and inherently serial transactions.
2. The size of files that must be stored in the repository is proportional to the number of classes, including all combinations of classes actually employed to describe some host.

In other words, there is little economy that comes with scale. This has led to innovations that address each weakness.

11.2. *Client pull strategies*

The slowness of server push strategies can be eliminated via *client pull* strategies, in which clients instead contact a server regularly to check for changes. Cfengine has championed this approach [9–11,18]. In a pull strategy, each client has a list of *servers* (Figure 13). This list is smaller and easier to maintain. The clients make decisions about which servers to employ, allowing extremely heterogeneous networks to be managed easily.

Another related strategy developed earlier this year is that of *pull–push*. This is a hybrid strategy in which a centralized server first polls clients for their configurations, and then pushes a configuration to each one. This allows one to customize each client's configuration according to properties obtained by querying the client. This allows a centralized planning process to utilize data about client capabilities and current configuration. Pull–push management is a design feature of the *Puppet* toolset for configuration management [62], and also implemented to some extent in both *BCFG2* [40,41] and *LCFG* [2,4].

11.3. *Generating configuration files*

The second problem with file distribution is that the number of files to be stored increases exponentially with the heterogeneity of the network. One way of dealing with the second problem is to copy generic files to each host and then edit files on the client in order to customize them. This is particularly viable if the changes required are minor or can be accomplished by appending data to an existing master copy. This can be accomplished

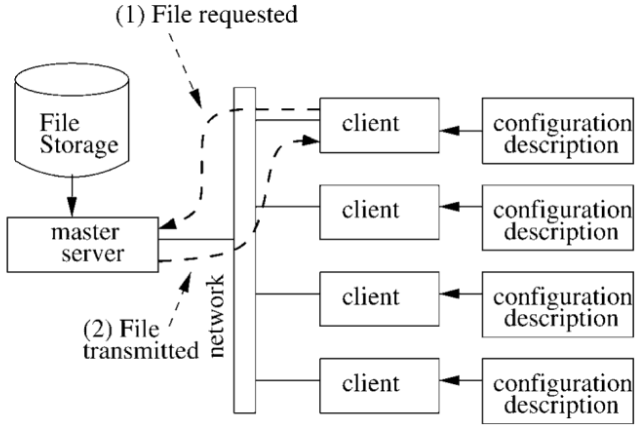


Fig. 13. A pull strategy for file distribution stores configuration information on clients, including the location of one or more master file servers.

via scripting, use of *Cfengine* [9–11,18], or post-copy scripts that are invoked during file distribution.

Another technique for avoiding an exponential explosion of files is to generate all of the configuration files for each host from a database [44,45] rather than storing each one explicitly in a repository. The database contains both requirements and hardware specifications (hard limitations) for each host. One can generate configuration files either on a centralized master distribution host or via an agent that runs on the target host. Generation scripts are easier to understand and maintain than scripts that modify pre-existing files. These rules form the core logic of many current-generation configuration management tools.

A third way to deal with the combinatorial explosion is via templating [58]. This is a compromise between generating client configuration files and editing them. In this strategy, one stores generic templates of configuration files in a file repository with some details left blank, and fills in these details from a master database or file. This strategy takes advantage of the fact that in very large configuration files, such as those for web servers, much of the configuration is fixed and need not be changed for different hosts in the network [68]. By templating the unchanging data, and highlighting changes, one limits the number of files being stored in a repository.

11.4. *Script portability management*

There remain many scripts that are not easily expressible in terms of templates, copying or generation. One common example is a script that compiles source files into binary code. The *script management problem* is to control script execution to achieve consistent results.

One form of script management evolved from mechanisms originally invented to help in maintaining portable software. When compiling a program via the *make* [59] program, there are a number of possible program parameters that change from machine to machine, but are constant on any particular machine, such as the full pathnames of specific programs, libraries, and include files. `Imake` is a program that codes these constant parameter values into a file for reusability, and generates `Makefiles` for *make* that have the values of these parameters pre-coded. Likewise, `xmkmf` is a version of `Imake` that utilizes pre-coded values for the X11 Window system, to allow building graphics applications. The idea of `Imake` and `xmkmf` is that *local dependencies can be isolated from portable code in crafting a portable application.*

It is not surprising that these tools for software configuration management have their counterparts in network configuration management. The Process-Informant Killer Tool (*PIKT*) [60] is a managed scripting environment that can be utilized for configuration management. *PIKT* scripts employ special variables to refer to system locations that may vary, e.g., the exact location of configuration files for various system functions. By recording these locations for each architecture or version of operating system, one can utilize one script on a heterogeneous network without modification. PIKT also contains a variety of programming aids, including a declarative mechanism for specifying loops that is remarkably similar to the much later XQUERY mechanism for XML databases.

11.5. *Managing script execution*

Scripts have several major drawbacks as configuration management tools: they are sensitive to their execution environments, difficult to maintain, and must be applied consistently to all hosts in a soft class. Particularly, if a sequence of scripts is to be applied to a homogeneous set of hosts, the effect can differ if the scripts are applied in different orders on each host. Consider two scripts, one of which edits a file (S1) and the other of which creates the file to be edited (S2). Unless S2 precedes S1, the result will not be what was intended. Again, one solution to this problem was first invented for use in software configuration management.

ISConf [72,73] utilizes a variant of the software compilation tool *make* [59] to manage system and software configuration at the same time. Since many software packages are delivered in source form, and many site security policies require use of source code, use of *make* is already required to compile and install them. The first version of *ISConf* utilizes *make* in a similar fashion, to sequence installation scripts into linear order and manage whether scripts have been executed on a particular host. Later versions of *ISConf* include Perl scripts that accomplish a similar function.

ISConf copes with latent effects by keeping records of which scripts have been executed on which hosts, and by insuring that every host is modified by the exact same sequence of scripts. *ISConf*'s input is a set of 'stanzas': a list of installation and configuration scripts that are to be executed in a particular order. Each stanza is an independent script in a language similar to that of the Bourne shell. A class mechanism determines which stanzas to execute on a particular host. *ISConf* uses time stamp files to remember which stanzas have been executed so far on each host. This assures that hosts that have missed a cycle of configuration due to downtime are eventually brought up to date by running the stanzas that were missed (Figure 14).

11.6. *Stateful and stateless management*

A configuration management tool is *stateful* if it maintains a concept of state on the local machine other than the configuration files themselves, and conditions its actions based upon that state. Otherwise, we call the tool *stateless*. Because it keeps timestamps of the scripts that have been executed on disk, *ISConf* supports a *stateful* configuration management strategy, as opposed to *Cfengine*, which is (in the absence of user customizations) a mainly *stateless* strategy.[6]

The stateful behavior of *ISConf* solves the problem of script pre-conditions in a very straightforward (but limited) way: each script utilizes the post-conditions of the previous one and assures the pre-conditions of the next, forming a 'pipeline' architecture. The advantage of this architecture is that – provided that one starts using *ISConf* on a cleanly installed host – scripts are (usually) only executed when their pre-conditions are present, so that repeatable results are guaranteed. There are rare exceptions to this, such as when the environment changes drastically due to external influences between invocations of stanzas.

[6] Some long term state is memorized in *Cfengine* through machine learning and associated policies, but this is not a part of the configuration operators.

A key concept in package management is that of package *dependencies*. The function of a particular package often requires installation of others as prior events. This is accomplished by keeping track of both installed packages and the dependencies between packages. To illustrate this, consider the structure of the RedHat package manager (*RPM*). Each package internally declares the attributes that it *provides* and the attributes that it *requires*, as character strings. These are *abstract attributes*; there is no correspondence between requirements and files in the package. The package manager command, with access to data on which package requires which others, is able to pre-load prerequisites so that packages are always loaded in dependency order. Correctness of dependencies is left to the package creator. While experience with packages in major operating systems is good, incorrectness of dependencies is a major problem for contributed software packages maintained by individuals [46].

The intimate relationship between package management and configuration management arises from the fact that most packages have a configuration that must be managed and updated to control the function of the software in the package. Thus it is difficult to separate delivery and updating of the software package itself from the process of configuring that software for proper function. In a way, package management is doomed to be confused with configuration management, forever, because of the overlaps between package and configuration management in both mission and method.

The author is aware of at least a few sites that currently utilize a package management mechanism to manage configuration scripts as well as software packages; each script is encapsulated in a package and each such package depends upon the previous. Then the package manager itself manages the 'installation' of the configuration scripts, and insures that each script is run once and in the appropriate order, exactly as if ISConf were managing the scripts. This is exactly the mechanism by which enterprise management is accomplished in Microsoft environments, using the MSI package format [76].

13. Declarative specification

One alternative to the complexity of scripting is to recode script actions as declarative statements about a system. We no longer execute the declarations as a script; we instead interpret these declarations and assure that they are true about the system being configured (through some unspecified mechanism). This avoids the script complexity problems mentioned in the previous sections, and leaves programming to trained programmers rather than system administrators.

13.1. *Cfengine*

The most-used tool for interpreting declarative specifications is *Cfengine* [9–11,18]. Unlike `rdist`, whose input file is interpreted as a script, the input file for *Cfengine* is a set of declarations describing desirable state. These declarations subsume all functions of `rdist` and add considerably more functions, including editing of files and complete management of common subsystems such as NFS mounts. An autonomous agent (a software

tool) running on the host to be configured implements these declarations through fixed-point convergent and 'idempotent' operations that have desirable behavioral properties.

A 'convergent' operation is one that – over time – moves the target system toward a desirable state. For a system in a desirable state, a 'convergent' operation is also 'idempotent'; it does not change anything that conforms to its model of 'system health'. Convergent and idempotent operations support an *immunological model of configuration management* [11, 14], in which repeated applications of an immunizing agent protect a system from harm. This approach is distinct from most others; most configuration management tools rely upon a single action at time of change to accomplish configuration management.

Constructing system operations that are idempotent and/or convergent is a challenging programming task – involving many branches – that is beyond the programming skills of a typical system administrator (Figure 16). A simple linear script becomes a multiple-state, complex process of checking for conditions before assuring them. Thus we can view the declarative input file for *Cfengine* as a form of information hiding; we specify 'what' is to be done while omitting a (relatively complex) concept of 'how' to accomplish that thing. The configuration manager need not deal with the complexity of 'how'. This makes the *Cfengine* input file considerably simpler in structure than a script that would accomplish the same effect. The same strategy makes the input file for the package manager *Slink* [25, 32] considerably less complex in structure than a script that accomplishes the same actions.

Unfortunately, there is a cost implication in use of this strategy. Many packages are provided to the system administrator with imperative scripts that install them, either via the

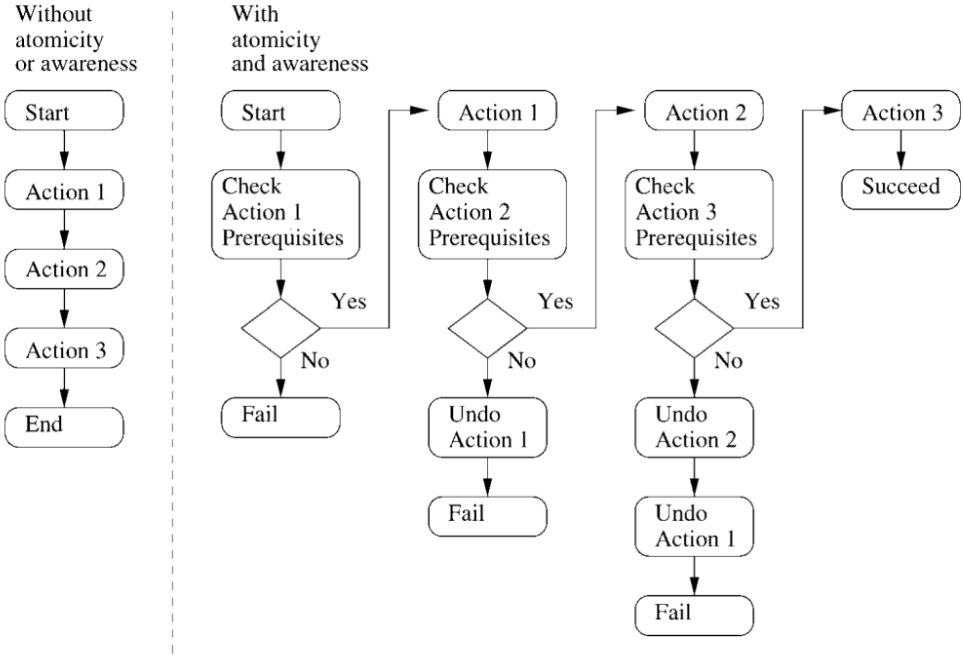


Fig. 16. Adding atomicity and awareness to a script causes greatly increased branch complexity.

make compilation control program or via a package manager post-install script written in a shell language. To make the process of configuration 'truly declarative', one must reverse-engineer the impact of these scripts and code their actions instead as declarative statements. This is a non-trivial task, and must be repeated each time a package is updated or patched. For example, many Redhat Package Manager packages contain post-installation scripts that make non-trivial changes to configuration files, that must be taken into account when installing the RPM as part of a configuration change.

Cfengine introduces several new strategies to configuration management. Unlike `rdist`, which utilizes a *server-push strategy* (master to slave) (Figure 12), *Cfengine* implements a *client-pull strategy* for file copying (Figure 13). The *Cfengine* agent runs on the host to be configured and can be invoked manually, under control of the *cron* timing daemon, or remotely. Once running, *Cfengine* can perform local edits to files and request copies of configuration files from perhaps multiple servers. Since the action of configuration is controlled from the client being configured, the server bottleneck that plagues `rdist` can be effectively load-balanced (also called 'splaying').

Cfengine employs classes to aid in building different kinds of hosts. *Cfengine*'s class variables define classes or groups of hosts differently from the way classes are defined in other configuration tools. In many of *Cfengine*'s predecessors, a class is simply a list of hosts. In *Cfengine*, a class is a *set of conditions upon a host*. This new meaning is more consistent with the meaning of the word 'class' in object-oriented programming than the former meaning. This definition, however, means that a particular instance of *Cfengine* only knows whether the host being configured is a member of each class, and cannot infer the identities of other members of a class from the class description.

Each class variable is a boolean value; either true or false. 'Hard classes' describe hardware and other system invariants, e.g., the operating system. For example, the hard class 'linux' is instantiated if linux is the host operating system. 'Soft classes' describe the desired state for the current configuration. These can be created by constructing simple scripts that probe system state. Each action in *Cfengine* is only applicable to a particular system state, which is expressed as a boolean expression of (both hard and soft) class variables.

A typical Cfengine declaration includes a stanza, guard clause, and action. For example, the stanza in Figure 17, copies the file `cf.server.edu:/repo/etc/fstab` into `/etc/fstab` on every host that is in the hard class `linux` and *not* (!) in the soft class `server`. The stanza identifier is `copy`, which determines what to do. The guard clause is `linux.!server`, which says to perform the following actions for linux machines that are not servers. The next line describes what to do. Note that this is a declarative, not an imperative, specification; the English version of this statement is that `/etc/fstab` *should* be a copy of `cf.server.edu:/repo/etc/fstab`; this is not interpreted as a command, but as a condition or state to assure.

```
copy:
    linux.!server::
        /repo/etc/fstab dest=/etc/fstab server=cf.server.edu
```

Fig. 17. A Cfengine copy stanza includes stanza identifier, guard clause and action clauses.

There is a striking similarity between *Cfengine*'s configuration and logic programming languages [30]. Each *Cfengine* class variable is analogous to a Prolog fact. Each declaration is analogous to a Prolog goal. This makes a link between the meaning of 'declarative' in programming languages and the meaning of 'declarative' in system administration: both mean roughly the same thing.

Another new idea in *Cfengine* is that of 'immunization' of an existing system to keep it healthy [11,14]. Most configuration management systems are executed only when changes are required. By contrast, *Cfengine* can be set to monitor a system for changes and correct configuration files as needed.

Cfengine has several other unique properties as a configuration management aid. Software subsystems, such as networking, remotely accessible disk, etc., are manageable through convergent declarations. *Cfengine* introduces the possibility of randomization of configuration actions. An action can be scheduled to run some of the time, or with a certain probability. This enables game-theoretic resource management strategies for limiting filesystem use without quotas [11,66]. Finally, it is also possible to set up *Cfengine* to react to changes in measured performance, so that it makes changes in configuration in accordance with system load and/or deviation from normal load conditions.

Cfengine breaks down the boundaries between configuration management, resource management, user management, and monitoring. *Cfengine* directives can clean up user home directories, kill processes, and react to abnormal load conditions. The traditional meaning of configuration management as acting on only system files is gone; all aspects of the filesystem and process space are managed via one mechanism.

13.2. *File editing*

One perhaps controversial aspect of *Cfengine* is the ability to edit configuration files to contain new contents. Users of *Cfengine* find this facility indispensable for dealing with the 'combinatorial explosion' that results from file copying. Instead of simply copying a file, one can edit the file in place. Editing operations include the ability to add lines if not present, and to comment out or delete lines of the file in a line-oriented fashion.

What makes file editing controversial is that – although undoubtedly useful, and indeed very widely used – file editing is particularly vulnerable to pre-existing conditions within a file. For example, suppose one is trying to edit a configuration file where duplicate instances are ignored, e.g., `/etc/services`, and the line for the *ssh* service in the existing file differs slightly from the established norm. Then using the `AppendIfNotPresent` operation of *Cfengine* to append a line for *ssh* will result in having *two* lines for *ssh* in the file; the prior line will be treated as *different* even though it describes the *same service*. Depending upon how this file is interpreted by the resolver, either the first line or the second line will be used, but not both. In other words, file editing must be utilized with great care to avoid creating latent conditions that can manifest later as behavior problems. One solution to the file-editing quandary is to provide operators that treat files as database relations with local and global consistency constraints [31]. Convergent operators for such 'higher-level languages' are as yet an unsolved problem, which *Cfengine* only approximates.

One effective strategy for dealing with editing pre-conditions is to copy in a 'baseline' version of the file before any edits occur. This assures that edits will be performed on a file with previously known and predictable contents. If this is done, file editing can be as reliable as generating files from databases. Unfortunately, there is a downside to this strategy when used within *Cfengine*; the file being edited goes through two 'gratuitous' rewrites every time the tool is invoked to check the configuration. This can lead to anomalous behavior if a user is trying to use a program that consults the file at the time the file is being updated.

13.3. *Change control*

Disciplined use of *Cfengine* requires a different regimen of practice that of *ISConf*, but for the exact same reasons [35]. One cannot ever be sure that *all* machines in a network are in complete compliance with a *Cfengine* configuration, any more than one can be sure that all machines are synchronized with respect to *ISConf*. This lack of confidence that *all* machines have been synchronized leads to somewhat non-intuitive requirements for edits to *Cfengine* configuration files.

The fundamental requirement for distributed uniformity is the *principle of management continuity:* once a file is 'managed' (copied, edited, etc.) by *Cfengine*, it must continue to be managed by *Cfengine* in some fashion, for as long as it remains constrained by policy. It is a normal condition for some hosts to be down, physically absent (e.g., laptops), or just powered off. Thus there is no way that one can be sure that one has 'reverted' a file to an unmanaged (or perhaps one should say 'pre-managed') state. Although not unique to *Cfengine*, this is part of the *Cfengine* ethos of 'living with uncertainty' [12].

14. Cloning

Another configuration management strategy arises from the tradition of file copying. Often, in creating a system that should perform identically to another, it is easier to copy *all* system files than to modify just the configuration files. This is called *cloning a system*. Then a manager (or a script, or a tool) can change only the files that have customized contents on each host, e.g., the files representing the identity of the host as a network entity.

Simple forms of cloning include *Norton Ghost*, which simply copies a master version of a system onto a client and allows simple one-time customizations via scripting. After the cloning and customization operations, there is no capability for further control or customization. To make changes, one must change the master device and/or customization script and then reclone each client.

Cloning can be expensive in time and potential downtime. To make a clone of an isolated system, all system information must flow over the network, and this flow generally occurs at a substantively lower rate than when compressed files are copied from disk and uncompressed, depending on network conditions.

Advanced cloning approaches are intimately related to monitoring technologies. Radmind [38] and its predecessors (including Synctree [54]) take an immunological approach

15.3. *Common information models*

File copying, generation, or editing are bulky procedures compared to the simple process of setting a configuration parameter. To ease the process of setting parameters on network devices, the Simple Network Management Protocol (SNMP) [57] allows individual parameters to be set by name or number. The Common Information Model (CIM) [42] is an attempt to generalize SNMP-like management to computing systems. The basic configuration of a computing system is encoded as parameters in a management information base (MIB), which can be changed remotely by SNMP. Much of the configuration and behavior of a host can be managed by utilizing this interface; it encompasses most system parameters but fails to address some complexities of software and package management. This approach is implemented in several commercial products for enterprise configuration management.

16. Simplifying configuration management

It comes as no surprise that configuration management is difficult and even – in some sense – computationally intractable. One of the major outgrowths of theoretical reasoning about configuration management is an understanding of why configuration management is difficult and steps that one can take to make it a simpler and less costly process [69]. The system administrator survives configuration management by making the problem easier whenever possible. We limit the problem domain by limiting reachable states of the systems being configured. The way in which we do this varies somewhat, but the aim is always simplicity:

1. By using a particular sequence of operations to construct each host, and not varying the order of the sequence. E.g., script management systems such as *ISConf* enforce this.
2. By exploiting convergence to express complex changes as simple states.
3. By exploiting orthogonality to break the configuration problem up into independent parts with uncoupled behavior and smaller state spaces.
4. By exploiting statelessness and idempotence to prune undesired states from the statespace of an otherwise unconstrained host.
5. By constructing orthogonal subsystems through use of service separation and virtuality.
6. By utilizing standards to enforce homogeneity.

Note that most of these are outside the scope of what is traditionally called ‘configuration management’ and instead are part of what one would call ‘infrastructure architecture’ [71]. But in these methods, we find some compelling solutions to common configuration management problems.

16.1. *Limiting assumable states*

All approaches to configuration attempt to limit the states a system can achieve. Script management enforces a script ordering that limits the resulting state of the system. Con-

vergent management attempts to reduce non-conforming states to conforming ones, hence reducing the size of the statespace. In both cases, the goal is to make undesirable states unreachable or unachievable.

A not-so-obvious way of limiting assumable states is to promptly retire legacy hardware so that it does not remain in service concurrently with newer hardware. This limits states by limiting the overall heterogeneity of the network, which is a help in reducing the cost of management and troubleshooting.

16.2. *Exploiting orthogonality with closures*

Another simple way configuration management is made tractable is by viewing the system to be configured as a composition of orthogonal and independent subsystems. If we can arrange things so that subsystem configurations are independent, we can forget about potential conflicts between systems.

A closure [31] is an independent subsystem of an otherwise complex and interdependent system. It is a domain of 'semantic predictability'; its configuration describes precisely what it will do and is free of couplings or side effects. Identifying and utilizing closures is the key to reducing the complexity of managing systems; the configuration of the closure can be treated as independent of that of other subsystems. Systems composed of closures are inherently easier to configure properly than those containing complex interdependencies between components.

A closure is not so much created, as much as it is discovered and documented. Closures already exist in every network. For example, an independent DHCP server represents a closure; its configuration is independent of that of any other network element. Likewise, appliances such as network fileservers form kinds of closures. Building new high-level closures, however, is quite complex [68], and requires a bottom-up architecture of independent subsystems to work well.

16.3. *Service isolation*

One way of achieving orthogonality is through service isolation. *Service isolation* refers to the practice of running each network service on a dedicated host that supports no other services. There are several advantages to service isolation:

1. Each service runs in a completely private and customized environment.
2. There is no chance of application dependencies interfering with execution, because there is only one application and set of dependencies.

Thus configuration management is greatly simplified, compared to configuring a host that must provide multiple services. Thus service isolation simplifies configuration by avoiding potential dependency conflicts, but costs more than running a single server.

16.4. *Virtualization*

One can reduce the costs of service isolation by use of *virtualization*. *Virtualization* is the process of allowing one physical host to emulate many virtual hosts with independent configurations. There are several forms of virtualization. One may virtualize the function of a whole operating system environment to form an 'information appliance' [67] that functions independently from all other servers and is free of service conflicts. One may instead virtualize some subpart of the execution environment. For example, the PDS system [1] virtualizes the loading of dynamic libraries by applications, *without* virtualizing the operating system itself. The result is that each program runs in a custom operating environment in which it obtains access only to the library versions that it desires, and no version conflicts are possible *in one application's execution environment.*

16.5. *Standardization of environment*

A final practice with some effect upon configuration management is that of standardization. The Linux Standard Base (LSB) [53] attempts to standardize the *locations* of common configuration files, as well as the link order and contents of system libraries. Applications expecting this standard are more likely to run on systems conforming to the standard.

The standard consists of three parts:

1. A specification of the contents of specific files and libraries.
2. An *environment validator* that checks files for compliance.
3. A *binary application validator* that checks that applications load proper libraries and make system calls appropriately.

LSB is based upon a logic of 'transitive validation' (Figure 18). If an environment $E1$ passes the environment validator, and an application A passes the binary validator, and A functions correctly in $E1$, and another environment $E2$ passes the environment validator, then A should function correctly in $E2$. This means that if LSB functions as designing, testing an application once suffices to guarantee its function in all validated environments.

Alas, LSB falls short of perfection for a few theoretical reasons. First, there is no way of absolutely being sure that binary code conforms to system call conventions (especially when subroutine arguments are computed rather than literal). This is equivalent to the intractable problem of assuring program correctness in general. Thus there is no way to address this problem.

A second issue with LSB is that the environment validator only validates the 'standard' environment, and does not take into account changes that might be made at runtime, e.g., adding a dynamic library to the library path. If this is done, the environment validation may become invalid without the knowledge of the LSB validator.

However, the important point to take from this is that *standards reduce system complexity, and thus reduce potential conflicts and other issues in configuration management.* Thus standardization is not to be ignored as a major cost-saving factor in configuration management.

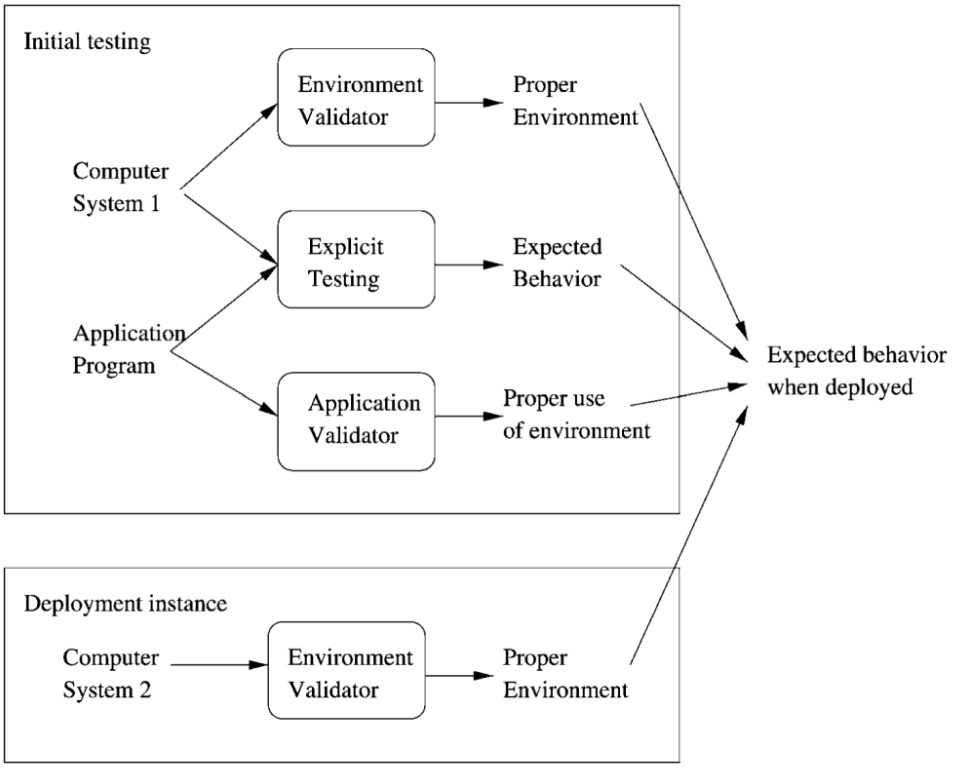


Fig. 18. Standardization and validation lead to claims of correct program behavior in environments in which explicit testing has not occurred.

17. Critique

In this chapter, we have outlined many different techniques, practices, and methods for solving the configuration management problem. In these practices, there emerge some central trends and patterns from which one can characterize the field.

First, the goal of current configuration management strategies is not simply to provide appropriate network behavior, but to minimize the cost of doing so. There are several factors that can minimize costs, including:

1. Limiting hardware heterogeneity:
   (a) By promptly retiring legacy systems.
   (b) By purchasing large numbers of identical computers.
2. Limiting gratuitous complexity in configurations:
   (a) By negotiating and implementing highly homogeneous policies for use of computers.
   (b) By opposing policies that contribute to higher cost without added value.
3. Limiting the possible configuration states of each computer:
   (a) By always starting from a blank disk and replaying the same scripts in order.

   (b) By generating consistent configuration files from databases.
   (c) By use of idempotent and stateless operations to reduce potential states of each target machine.
4. Splitting statespaces into orthogonal subspaces:
   (a) By identifying behaviorally independent subsystems and closures.
   (b) By synthesizing independence through service independence and virtualization.
5. Pre-validation of results of deployment:
   (a) By validating results for each target platform.
   (b) By combining environment and application standards and validators to exploit transitive validation properties of software.

This list of strategies is a fairly good summary of the state of the art. Note that in the list of strategies, the choice of tool for configuration management is not even mentioned. It is instead a side-effect of adopting best practices, which can be summed up in one imperative: *minimize complexity*.

We have thus come full circle. We started by discussing tools and how they work, as part of a configuration management strategy. We are now at the level where the tools are no longer important, and are replaced by the goals that they accomplish. These are goals of practice, not tool choices, and many of them require human interaction and decision making to reduce cost and maximize value. The true art of configuration management is in making the strategic decisions that define mission in a way that reduces cost. For example, one administrator recently computed a savings of roughly $100,000 by replacing all desktop computers in a hospital with linux thin clients. This savings had nothing to do at all with the tools used to manage the network. The complexity of the network itself was reduced, leading to a reduction in cost *by design* rather than *by practice*, i.e. when one manages the whole enterprise as a single community of cooperating humans and computers [13].

18. Current challenges

We have seen above that the practice of configuration management arises from two traditions: high-level policy definition and low-level scripting. At the present time, the two traditions continue as relatively separate practices with little overlap. A majority of the world uses some form of scripting, including convergent scripting in *Cfengine*, to accomplish configuration management. Very few sites have adopted overarching tools that generate configuration from high-level policies, and in a recent informal poll of configuration management users, most users of high-level ('generative') tools wrote their own instead of employing an existing tool.

In our view, the principal challenge of configuration management is that the cost of management is only very indirectly related to its value [36]. The cost of high-level strategies seems higher, and their value cannot be quantified. Meanwhile, the average practicioner has limited ability to justify approaches that cost significant startup time but have unquantifiable benefits. For example, it takes significant effort to re-engineer software packages so that their installation is declarative rather than imperative. Thus, one main challenge of

[31] A. Couch, J. Hart, E. Idhaw and D. Kallas, *Seeking closure in an open world: A behavioral agent approach to configuration management*, Proc. LISA-XVII, USENIX Association, San Diego, CA (2003).
[32] A. Couch and G. Owen, *Managing large software repositories with SLINK*, Proc. SANS-95 (1995).
[33] A. Couch and Y. Sun, *Global impact analysis of dynamic library dependencies*, Proc. LISA-XV, USENIX Association, San Diego, CA (2001).
[34] A. Couch and Y. Sun, *On the algebraic structure of convergence*, Proc. DSOM'03, Elsevier, Heidelberg, DE (2003).
[35] A. Couch and Y. Sun, *On observed reproducibility in network configuration management*, Science of Computer Programming, Special Issue on Network and System Administration, Elsevier, Inc. (2004).
[36] A. Couch, N. Wu and H. Susanto, *Toward a cost model for system administration*, Proc. LISA-XVIII, USENIX Association, San Diego, CA (2005).
[37] Cowan et al., *Timing the application of security patches for optimal uptime*, Proc. LISA 2002, USENIX Association (2002).
[38] W. Craig and P.M. McNeal, *Radmind: The integration of filesystem integrity checking with filesystem management*, Proc. LISA-XVII, USENIX Association (2003), 1–6.
[39] I. Crnkovic and M. Larsson, eds, *Building Reliable Component-Based Software Systems*, Artech House (2002).
[40] N. Desai, R. Bradshaw, R. Evard and A. Lusk, *Bcfg: A configuration management tool for heterogeneous environments*, Proceedings of the 5th IEEE International Conference on Cluster Computing (CLUSTER03), IEEE Computer Society (2003), 500–503.
[41] N. Desai et al., *A case study in configuration management tool deployment*, Proc. LISA-XVII, USENIX Association (2005), 39–46.
[42] Distributed Management Task Force, Inc., *Common information model (CIM) standards*, http://www.dmtf.org/standards/cim/.
[43] R. Evard, *An analysis of UNIX machine configuration*, Proc. LISA-XI, USENIX Association (1997).
[44] J. Finke, *An improved approach for generating configuration files from a database*, Proc. LISA-XIV, USENIX Association (2000).
[45] J. Finke, *Generating configuration files: The director's cut*, Proc. LISA-XVII, USENIX Association, San Diego, CA (2003).
[46] J. Hart and J. D'Amelia, *An analysis of RPM validation drift*, Proc. LISA-XVI, USENIX Association, Philadelphia, PA (2002).
[47] M. Holgate and W. Partain, *The Arusha project: A framework for collaborative Unix system administration*, Proc. LISA-XV, USENIX Association, San Diego, CA (2001).
[48] L. Kanies, *Isconf: Theory, practice, and beyond*, Proc. LISA-XVII, USENIX Association, San Diego, CA (2003).
[49] G. Kim and E. Spafford, *Monitoring file system integrity on UNIX platforms*, InfoSecurity News **4** (1993).
[50] G. Kim and E. Spafford, *Experiences with TripWire: Using integrity Checkers for Intrusion Detection*, Proc. System Administration, Networking, and Security-III, USENIX Association (1994).
[51] C. Kubicki, *The system administration maturity model – SAMM*, Proc. LISA-VII, USENIX Association (1993).
[52] A. Leon, *Software Configuration Management Handbook*, 2nd edn, Artech House Publishers (2004).
[53] The Linux Standard Base Project, *The linux standard base*, http://www.linuxbase.org.
[54] J. Lockard and J. Larke, *Synctree for single point installation, upgrades, and OS patches*, Proc. LISA-XII, USENIX Association (1998).
[55] M. Logan, M. Felleisen and D. Blank-Edelman, *Environmental acquisition in network management*, Proc. LISA-XVI, USENIX Association, Philadelphia, PA (2002).
[56] K. Manheimer, B. Warsaw, S. Clark and W. Rowe, *The depot: A framework for sharing software installation across organizational and UNIX platform boundaries*, Proc. LISA-IV, USENIX Association (1990).
[57] D.R. Mauro and K.J. Schmidt, *Essential SNMP*, 2nd edn, O'Reilly and Associates (2005).
[58] T. Oetiker, *Templatetree II: The post-installation setup tool*, Proc. LISA-XV, USENIX Association, San Diego, CA (2001).
[59] A. Oram and S. Talbot, *Managing Projects with Make*, 2nd edn, O'Reilly and Associates (1991).
[60] R. Osterlund, *PIKT: problem informant/killer tool*, Proc. LISA-XIV, USENIX Association (2000).
[61] D. Patterson, *A simple model of the cost of downtime*, Proc. LISA 2002, USENIX Association (2002).

[62] Reductive Labs, Inc., *Puppet*, http://reductivelabs.com/projects/puppet/.
[63] D. Ressman and J. Valds, *Use of cfengine for automated, multi-platform software and patch distribution*, Proc. LISA-XIV, New Orleans, LA (2000).
[64] M.D. Roth, *Preventing wheel reinvention: The psgconf system configuration framework*, Proc. LISA-XVII, USENIX Association, San Diego, CA (2003).
[65] J.P. Rouillard and R.B. Martin, *Depot-lite: A mechanism for managing software*, Proc. LISA-VIII, USENIX Association (1994).
[66] F.E. Sandnes, *Scheduling partially ordered events in a randomised framework – empirical results and implications for automatic configuration management*, Proc. LISA-XV, USENIX Association, San Diego, CA (2001).
[67] C. Sapuntzakis, D. Brumley, R. Chandra, N. Zeldovich, J. Chow, J. Norris, M.S. Lam and M. Rosenblum, *Virtual appliances for deploying and maintaining software*, Proc. LISA-XVII, USENIX Association, San Diego, CA (2003).
[68] S. Schwartzberg and A. Couch, *Experience in implementing an HTTP service closure*, Proc. LISA-XVIII, USENIX Association, San Diego, CA (2004).
[69] Y. Sun, *The Complexity of System Configuration Management*, Ph.D. Thesis, Tufts University (2006).
[70] W.F. Tichy, *RCS – A system for version control*, Software – Practice and Experience **15** (1985), 637–654.
[71] S. Traugott, *Infrastructures.Org Website*, http://www.infrastructures.org.
[72] S. Traugott and L. Brown, *Why order matters: Turing equivalence in automated systems administration*, Proc. LISA-XVI, USENIX Association, Philadelphia, PA (2002).
[73] S. Traugott and J. Huddleston, *Bootstrapping an infrastructure*, Proc LISA-XII, USENIX Association, Boston, MA (1998).
[74] Y.-M. Wang, C. Verbowski, J. Dunagan, Y. Chen, C. Yuan, H.J. Wang and Z. Zhang, *STRIDER: A black-box, state-based approach to change and configuration management and support*, Proc. LISA-XVII, USENIX Association, San Diego, CA (2003).
[75] J. Watkins, *Testing IT: An Off-the-Shelf Software Testing Process*, Cambridge University Press (2001).
[76] P. Wilson, *The Definitive Guide to Windows Installer*, Apress, Inc. (2004).
[77] W.C. Wong, *Local disk depot – customizing the software environment*, Proc. LISA-VII, USENIX Association (1993).

This page intentionally left blank

2. The Technology

2.1. *Commentary*

Network and System Administration is often perceived as a technology-centric enterprise. One thing is clear: dealing with a vast array of frequently changing technologies is a key part of a system administrator's lot. In this part the authors describe a handful of technologies, from the old to the new.

Unix has played a major role in developing system administration technologies and has bridged the gap between large systems and personal computing successfully, owing to its openness and great flexibility. However, mainframe business solutions developed by IBM in the 1960s are still very much in use today, albeit by only a few, and these offer benefits such as low-level redundancy and reliability at a high price. Stav sketches the relationship between the mainframes and better-known Unix. More well-known operating systems such as Windows and Unix have been omitted as they are dealt with in a vast array of training manuals and textbooks. VMS, AS400 and mainframe OS390, now z/OS are still used in some spheres of industry, particularly in the banking world.

Dealing with device diversity has always been one of the important challenges of system administration. Many devices had only the most rudimentary operating systems. In the beginning it was thought that special protocols (e.g., SNMP) could be designed to apply external intelligence to dumb devices; today it is common to find Unix-like images as embedded operating systems on many devices, thus allowing great flexibility of management.

Diversity is the price one pays for vibrant commercial development and this leads to other technologies to manage the lower level technologies, resulting in a many-layered approach. For some, the answer has been to attempt standardization, through organizations such as POSIX, IETF, and the DMTF etc. For others, the diversity is an evolutionary approach that leads to more rapid benefits and can be sewn together by meta-technologies. As Alvin Toffler wrote in his book Future Shock: *"As technology becomes more sophisticated, the cost of introducing variations declines."* [1].

E-mail is one of the most arcane and Byzantine systems on the Internet today. It is typical of many technologies that were designed in isolation as playful academic exercises during the early days of Unix and the network and which took root in social consciousness. At the other end of the spectrum, we see Web services entering as a form of ad hoc "standardization" today. Using XML as a protocol design tool, and the Hypertext Transfer Protocol as the transport layer, everything that was once old becomes new again in Web services.

Reference

[1] A. Toffler, *Future Shock*, Random House (1970).

Table 1

Decade	Architecture	OS	Addressing
1964	S/360	MVT	24-bit
1970s	S/370	MVS/370	24-bit
1980s	S/370-XA	MVS/XA	31-bit
1990s	S/390-ESA	MVS/ESA /OS/390	31-bit
2000	z/Arch.	z/OS	64-bit

investments and thus the backward compatibility is essential to protect these investments. The 'traditional' applications are able to coexist with 64-bit code in the z/OS environment.

There has been a major architectural expansion every decade. Table 1 gives a simplified overview [4], page 4.

z/OS is a 64-bit operating system introduced in March 2001, and runs on the System z family of computers. z/OS is based on OS/390, which again is based on the strengths of Multiple Virtual Storage (MVS) from the mid-1960s. z/OS is even today occasionally referred to as MVS by system programmers and in literature. z/OS has a fully fledged Unix environment integrated into the operating system, called Unix System Services (USS), which is POSIX compliant, has HFSes, common Unix utilities and tools, and a Unix shell. The next subsection gives a short historical presentation of Unix.

2.2. *Unix*

The development of Unix started in the early 1970s at AT&T Bell Labs on a DEC PDP-7, initially by Ken Thompson and Dennis Ritchie [3]. It was designed to be a portable operating system. The C programming language was developed in order to rewrite most of the system at high level, and this contributed greatly to its portability. Unix was made available cheaply for universities, and throughout the 1970s and 1980s it became popular in academic environments. In the mid-1980s, AT&T started to see a commercial value in Unix. During the 80s, Unix branched out in two variants because the commercial version, System V, did not include the source code any more. The other variant, called Berkeley Software Distribution continued to be developed by researchers at Berkeley. There exists a myriad of Unix dialects today, e.g. HP-UX, AIX, Solaris, *BSD, GNU/Linux and IRIX.

3. User interfaces

z/OS generally requires less user intervention than Unix systems, although there is currently an interest in 'autonomic' computing which aims to make Unix and Windows operating systems less human-dependent. The z/OS is highly automated, and a lot of routines exist for handling unexpected situations without operator help. In fact, almost 50 percent of the operating system code belongs to recovery and monitoring routines. The traditional user interface to z/OS was the 3270-terminal; these were dedicated computers attached to the mainframe.

These days the 3270-terminals are replaced by 3270 emulators running on PCs, like IBM Personal Communications (for Windows) or x3270 (for most Unix operating systems, including GNU/Linux). Users can interact with z/OS through the 3270 interface, using Time Share Option (TSO), or Interactive Systems Productivity Facility (ISPF). TSO is a command line environment for executing programs, editing, printing, managing data and monitoring the system. TSO is rarely used any more, and largely replaced by the more user friendly and many driven ISPF.

In Unix, users have traditionally interacted with the systems through a command line interpreter called a shell. This is still the most powerful interface to many system services, though today Graphical User Interfaces (GUI) are growing in popularity for many tasks. A shell is command driven, and the original shell was the Bourne shell 'sh'. The shell is still the primary user interface for Unix administration, but most Unix systems offers a GUI, which makes the interaction with the operating system more user friendly for novice users. This is why many regards Unix systems as more user friendly than z/OS.

4. Files and data-sets

The differences in system data organization deserve an explanation. For a person who is used to the Unix-way of organizing data, the way z/OS does it may seem a little strange. In the Unix-world, file data are stored in a sequential byte oriented manner. z/OS organizes data in data-sets, which are record oriented. A record is simply a chunk of bytes, that can either be of a fixed or variable size. A Unix file does not have a common internal structure, from an operating systems point of view. The file is just a sequence of bytes. The internal structure is organized by the application or the user.

Data stored in a data-set can be organized in several ways. The most common are sequential data-sets, where the records are organized in a sequential manner. Partitioned data-set (PDS) and Partition data-set extended (PDSE) contains a set of sequential members. Each of these sequential members is accessed through a directory in the data-set. A PDS is also called a library, and used among other things to store parameters, source programs and job control language (JCL) statements. JCL will be explained later.

Virtual Storage Access Method (VSAM) is yet an access method, used to organize records in a more flexible way than the PDS and sequential data-sets. Examples of VSAM data-sets are Key Sequence data-sets (KSDS), Entry Sequence data-sets (ESDS), Relative Record data-sets (RRDS) and Linear data-sets (LDS). VSAMs are used by applications, not by users. VSAM data-sets cannot be edited manually through an ISPF editor.

The way data-sets are organized also differs from how files are organized in Unix. The data-set structure in z/OS is not hierarchical, like Unix systems are, hence data-sets are not stored in directories. Information of the data-sets is stored in catalogues, such as data-set name and on which volume the data-set is stored. A system has one master catalogue, and multiple user catalogues to organize the data-sets. Data-set names must be unique, since no hierarchical structure (path) separates one data-set from another.

Naming conventions are also different. Names of data-sets must be in upper case, with maximum number of characters 44. A data-set name is divided into qualifiers, separated by periods. Each qualifier can contain maximum eight characters. The first qual-

ifier is called the high-level qualifier or alias, and indicates which catalogue the data-set resides in. A member in a PDS is given in parenthesis. An example of a PDS data-set is SYS1.PARMLIB(IEASYS00) with a member called IEASYS00. PARMLIB is the most important library for z/OS and contains system control parameters and settings. SYS1.PARMLIB can be compared to Unix's etc-directory.

Here follows an example of a source file to illustrate the differences. In Unix, the file is denoted by a file path and a file name, like this:

- /home/bob/project/source.c.

In z/OS the source code can be stored in a data-set like this:

- BOB.PROJECT.SOURCE.C.

This data-set is stored in a catalogue called BOB.

File security is managed in two completely different ways in Unix and z/OS. In Unix systems, security attributes are tightly attached and stored with the file on the file system. The default Unix file security scheme is robust and simple. In z/OS file security is managed by a centralized system component, a security server. This authorized component controls authorization and authentication for all resources in the system, including data-sets. All access to these resources must go through this facility. The security server offers more security control than any Unix system does, and has more fine grained security settings than the Unix approach.

5. I/O operations

The way I/O operations are managed differs a lot between z/OS and Unix systems. The z/Architecture is tailor-built for high-throughput and performs I/O operations in a very efficient way [6]. I/O management is offloaded to dedicated processors, called System Assist Processors (SAP) and other specialized hardware and software components, while general processors can concentrate on user related work in parallel. Big level-two caches, fast data buses and many I/O interconnects make sure that a large number of I/O operations can be handled simultaneously in a controlled and efficient manner.

6. Task- and resource management

The dispatcher and queue manager in z/OS is fairly sophisticated compared to most Unix systems, and executes different workloads in parallel. Average processor utilization in z/OS during normal operations is typically 80–100 percent, in contrast to maximum 20 percent on a Unix system.

Job control language (JCL) [4], page 128, is a language to control and execute jobs in z/OS. It is used to control execution order of programs, prepare and allocate resources, and define input and output data-sets. This is an area where the mainframe differs a lot from the Unix variants. z/OS 'pre-allocate' resources before programs are executed. This applies for disk usage (data-sets), processor power and memory. In contrast, Unix systems allocate resources dynamically during runtime. This makes z/OS very predictable, and it is possible to set specific goals for different classes of work. The dispatcher will, together with

Workload Manager (WLM), Job Entry Subsystem (JES) and Intelligent Resource Director (IRD) do whatever necessary dynamically to meet these requirements. Setting these kinds of goals is not possible in Unix. Because of the advanced queue and resource management in z/OS, it is capable of running many different types of work in parallel in a controlled and efficient way. In contrast, Unix systems tend to host a single or a few services per OS instance, hence the difference in the resource utilization. While the mainframe can host multiple databases by a single z/OS image, in the Unix world, more servers are needed, generally one server per database or application. In the next section we will see what kind of work z/OS and Unix typically do.

7. Platform applicability and typical workloads

Mainframes have traditionally been used mainly for hosting big databases, transaction handling and for batch processing. The current line of System z mainframes has generally become much more function rich and versatile compared to its predecessors. It is capable of hosting almost any kind of services running under z/OS. Many mainframe installations today run traditional CICS and IMS transactions concurrently with modern enterprise applications using J2EE technology and/or service oriented architecture (SOA) concepts. These machines have become more and more universal, and are still being the highest ranking alternative for security, scalability and availability.

z/OS is typically used for:

- Handling large amounts of users simultaneously
- Batch processing
- Online transaction processing
- Hosting (multiple) bigger databases
- Enterprise application servers

Multiple operating systems can run in parallel on the same System z machine, with a virtualization technology built into the hardware, which divides the hardware resources into logical partitions (LPARs). Each LPAR is assigned a dedicated portion of the real storage (memory), and processors and I/O channels can be shared dynamically across the LPARs or be dedicated to the LPARs.

Unix systems are becoming more stable and secure. Unix systems are capable of handling many users in parallel, but it is typically not capable of handling the same amounts as mainframe systems, particularly if the requests require database access.

Unix systems are typically used as:

- Application servers
- Hosting single databases
- Powerful workstations
- Internet and infrastructure services like mail, web servers, DNS and firewalls
- Processing intensive workloads

Please note that the typical usage stated above are a very simplified view of both z/OS and Unix systems. It is important to understand that the platforms are not limited to these characteristics.

Unix systems may outperform the mainframe on compute intensive work, like weather forecasting, solving complex mathematical equations, etc., where raw processor power is the single most important factor. These machines are often referred to as 'supercomputers'. Linux is a popular operating system in such environments because of its lightweight kernel, and flexible and modular design. The fact that GNU/Linux is open source software makes it possible to customize the kernel for almost whatever purpose, thus it is very popular in academic and research environments. Linux is widely used in commercial environments as well. The fact that Linux is cost effective, stable and portable makes it popular as a server operating system, also on the System z servers. Read more about Linux on the mainframe in the following section.

8. Linux on the mainframe

In today's market, Linux is the fastest growing server operating system, and one of Linux's big strengths is platform interdependency. It runs on all kinds of processors, from small embedded computers to big mainframe installations. Linux can run on the System z servers natively in an LPAR or under z/VM. z/VM is a mainframe operating system that virtualizes the z/Architecture. It provides a very sophisticated virtualization technology that has evolved for more than 35 years.

Linux makes the System z hardware platform an alternative to small distributed servers, and combines the strengths of Linux with the high availability and scalability characteristics of the z/Architecture hardware. Linux complement z/OS in supporting diverse workload on the same physical box; it gives more choices for the z platform.

Some of the benefits of running Linux on System z are better hardware utilization and infrastructure simplification. A System z server is capable of running hundreds of Linux images simultaneously. z/VM makes it possible to set up virtual LANs and virtual network switches between the guest operating systems, and data transfer across these virtual network interconnects is as fast as moving data from one place in memory to another. This kind of IP connectivity has a major performance gain compared to data transportation over a physical wire, and this is especially beneficial in infrastructures where the applications are hosted on the same engine as the databases, which very often applies.

9. The future of the mainframe

The history of the mainframe goes back over 40 years, more than any other computer platform. The current line of System z servers has evolved from the System 360 in the mid sixties. The mainframes dominated the IT industry during the 1970s and the 1980s. In the late 1980s and the 1990s cheap microprocessors-computers and distributed computing became increasingly popular. In fact, some people predicted the mainframe's death, when they saw more and more Unix systems in IT infrastructures throughout the 1990s. But this prediction failed; the mainframe is still a strong platform and the reasons are evident.

IBM invests considerable resources in future development and innovation in the System z, both on hardware-, os- and middleware level. In addition to still improve mainframe

Table 2
Continued

Mainframe	Unix	Comments
EBCDIC	ASCII	Native text encoding standard used
Processing unit (PU)	Central Processing Unit	
SYS1.PARMLIB (including others)	/etc/	Default repository for system wide configurations
Assembler, PL/S	Assembler, C	Default kernel development language
Record oriented (Blocks of data)	Byte oriented (Streams of data)	Traditional data format on file system level [1, p. 130]
Control blocks stored in each address space, called the 'common area' and consist of System Queue Area (SQA), Link Pack Area (LPA) and Common Area (CSA)	/proc/ (represented as virtual files)	Control and information repository for the kernel, where important kernel/ nucleus structures are held during runtime
Supervisor call (SVC)	System call	User processes invoking operating system services, which often run in privileged mode
SDSF	ps, kill	Program management
ISPF editor	emacs, ed, vi, sed ...	Text editors

References

[1] M. Ebbers et al., *Introduction to the New Mainframe: z/OS Basics*, IBM (2006).
[2] E. Irv, *The Architecture of Computer Hardware and System Software, An Information Technology Approach*, Wiley (2000).
[3] D.M. Ritchie and K. Thompson, *The Unix time-sharing system*, Communications of the ACM **17** (7) (1974).
[4] P. Rogers et al., *Abcs of z/os system programming*, IBM (2003).
[5] K. Stav, *Clustering of servers, a comparison of the different platform implementations*, University of Oslo (2005).
[6] D.J. Stigliani et al., *IBM eserver z900 I/O Subsystem*, 40, IBM (2002).
[7] *z/Architecture Principles of Operation*, (5), IBM (2005).

– 2.3 –

Email

C.P.J. Koymans[1], J. Scheerder[2]

[1] *Informatics Institute, University of Amsterdam, Kruislaan 403, 1098 SJ, Amsterdam, The Netherlands*
E-mail: ckoymans@science.uva.nl

[2] *E-mail: js@xs4all.nl*

It is surprisingly hard to write an electronic mail system without messing up.
Wietse Venema

1. Introduction

This chapter of the handbook deals with electronic mail, mostly in the familiar form commonly found on the Internet. Email is discussed in terms of its architecture, the protocols that constitute it, related technology, and the surrounding problems of both technological and non-technological nature. Where appropriate other forms of electronic mail are addressed.

2. Mail history

As one of the oldest and most valuable applications of computers and internetworking, the now 40 years old electronic mail has a rich history which will be described in the following subsections.

HANDBOOK OF NETWORK AND SYSTEM ADMINISTRATION
Edited by Jan Bergstra and Mark Burgess

2.1. *Pre-Internet mail*

Contrary to popular belief, email[1] did not arrive on the scene with the Internet. Email predates the Internet, and even the ARPANET. The first email system, a messaging system for multiple users on a single time-sharing mainframe system, MITs Compatible Time-Sharing System, was deployed in 1965. Email message exchange between distinct computers by means of a computer network started in 1971. RFC 196 [47], by Dick Watson, dated July 1971, already described a (probably never implemented) email system, while Ray Tomlinson sent the first actual network email message in late 1971. The vehicle for these early email exchanges was ARPANET where messages piggybacked on the already available File Transfer Protocol (FTP). Email was not yet a first-class-citizen, but it was a very powerful mechanism for communication.

Quoting Ray Tomlinson [43] on the infancy of network email:

> The early uses were not terribly different from the current uses: The exceptions are that there was only plain text in the messages and there was no spam.

2.2. *Internet mail's predecessors: UUCP, BITNET and DECnet*

The early 1970s saw the birth and procreation of Unix. As an asynchronous way of loosely interconnecting otherwise unconnected computers, the Unix to Unix Copy Protocol (UUCP) was invented in 1978 at Bell Labs. UUCP provides data exchange between computers on the basis of ad-hoc connectivity, such as direct dial-up connections between computer systems.

Based upon UUCP, store-and-forward protocols for message exchange and message routing were designed and globally implemented. This included not only (or even primarily) email messaging: UUCPs first-class citizen, one could say, was in fact not email, but Usenet News. This is the email and news as we know them today, although both have long since transitioned to data transfer using Internet transmission protocols rather than UUCP.

Similar to UUCP, the 'Because It's Time Network' (BITNET) interconnected IBM mainframes in academia, and similar store-and-forward message exchange services were started on BITNET in 1981.

Unlike UUCP, and much more like today's Internet, the then prevalent computer company Digital Equipment Corporation offered inter-computer network connectivity for its own computers by the name of DECnet, which started in 1975. DECnet included full message exchange and message routing facilities.

2.3. *The start of Internet mail*

In the early 1980s, the Internet mail infrastructure was created. Most important here were the efforts made by Postel and colleagues in creating SMTP, the Simple Mail Transfer

[1] We follow Knuth in writing email instead of e-mail [26].

Protocol, within the IETF. In 1982, while the ARPANET was still transitioning from the old NCP protocol to TCP/IP, the first widespread implementation of mail over SMTP was implemented by Eric Allman. Sendmail, the followup of delivermail, was shipped with 4.1c BSD Unix, which promoted email to a first-class citizen on the then emerging TCP/IP-based Internet.

At that time, several message exchange networks existed already. In order to cope with all these different networks and addressing systems, an elaborate system for rewriting was introduced into sendmail. This powerful rewrite engine is still present in modern versions of sendmail, although there is a tendency to replace it by table-driven lookups in modern MTAs (see Section 3.3.2) like postfix, qmail and the upcoming sendmail X.

A *gateway* between two messaging systems is a system that can accept and send messages for both messaging systems, and can direct messages between them, manipulating them as needed. A main Internet mail relay in these early days may very well have handled Internet mail, UUCP mail, BITNET mail and DECnet mail, gatewaying between all those. Due to differences in several details, like addressing, this involved a lot of rewriting magic, which sendmail handles particularly well.

From now on, when we mention email (or Internet mail), we mean SMTP-based mail running over TCP/IP.

3. Mail architecture

In this section we will discuss the main architectural components for an email infrastructure. These components are addresses, protocols, agents and message formats.

3.1. *Mail addresses*

During the development of mail, several addressing schemes have been used. UUCP used full path addresses like the famous 'bang path' `mcvax!moskvax!kremvax!chernenko` used by Piet Beertema in his 1984 Usenet hoax, archived by Google [2]. DECnet addresses took the form *host::user*, where host was supposed to be unique throughout DECnet. Several other addressing schemes existed for local or private networks.

The oldest mail address format was the one used in ARPANET, which was later also used in BITNET, being *user@host*. Again, this presupposes a flat namespace, where every host throughout ARPANET had a unique name. In the beginning that was reasonable, but with the explosive growth of the Internet no longer maintainable. Hence a new, hierarchical scheme was desperately needed.

3.1.1. *Mail hierarchy and DNS* The solution to the problem of the large namespace came with the invention of DNS, the Domain Name System, in 1983. The domain name system offers a distributed database, where administration of parts of the hierarchically organized

namespace can be delegated to autonomous servers, which have authority over the delegated subtree. The labels used for the different layers of the hierarchy are separated by dots. Each layer only needs to check that the labels used for its own sublayers are locally unique. Email addresses now take the form *user@host.some.domain*, where an arbitrary number of layers is allowed after the @-sign.

Another extension was added to DNS in 1986: MX records. Traditionally, mail has always used a store-and-forward mechanism to route and deliver mail. There is no need for a direct connection between the sending machine and the machine where the mail should finally be delivered. MX records provide an abstraction layer for email addresses. For instance mail to *user@some.domain* could be instructed towards the next hop by way of an MX record, for example pointing to relay.some.domain. This relaying machine could accept and store the mail in order to forward it on to its, possible final, destination.

It is even possible to forward to non-Internet destinations like machines in the UUCP network. This way users on machines that are not connected to the Internet can have Internet-style mail addresses. This option should not be confused with the use of pseudo domains in addresses like *user@host.UUCP*. The UUCP top level domain does not exist in DNS, and only lives inside rewrite rules of sendmail or similar programs.

It is an established 'best practice' to make use of DNSs possibilities to create a hierarchical layer whenever this is possible within an organisation. System engineers should arguably also try to mimic this same hierarchy when the email infrastructure is set up, even in the case of a centralised administration. This results in more flexibility, scalability and the option to reorganise the infrastructure more easily or treat certain groups in special ways.

3.2. *Mail protocols*

Now that addressing has been taken care of, we can start looking at the protocols used for mail transport and pickup. As already mentioned, email became a first-class citizen only after the specification of SMTP in August 1982 in RFC 821 [37]. This protocol is used for mail transport. Later on protocols like POP[2] and IMAP[3] were introduced to facilitate a network based interaction with mail readers.

3.2.1. *(Extended) Simple Mail Transfer Protocol* SMTP was indeed a very simple protocol intended for mail transport. It lacked some of the more advanced features of the X.400 message service. In the original specification only a few commands were defined.

The most important commands are

- The Hello command (HELO) is used to introduce the sending SMTP client to the receiving SMTP server. In the response to this command the server introduces itself to the client.

[2] Mostly version 3. We will write POP instead of POP3.

[3] Mostly version 4rev1. We will write IMAP instead of IMAP4rev1.

3.3.2. *Mail Transport Agents* The heart of the email infrastructure is formed by the mail transport agents,[5] that take care of mail routing (see Section 4) and mail transport across the Internet. MTAs talk SMTP to each other to accomplish this task. MTAs need to understand email addressing formats, might do gatewaying to non-Internet mail systems, must implement a queueing system for mail in transit, handle bounces and other error conditions and take care of forwarding, aliases and special delivery to programs or mailing lists. It is important that MTAs implement some security mechanism and/or access control in order to thwart abuse, see Sections 6 and 8.1. At the ingress MTAs talk to MUAs (or sometimes MSAs) and at the egress they deliver mail to MDAs.

Examples of MTAs are Sendmail, Postfix, qmail and Exim. More on these mail servers can be found in Section 5.1.

3.3.3. *Mail Submission Agents* Not all mail user agents are able to inject email into the system in a completely satisfactory way. For instance, domains used in the SMTP-envelope should be fully qualified domain names. And also the syntax of header fields inside the message from the SMTP-DATA transfer might need some extra attention. This is where the mail submission agent plays its role, as an intermediate step between MUA and MTA. One often sees that this MSA-functionality is built into the mail transfer agent itself.

More on the specific tasks of mail submission agents can be found in RFC 2476 [16].

3.3.4. *Mail Delivery Agents* When an email has reached its final destination, it should be put in the user's incoming mailbox, often called 'INBOX'. Traditionally this was a plain file in mbox format (see Section 3.4), but mailbox storage can also be implemented differently, e.g., as mail directories or mail databases. It is up to the mail delivery agent to perform the task of storing incoming mail in a permanent place for subsequent access by the user. An MDA can operate as simple as the old `/bin/mail`-program that just appends incoming mail to the user's INBOX or as sophisticated as the `procmail`-program that exercises all kinds of operations like user filtering, header manipulation or forwarding on the incoming mail before final delivery.

LMTP, as defined in RFC 2033 [32], is a protocol designed for the purpose of passing messages by a mail delivery system to the final delivery agent. This final delivery agent may run on the local host, but it can also run on a remote host. LMTP can be viewed as a simplification of the SMTP protocol, with the queueing mechanism taken out. Using it provides a little extra functionality, since ESMTP extensions can be catered for. Additionally, LMTP can be used to abstract from actual mail delivery in such a way that mail acceptance and mail delivery become many-to-many relations instead of one-to-one relations, thus offering virtually boundless scalability. It is supported by the main MTAs mentioned before.

3.3.5. *Mail Access Agents* After a mail message has been sent by a user, transported by MTAs and delivered by an MDA, it waits for the intended recipient to access his email. Most of the time the recipient's user agent does not have direct access to the mail store and contacts a mail access agent to interact with the message store on its behalf. The

[5] Sometimes 'transport agents' are referred to as 'transfer agents'.

MAA talks with the user agent using an access protocol like POP or IMAP, as discussed in Section 3.2.2.

Examples of MAAs are Courier, Cyrus, UW IMAP, Dovecot and Qpopper.

3.3.6. *Mail Retrieval Agents* Sometimes it is not a user agent that directly (or indirectly via an MAA) checks for new incoming mail, but an automated agent that works on behalf of the user, called a mail retrieval agent. The MRA accesses the mail store without explicit user intervention, using an MAA when needed. After local processing it might reinject the mail into the mail system for further routing and processing. It is often used to access secondary mailboxes users might have on different servers in order to integrate them with the primary mailbox.

As can be seen in Fig. 1, as illustrated with the dashed lines, this might lead to several iterations of the mail cycle, hopefully not creating non-terminating loops. Typical examples of MRAs are fetchmail and getmail.

3.4. *Mail message formats*

A companion document to RFC 2821 on SMTP is RFC 2822 [39], titled 'Internet Message Format'.[6] In this specification the format of the actual message itself, which is transferred during the SMTP DATA phase, is defined. The first part of a message consists of a number of mail header lines, followed by an empty line, followed by the message body. Lines consist of US-ASCII (1–127) characters and are terminated by the combination CRLF of a carriage return followed by a line feed, as is prescribed customary in text-oriented Internet protocols. Lines are allowed to contain up to 998 characters, but it is good practice to limit lines to 78 characters in order for the mail to be readible on 80 character wide terminal displays.

3.4.1. *Mail headers* Each header line consists of a field name (made out of printable ASCII characters (33–126)[7]), a colon(':'), a space character ('␣'), and the field body, in this order. To keep header lines readable they may be 'folded' over separate physical lines by preceding existing whitespace in the field body with a linebreak, effectively creating continuation lines that start with whitespace. The body of a message is essentially free-format, although the MIME specification introduces new subformats.

Mail headers contain important meta-information for the content of the message. The headers contain information like the date of mail composition (*Date:*), originator fields like author (*From:*) and actual sender (*Sender:*), destination address fields like recipient (*To:*) and carbon copy recipients (*Cc:*), identification fields like a unique message identifier (*Message-ID:*) and referencing information (*In-Reply-To:*, *References:*), informational fields like subject (*Subject:*), trace fields which contain parts of the SMTP-envelope like intermediate MTAs (*Received:*) and return envelope-sender address (*Return-Path:*), but also a whole range of (optional) fields added by user agent or filtering software.

[6] This title is a little bit more compact than the title of its predecessor, RFC 822, being 'Standard for the format of ARPA Internet text messages', but it still does not explicitly mention Mail.

[7] With the exception of 58, which represents a colon.

Another important purpose of the alias mechanism is the ability to deliver email to a program instead of storing it in the mail store. For security reasons it is forbidden to mail directly to programs, but the administrator of a mail domain can enable this possibility indirectly by using an alias, for example to support mailing lists, see Section 4.3.

4.3. *Mailing lists*

Mailing lists are used to expand one email address into a lot of email addresses in order to reach a community of users interested in the same topic. It is possible to implement this behaviour by using the alias mechanism just described. In order to have more flexibility, to be able to rewrite or extend the mail message, to be able to archive messages and so on, an implementation often uses the alias file to deliver the email to a program that takes the responsibility of expanding the mailing list address into the list of subscribers to the list while also performing additional administrative tasks. With very large mailing lists, a subscriber is often itself the address of a mailing list exploder. An important property of properly configured mailing lists is that it replaces the original envelope sender address with a predetermined envelope sender address of the mailing list owner, this in contrast with what happens during normal exploding or forwarding. The idea here is that error messages are sent to the mailing list owner in stead of to the mail originator, because the mailing list owner is the one able to act upon these errors.

4.4. *Mail relaying*

If a machine accepts incoming SMTP mail and forwards it through an outgoing SMTP connection to another MTA, this machine is called a relay. Within organisations this mechanism is often used to transport mail from or to a central machine, which operates as the main mail relay for a certain domain. Mail routing between domains is mostly taken care of in one hop, but sometimes the MX mechanism supplies a fallback mechanism for mail delivery in case direct delivery is (temporarily) not possible. This is an intended and useful way of relaying.

If a machine accepts arbitrary incoming SMTP connections with envelope recipients to arbitrary other domains, this machine is called an ‘open relay’. An open relay can easily be abused by spammers and other abusers to inject large volumes of unwanted mail. More about this can be read in Section 8.1 on mail abuse.

4.5. *Virtual domains*

A mail server can be configured to accept email not only for its principal domain, but also for many other domains. These domains are called virtual domains. The mechanism by which mail arrives at the correct server is again the MX mechanism. What happens after that depends on the kind of virtual hosting. One possibility is to map virtual mail addresses to addresses in the real principal domain or even in foreign domains by making use of

some virtual domain table lookup. Another possibility is to have real separate mail stores for your virtual domains, but this requires a virtual domain aware POP or IMAP server to make the mail store accessible on a per domain basis.

4.6. *Delivery status notification*

RFCs 1891–1894 [30,31,44,45] define an extension to SMTP (see Section 3.2.1) that is meant to be used as a more concise and complete way of generating success and failure messages related to mail transport and delivery. Mail clients can ask per recipient for a combination of notifications of success, failure or delay, while specifying whether the full message or only the message headers should be returned.

4.7. *Mail error notification*

An important aspect of mail handling is error notification in case something prohibits normal delivery of email. Because error notification uses the mail infrastructure itself, great care has to be taken that no mailing loops arise, see also Section 4.9. Many error conditions exist, ranging from syntax errors in commands or parameters to problems with delivery like non-existing mailboxes, storage limit excess, or authorisation errors. In most cases errors are processed by the sending mail transport agent, which generates a (new) bounce email addressed towards the envelope sender of the email. Every mail receiving domain is required to have a 'postmaster' mailbox or alias which ultimately should be read by a human responsible for fixing errors and problems with the mail system. The postmaster in charge of the sending mail transport agent gets a copy of this bounce message with the mail body of the original mail suppressed.

In case the specially constructed error message generates a bounce itself, a so-called 'double bounce', this double bounce should be delivered to a local postmaster able to resolve any configuration errors. Triple bounces should never occur.

In sending bounces a special convention is used for the envelope sender address, being <>. This special address can not occur as a normal recipient address, but should always be accepted as a valid sender address.

4.8. *Mail address rewriting*

In the course of mail processing it is often the case that addresses, sender and recipient, envelope and header, have to be rewritten. This is evident for envelope recipient addresses in case of alias expansion or forwarding. This is also evident for calculating return addresses for envelope senders. A need for rewriting header addresses occurs when domains want to rewrite addresses into preferred canonical forms. Often these rewrites can be implemented with the help of table lookups. The pre-X versions of sendmail have a very powerful rewriting engine based on regular-expression-like pattern matching on tokenised address strings. The art of crafting one's own rewriting rulesets and rules, as practised by

sendmail magicians, has been dying slowly and steadily though, and these days sendmail's configuration files are usually generated by m4-based macros from high level option specifications. Additionally, the need for a general rewriting engine for address formats has largely disappeared because of standardisation on Internet style mail addressing.

4.8.1. *Masquerading* Masquerading is a special form of address rewriting that is applied in order to hide the specific details of the mail infrastructure of an organisation for outgoing and incoming mail. At the boundary of a mail domain any mail addresses referring to internal hosts or domains in outgoing mail are rewritten to refer to a top level domain only. Incoming mail, addressed to users at this unified top level domain, can be routed to the specific internal destinations by using rewriting or table lookup.

4.8.2. *Canonical mapping* Canonical mapping is a special form of rewriting of the local part of a mail address in order to format these local parts according to conventions an organisation wishes to use in the outside world. In most cases it is used to rewrite a login account name to a FirstName.Surname or Initials.Surname convention. Canonical mapping is often combined with masquerading to completely transform internal addresses to addresses exposed to the outside world at the boundary mail relays.

For any canonicalisation of addresses care needs to be taken to avoid potential conflicts. The larger the user base, the more potential for conflict. Furthermore, the canonicalisation scheme is important: the lesser distinguishing characteristics appear in the canonical form, the more likely a clash will be. For example, FirstName.Surname can be expected to generate fewer clashes than Initials.Surname, since two persons sharing a surname might very well have identical initials, even though their first names differ. John and Jane Doe have unique canonical addresses in the former canonicalisation scheme, but not in the latter.

Canonical mapping almost always uses table lookups, because there is usually no regular pattern in the substitution.

4.9. *Mail loops*

One of the dangers that mail systems have to be careful about is the occurrence of a mailing loop. Most mailing loops are generated by automailers, be it bouncers, vacation programs or user scripts. There is no foolproof system to avoid mailing loops, but several precautions can be taken to avoid them. Let us first look at some scenarios.

4.9.1. *Mail loop scenarios*

SCENARIO 1. The classic example of a mailing loop is created when a user has two email addresses and forwards email from the first to the second address and vice versa. Some hop limit count will finally bounce the message to the original sender. If the user not only forwards the mail but also delivers a local copy, multiple copies of the email, equal to the hop count, will be delivered to the combined addresses.

5.1.2. *Postfix, qmail and Exim* Both postfix and qmail are MTAs that aim for a completely different architecture from sendmail's. A central notion in both software packages is that of a queue manager that assigns work coming from a number of queues, for incoming, outgoing, deferred or erroneous messages, to a number of processing agents, for local or remote delivery and for address rewriting and lookup. This architecture accounts for the modularity and scalability of mail processing these MTAs are known for.

Starting in 1995 Philip Hazel's Exim, based on ideas from an earlier program called Smail-3, was developed as a stand-in replacement for sendmail. Its properties are a more user friendly configuration than sendmail, a good security record, extensive facilities for checking incoming mail and extendability.

5.2. *Mail clients*

Email clients abound. We only discuss the different classes of email clients, with a few examples. An obvious distinction can be made between clients with a graphical user interface and clients with a terminal-based text interface. A more important distinction is whether the client is native (speaking IMAP, POP and/or SMTP itself) or is in fact accessing a proxy, using for instance a web interface, to do the mail handling.

5.2.1. *Native clients* Examples in this category are Eudora, Mozilla Thunderbird, Evolution and KMail as graphical clients and mutt or pine as text based clients. Microsoft Outlook is also a messaging client, using something similar, but not quite identical, to standard Internet mail.

Graphical clients are often considered easier to use, but have also given rise to a habit of sending HTML-based mail instead of plain text mail, even in situations where plain text is the essential content of the message, unnecessarily complicating the task of reading, understanding and replying to email messages.

Being able to use a terminal based client offers the capability to access your email in situations where only minimal support is available and a graphical client is not present. Email clients like mutt are quite advanced and support access to your mail via IMAP, securely if needed, and with support for content encryption via PGP, GnuPG or S/MIME, see also Section 6.

5.2.2. *Proxy clients: Webmail* In certain situations no access is available to a graphical mail client and also not to terminal emulation and a text mail client. One example of this is most Internet cafes. Already in 1995 a startup called Hotmail, acquired in 1998 by Microsoft, was offering email services through a browser interface as a first step to make the Internet experience browser centric, leading many people into the belief that the Internet and the Web are synonyms.

The almost universal availability of browser access is the main advantage of webmail. Disadvantages are that offline preparation of email is relatively clumsy, many commercial providers of webmail offer rather limited mail storage quota and organisation of mail on a local hard drive is not possible. Webmail, at this point, is no universal replacement for a true email client.

One of the more popular, open and extensive webmail server packages is SquirrelMail [6].

5.2.3. *User filtering* Whatever email client one is using, an important feature is the ability to do user filtering. Server side filtering is certainly an option, made available to users on an opt-in basis, but most flexibility is retained by using client side filtering. Marking messages as belonging to a certain category is a server side feature that helps in client side filtering. Filtering is used to organise mail into folders, to recognize and act upon spam, junk mail or mail containing malware and to auto-reply to certain kinds of messages, like when on vacation.

On Unix systems mail filtering is often implemented as a separate program acting on behalf of the mail recipient. Procmail is a well-known and advanced mail filter and delivery agent. Email clients often have built-in filtering rules for detecting junk email, most of them based on Bayesian filtering, as pioneered by SpamCop [17] and Microsoft Research [28]. More on spam and junk mail in Section 8.1.1.

Sieve [42] is a mail filtering language designed for filtering email messages at final delivery time. It is designed to be implementable on either a mail client or mail server. Dovecot's mail delivery agent, LDA, implements it, as does the Cyrus IMAP server. In the past, the Mulberry MUA allowed users to create user-level Sieve filters in such a way that the resulting filters could be pushed to the server, thus combining direct user manipulation of mail filters with actual delivery-time mail filtering.

6. Mail security

Pondering the Tomlinson quote (Section 2) one could say that email in the early days did not need security because it was used in an open and collaborative environment. Times have changed. Malware and other nuisances have substantially turned email's playground more hostile, so it needs to exercise caution too.

When discussing security five aspects are often distinguished: *authentication*, *authorization*, *integrity*, *confidentiality* and *non-repudiation* (accountability).

In this section we will look at mail security from the angles of *transport protection* by way of encrypted and (asymmetrically) authenticated connections and *content protection* by way of encryption and signing of message bodies.

6.1. *Transport security*

When transporting email over the Internet via SMTP or accessing an MAA to collect email by IMAP or POP, it would be nice to know that the client is talking to the correct, intended mail server: typically, one wants to disclose authentication information to the proper server only, not to just any system that (maliciously?) tries to assume its identity. On the other hand, a server may wish to know what client it is talking to. Differently put: the client and server may need to certify each others authenticity.

For IMAP and POP it is essential to establish the user's identity prior to granting access to the user's mailbox, that is to authenticate the user prior to authorizing access. For SMTP, client authentication might help against spam and other nuisances. Without SMTP authentication, anyone can inject any message under any identity, so any message can be sent by anyone. Without message authenticity there is no non-repudiation.

Traditionally authentication is performed using a plain text username and password that can easily be intercepted in transit. The risks of *spoofing* and *password sniffing* make traditional authentication (and, therefore, authorization) highly vulnerable for potential abuse.

However, the risks do not stop there. In fact, it is not just the authentication details of a traditional SMTP, POP or IMAP session that can easily be 'sniffed'. The full details, including the potentially confidential message content, are open to prying eyes this way: confidentiality is poor. Additionally, the sessions may even be intercepted and tampered with, putting data integrity at risk.

In the web world mechanisms for resolving these issues have existed since Netscape specified its Secure Socket Layer. Using certificates a server can prove its identity to clients and, if asked for, clients can prove their identity with certificates of their own. An encryption key is negotiated to make the connection confidential. This has the extra benefit of protecting a username/password protocol for authentication from snooping. This web world mechanism can also easily be applied to mail protocols.

6.1.1. *SSL, TLS and STARTTLS* SSL (Secure Socket Layer) has been invented by Netscape for use in e-business applications on the Web. SSL version 3.0 has later been adopted by the IETF and renamed, with minor modifications, to TLS (Transport Layer Security) version 1.0 in 1999 [10]. It provides mechanisms for host identity verification as well as data encryption in order to prevent spoofing, sniffing and tampering. This mechanism was used to protect Hyper Text Transfer Protocol (HTTP) transmissions, turning it into https, secure http, with default TCP port 443 instead of the usual port 80.

These same techniques can also be applied to IMAP and POP, leading to secure imap (imaps; default port 993 instead of 143) and secure pop (pop3s; default port 995 instead of 110). Somehow, understandably so considering Section 6.3, the same mechanism for SMTP, secure smtp (smtps; default port 465 instead of 25) is hardly used and was never formally acknowledged. IANA [20] even lists 'URL Rendesvous Directory for SSM' (urd) as using 465/tcp.

An alternative mechanism, STARTTLS [18] (an SMTP service extension), is used to 'upgrade' an existing, unprotected TCP connection to a TLS-based, protected TCP connection. STARTTLS is also available in many IMAP and POP implementations, usually in the form of an advertised capacity, with 'sensitive' features, such as authentication, becoming available only after successfully establishing encryption for the existing connection first. That way, only the standard SMTP port needs to be assigned and published, while still offering the facility of network transport encryption. No dedicated port to provide the encrypted variant for SMTP or other protocols is needed when using STARTTLS. A special port is assigned by IANA (submission 587/tcp) for connections to MSAs where authentication via a secure connection is preferred. Most mail systems still use port 25 for submission and transport without any protection or authentication.

6.2. *Authentication frameworks*

For IMAP and POP, after a secure connection is established between client and server, the usual builtin commands specifying user and password can be used safely. SMTP does not have a standard authentication command. In RFC 2554 [34] an 'SMTP Service Extension for Authentication' was defined, using the 'AUTH' command which is based on the 'Simple Authentication and Security Layer' (SASL [33]) mechanism. SASL enables several authentication schemes, like Kerberos, One Time Passwords and GSSAPI-based schemes, and is currently widely used.

6.3. *Mail content security*

Another matter is when users feel the urge to protect the content of their mail messages from prying eyes.

When that need arises, one should take notice of the fact that the very store-and-forward nature of the SMTP protocol itself acts contrary. Even with full transport security at the network level for an SMTP transaction between two hosts, the messages will still, typically, be stored in order to be queued for further forwarding. This process will repeat itself for any 'hop' on the way to the final message destination. Any host on this delivery path gets the full message, without protection. Any mail host kan keep copies of the messages it passes on, and it can even change the messages when doing so.

When confidentiality, integrity and/or authenticity of the message itself is important, the MUA therefore needs to apply some extra measures in order to encrypt and/or sign the message content, using MIME to deliver the encrypted message and/or the signature. The two most prominent standards implementing privacy enhanced mail, not to be confused with the first 'secure email standard' called Privacy-Enhanced Mail (PEM [1,21,22,27]), are OpenPGP/MIME and S/MIME. Another standard is MIME Object Security Standard (MOSS [8]). For an overview of the different mechanisms and the pitfalls of the simple sign-&-encrypt habit see [9].

6.3.1. *PGP, GnuPG and OpenPGP* PGP stands for 'Pretty Good Privacy' and was developed by Phil Zimmermann to provide privacy and authentication of documents in general. Many subtly different versions of PGP have been released during the years. This led to the development of the OpenPGP [5] standard and an open source implementation called the Gnu Privacy Guard (GnuPG [25]). Both PGP and GnuPG have been used to encrypt and sign messages inline. Later the MIME standard was used to separate the content of the mail message from the signature or to convey an ASCII-armored encrypted message [11]. MIME types 'multipart/signed' and 'multipart/encrypted' are used for this purpose.

The trust model proposed by OpenPGP is that of a web of trust. There is no centralised hierarchy of authorities, but a loosely coupled system of users signing other user's public keys. Keyservers collect these public keys with attached signatures in order to distribute these on demand to MUAs who need them to send secure email.

6.3.2. *S/MIME* S/MIME stands for Secure/MIME [38], which describes how to add cryptographic signature and encryption services to MIME data. It is based on the 'Cryptographic Message Syntax' (CMS [19]), which is derived from PKCS #7 as specified by RSA Laboratories. S/MIME uses 'application/pkcs7-mime' and 'application/pkcs7-signature' as MIME types for encryption and signing.

The trust model used in S/MIME is based on X.509 certificates and a centralised hierarchy of Certification Authorities.

7. Relations to other technologies

On the Internet several other technologies exist that have an intimate relationship with email, or that serve a similar purpose.

7.1. *Directory services*

Email cannot exist without proper functioning directory services. A crucial directory service used by email is DNS, without which no Internet application could survive. Specific use of DNS is made by mail transport agents by looking at MX records to find the next hop mail server.

Directory services are needed by mail upon local delivery as well. When accepting a message for a purported local user, the mail system has to check whether that user exists. Traditionally, the mail system consulted a local user database for this purpose. Other directory service mechanisms, such as LDAP, are gaining importance though.

Email software may additionally use directory services (again, typically LDAP) for authentication, routing, aliasing or forwarding.

7.2. *Usenet news*

Email is designed as a medium for interpersonal message exchange: one on one, basically. *Usenet news*, or simply *news*, is a similar medium, with the fundamental difference that messages are directed to a collective rather than a person. News *articles* are very similar in format and structure to email messages. In fact, one and the same message format RFC applies to both email and news messages. However, instead of sent to a person's email address, these articles are *posted* to a *newsgroup*. Anyone who is interested can fetch the article from any newsserver that carries the newsgroup in question.

In addition to the extensive similarities between email messages and news articles, email and news share more common ground. Both are based on asynchronous message exchange technologies, using a store-and-forward approach for message transport. With this much similarity, the existence of gateways between email and news, spanning both worlds, should not be too surprising.

8.1.4. *Filtering* Spam and worms confront the user with unwanted, potentially even dangerous, messages. To help guard the user, filtering technology can be applied. Popular packages for the detection of spam and worm/virus/trojan/rootkit attachments are SpamAssassin and ClamAV, respectively.

In addition to common user annoyances, corporations may very well impose policy restrictions on email use and apply mail filters to enforce these policies. This way, the use of mail attachments, for example, may be banned altogether or restricted to certain file types, or files up to a certain maximum size.

Filtering can be invasive, and when applied naively prove countereffective. For example, a virus scanner may refuse encrypted mail messages. On the one hand, that may be a good thing. Nothing goes through unless checked. On the other hand, this policy may unintentionally give rise to the dissemination of confidential information. If the only way to send information is to do it insecurely, then people will be forced to do so.

8.1.5. *Black- and whitelisting* Immediate pass- or block-filtering based upon a few simple criteria has become increasingly popular recently. The basic idea here is that some obvious particular message property is looked up in a list. When the particular property is found in the list, that indicates either that the message is undesirable, in which case the list in question is known as a blacklist, or that the message is acceptable, in which case the list is called a whitelist.

Properties typically used in black- and whitelisting are the envelope sender, the envelope recipient or the IP address of the host that's offering the message.

Black- and whitelist lookups are often, but not necessarily, queries of an external resource. Typically, these queries are implemented as DNS lookups, where a non-null response indicates a positive match. This is commonly referred to as real-time black- and whitelisting. Many real-time black- and whitelisting services exist, both non-profit as well as for-profit.

8.1.6. *Greylisting and other approaches* A somewhat recent approach to fight mail abuse builds upon the observation that spammers really need simple and cheap ways to spread massive amounts of messages. To keep it simple and cheap, the observation continues, spammers do not really play by the rules imposed by the SMTP protocol.

One approach based upon this observation can be seen in the `greet_pause` option introduced in Sendmail v. 8.13. Not caring what the other end has to say, spammers typically do not pay attention to its remarks and immediately start sending SMTP commands to inject their message once connected. With the `greet_pause` option set, Sendmail refuses to accept messages offered by a peer before it even got the chance to greet ('HELO', 'EHLO').

Similarly, Postfix' `reject_unauth_pipelining` parameter makes Postfix refuse messages from peers that send SMTP commands ahead of time.

Greylisting leverages the SMTP protocol to a somewhat deeper extent. When stumbling upon a transient (non-fatal) error while offering a message, the SMTP protocol requires that the message gets offered again after a brief pause, but within reasonable time. Even without qualifying the notions of 'brief' and 'reasonable' here any further, it should be obvious that this introduces a little complication for the sending party. Greylisting looks at

three parameters, which together are called a 'mail relationship': the IP address of the host attempting a delivery, the envelope sender, and the envelope recipient. If a mail relationship is either new or too recent, service is refused with a transient error message. However, mail relationship is timestamped and remembered. Now when a message – probably the same message – is offered with a mail relationship that is sufficiently old according to its timestamp the message is accepted without further ado.

Greylisting effectively puts a stop to the typical fire-and-forget tactic often used by spammers at the cost of a small, but one-time only delay for valid mail correspondences. Even if spammers actually perform the extra work of offering their messages again at a later time, that means that the cost of spamming has substantially increased. Furthermore, between the first refusal and the second offering real-time blacklists and dynamic signature databases of spam-detection mechanisms may very well have been updated for it. This way, greylisting turns out as a measure that not only fights spam technologically, but it also affects the basic economics of spamming.

8.2. *Mail usage*

8.2.1. *Conventions* A few conventions have historically developed. Early mail users, working on fixed size character terminals, found ways to communicate effectively within the confinements of their working environment.

A few habits deserve mentioning. *Author attribution* was conscientiously used to indicate which part of a mail message was attributed to which author. In conjunction with that, a strong discipline of quotation was maintained: when replying to a message, any existing quoted text was marked with a designated quote prefix (>). Existing quoted material became prefixed with >> that way, and so on. This made it possible to unambiguously determine which text parts led back to which authors.

Another notable email habit was to write remarks in context, using proper attribution and quotation of course. Any remark to be made immediately followed the relevant (quoted) text part in the original message, thus interweaving remarks with quoted material from the original message. Proper context also implied that only the relevant parts of the original message, providing sufficient context, were quoted in replies.

This way, messages were minimal in size, sufficiently self-contained as well as unambiguously interpretable.

8.2.2. *Etiquette* To help keep email communicaton effective, some basic rules are often proposed. The established, yet besieged, 'best practices' described in Section 8.2.1 are usually found back in guidelines on email etiquette. A lot more suggestions, that all fall in the category of things that should really be painfully obvious, but somehow turn out otherwise, can be made and float around the Internet.

9. Conclusion

Several protocols make up electronic mail as we know it, each acting one or more of the set of roles to be played. Technology related to electronic mail, and a slew of problems

surrounding electronic mail of both technological and non-technological nature have been discussed.

References

[1] D. Balenson, *Privacy enhancement for internet electronic mail: Part iii: Algorithms, modes, and identifiers*, http://www.ietf.org/rfc/rfc1423.txt (1993).

[2] P. Beertema, *USSR on Usenet*, http://groups.google.com/group/eunet.general/msg/cf080ae70583a625.

[3] D.J. Bernstein, *qmail: the Internet's MTA of choice*, http://cr.yp.to/qmail.html.

[4] M. Butler, J. Postel, D. Chase, J. Goldberger and J. Reynolds, *Post Office Protocol: Version 2*, http://www.ietf.org/rfc/rfc0937.txt (1985).

[5] J. Callas, L. Donnerhacke, H. Finney and R. Thayer, *OpenPGP Message Format*, http://www.ietf.org/rfc/rfc2440.txt (1998).

[6] R. Castello, *SquirrelMail – Webmail for Nuts!*, http://www.squirrelmail.org/.

[7] M. Crispin, *Internet Message Access Protocol – Version 4rev1*, http://www.ietf.org/rfc/rfc3501.txt (2003).

[8] S. Crocker, N. Freed, J. Galvin and S. Murphy, *Mime Object Security Services*, http://www.ietf.org/rfc/rfc1848.txt (1995).

[9] D. Davis, *Defective Sign Encrypt in S/MIME, PKCS7, MOSS, PEM, PGP, and XML*, http://citeseer.ist.psu.edu/443200.html.

[10] T. Dierks and C. Allen, *The TLS protocol version 1.0*, http://www.ietf.org/rfc/rfc2246.txt (1999).

[11] M. Elkins, D.D. Torto, R. Levien and T. Roessler, *MIME Security with OpenPGP*, http://www.ietf.org/rfc/rfc3156.txt (2001).

[12] N. Freed and N. Borenstein, *Multipurpose Internet Mail Extensions (MIME) part one: Format of internet message bodies*, http://www.ietf.org/rfc/rfc2045.txt (1996).

[13] N. Freed and N. Borenstein, *Multipurpose Internet Mail Extensions (MIME) part two: Media types*, http://www.ietf.org/rfc/rfc2046.txt (1996).

[14] N. Freed and N. Borenstein, *Multipurpose Internet Mail Extensions (MIME) part five: Conformance criteria and examples*, http://www.ietf.org/rfc/rfc2049.txt (1996).

[15] N. Freed, J. Klensin and J. Postel, *Multipurpose Internet Mail Extensions (MIME) part four: Registration procedures*, http://www.ietf.org/rfc/rfc2048.txt (1996).

[16] R. Gellens and J. Klensin, *Message Submission*, http://www.ietf.org/rfc/rfc2476.txt (1998).

[17] J. Haight, *Spamcop.net – Beware of cheap imitations*, http://www.spamcop.net/.

[18] P. Hoffman, *SMTP Service Extension for Secure SMTP over TLS*, http://www.ietf.org/rfc/rfc2487.txt (1999).

[19] R. Housley, *Cryptographic Message Syntax (CMS)*, http://www.ietf.org/rfc/rfc3852.txt (2004).

[20] IANA, *IANA home page*, http://www.iana.org/.

[21] B. Kaliski, *Privacy Enhancement for Internet Electronic Mail: Part IV: Key Certification and Related Services*, http://www.ietf.org/rfc/rfc1424.txt (1993).

[22] S. Kent, *Privacy Enhancement for Internet Electronic Mail: Part II: Certificate-Based Key Management*, http://www.ietf.org/rfc/rfc1422.txt (1993).

[23] J. Klensin, *Simple Mail Transfer Protocol*, http://www.ietf.org/rfc/rfc2821.txt (2001).

[24] J. Klensin, N. Freed, M. Rose, E. Stefferud and D. Crocker, *SMTP Service Extensions*, http://www.ietf.org/rfc/rfc1869.txt (1995).

[25] W. Koch et al., *The GNU Privacy Guard – gnupg.org*, http://www.gnupg.org/.

[26] D. Knuth, *Email (let's drop the hyphen)*, http://www-cs-faculty.stanford.edu/~knuth/email.html.

[27] J. Linn, *Privacy Enhancement for Internet Electronic Mail: Part I: Message Encryption and Authentication Procedures*, http://www.ietf.org/rfc/rfc1421.txt (1993).

[28] Microsoft, *Microsoft Research Home*, http://research.microsoft.com/.

[29] K. Moore, *MIME (Multipurpose Internet Mail Extensions) part three: Message header extensions for non-ascii text*, http://www.ietf.org/rfc/rfc2047.txt (1996).

[30] K. Moore, *SMTP Service Extension for Delivery Status Notifications*, http://www.ietf.org/rfc/rfc1891.txt (1996).

Subject Index