ALVA L. COUCH

system administration thermodynamics

Alva Couch is an Associate Professor of Computer Science at Tufts University, where he and his students study the theory and practice of network and system administration. He served as Program Chair of LISA '02 and was a recipient of the 2003 SAGE Outstanding Achievement Award for contributions to the theory of system administration. He currently serves as Secretary of the USENIX Board of Directors.

couch@cs.tufts.edu

VIRTUALIZATION PROVIDES SEVERAL ways to transform the question "Why does this fail?" into the related question "Is this fast enough?"

Fellow system administrators, do you find yourself troubleshooting systems more and enjoying it less? Do you spend most of your time correcting the "same old problems"? Are legacy systems millstones around your neck? Then, from what I can tell, you are like most system administrators. For those of you in this situation, I have a controversial message: *The troubleshooting you are doing now is already obsolete.*

In the following, I will outline techniques for minimizing common kinds of trouble by use of virtualization. Most of these techniques are common knowledge, and I apologize in advance for stating the obvious. But, in my experience, many system administrators of good faith and stronger character than my own still endure these various tribulations. This article is written for them because I think they remain in the majority.

No strategy I am going to suggest actually eliminates trouble. Instead, trouble is *transformed* into a hopefully more manageable form. Virtualization allows one to replace configuration troubleshooting with performance troubleshooting. One key to understanding this transformation is to consider it as part of the "thermodynamics of system administration." System administrators, like mechanical engineers and physicists, have to cope with conservation laws, and one thing that is conserved is trouble. We cannot eliminate trouble, but we can make choices that transform it into a perhaps more manageable (and hopefully "user-friendly") form.

The Three Laws of Thermodynamics

Trouble is a form of entropy, and thus it is subject to the laws of thermodynamics. Ginsberg once described the three laws of thermodynamics as "One can't win, one can't break even, and one can't get out of the game." In system administration terms, we might restate these laws as follows:

- There is no way to prevent trouble.
- There is no zero-cost way of transforming trouble into other forms.
- Trouble approaches zero only as system use approaches zero.

The theme of this article is the second law. In system administration, as in thermodynamics, one

can, by applying some energy, transform trouble to a (hopefully) "more convenient" form.

As a physical analogy, suppose that you are careening down a hill at high velocity toward an obstacle. To mitigate this, you can apply a brake, but the action of applying a brake has its own problems, including heat buildup. Your action can transform the problem of careening down the hill into the problem of controlling the heat from a brake, but it helps to know how to handle heat from the brake before applying it!

In the same way that brakes convert velocity to heat, I will outline several techniques for transforming configuration issues into performance issues. I believe that the most powerful tool the system administrator has for dealing with trouble is *architectural design*. The proactive system administrator strives to make trouble easier to handle by employing virtualization to *limit the forms in which trouble arises*. When one changes the form of trouble, one may need new skills to mitigate new kinds of trouble. But if one designs cleverly, the total time one actually spends, the amount of downtime, and the knowledge needed to cope can all be dramatically reduced.

Minimizing Coupling and Maximizing Cohesion

Our first two steps borrow principles from software engineering. A good architectural design minimizes coupling and maximizes cohesion [1]. Two components are coupled to the extent that they interact; couplings correspond to "things to remember." By contrast, a cohesive component groups related functions together inside one entity.

Unnecessary coupling is a major cause of troubleshooting and maintenance cost. Examples of coupling problems include version skew in libraries and/or packages, disagreement between two parameter values that should agree, or conflicting (and thus impossible) requirements for assuring the function of two co-resident applications.

As a trivial example, it is impossible to install both php4 and php5 Apache modules at the same time. Such dependencies arise from application requirements (e.g., one php4 application and one php5 application that are intended to execute on the same physical server).

The main trick I will use to transform trouble is to trade performance problems for combinatorial problems. A "combinatorial problem" is an error in how software is configured or how it interacts with other software, whereas a "performance problem" is a situation in which an operation executes properly but perhaps more slowly than might be desired.

For example, one can solve the php4/php5 problem by creating two virtual operating system instances, each running its own Apache server. One server includes php4 and the other includes php5. The illusion that both are running on the same machine can be maintained by making the original machine into a proxy server.

Segregating services onto distinct components changes the kind of trouble that can arise for the services. If they are running on separate servers (either through physical or virtual separation), then the services are prevented from interacting in ways that co-located services can, so there is absolutely no problem in supporting php4 on one instance and php5 on the other. But we may have to maintain, by other means, the illusion that the applications execute on the same server (e.g., by some form of service switching). One form of complexity replaces another.

It is possible, though, to minimize coupling too much. One should also strive for *cohesion*. Two services are cohesive if they interact with a shared information domain. For example, putting DNS and DHCP on the same server is (usually) cohesive because both pertain to IP, but co-locating DHCP and a Web server is (usually) not cohesive, because the information domains of the two services (usually) overlap very little.

The concepts of coupling and cohesion are borrowed from software engineering but the justifications are perhaps even stronger for system administration. In software engineering, coupling between program modules leads to a need for increased *communication between module authors,* which delays software development. In system administration, coupling between components leads to a need for increased *knowledge* on the part of the individual administrator trying to make them work together, which means more time spent in initial setup and in troubleshooting the interacting components.

Through use of virtualization, *dependency troubleshooting of co-located services is an obsolete skill,* because two software packages that implement services can be positioned within different (virtual) platforms that cannot "depend" on one another. The whole process of installing an instance becomes centered on one application and its needs. But in the latter case, a new form of trouble can arise, in the form of *resource dependencies* (e.g., shortage of CPU cycles or I/O bandwidth among two or more instances). These dependencies cannot break an application, but they can cause it to execute unexpectedly slowly. We do not eliminate trouble; we merely transform its nature.

One side effect of using virtualization is that some of the complexities of configuration management are also obsolete. One thing that makes configuration management difficult is change. In a virtual environment, one can often afford to build a new server instance while existing server instances are live, so that one can start afresh whenever a change is needed. This mitigates several kinds of configuration management problems.

Exploiting Social Pressure

A second design guideline is so obvious that many of us might forget it. Software cannot ever be completely tested. Therefore, it makes sense to design one's systems around software environments that others have aggressively utilized and tested, because each application is more likely to have been thoroughly debugged for those environments than for others. In particular, bugs resulting from configuration problems (e.g., hidden dependencies) are much less likely to arise in commonly deployed environments. The simple reason for this is *social pressure;* the widespread use of a particular environment means that most bugs for that environment will be discovered, reported, and, hopefully, repaired. The most common environments for an application thus naturally become the most tested and functional, because there is a higher incentive for developers to address the bugs with the widest social exposure. Thus, it is typically much more likely that an application will run properly in a vanilla environment (e.g., the default installation of a Linux distribution) than in a customized one. If problems do arise, it is more likely that others have seen them before and have already found and published work-arounds.

By using virtualization one can arrange, much more easily than ever before, for an application's environment to be the one with the greatest social footprint, because the environment for each application can be chosen independently.

Horror stories about failing to exploit social pressure abound. One should not adopt software on the bleeding edge unless one expects to bleed along with it.

For example, our site was one of the first adopters of Sun's NIS+ directory service, because it had many neat features we wanted. Unfortunately, it also had many painful bugs we did not want. What we did not understand or account for at the outset of this project was the power of social pressure. NIS+'s deployment footprint never became large enough for the bugs to be addressed (we heard later that Sun had not used it internally), and we replaced NIS+ with LDAP before the problems we encountered were resolved. By contrast, by possessing an enormous and multi-platform social footprint, LDAP has been pounded upon by a large number of conscientious users and has thus been forged by social pressure into a reliable tool.

This is an object lesson in the danger of creativity. Becoming a follower rather than a leader often involves less pain and suffering. This principle takes many forms, from avoiding first adoption of a tool to avoiding being the first to apply a new security patch [2].

It may seem obvious that our jobs as system administrators do not involve making developers fix their bugs but, instead, require us to provide mechanisms for getting useful work done in the presence of those bugs. Although we file bug reports as a public service, no system administrator can reasonably expect a user to wait for a bug fix. We are instead the masters of the work-around, not the masters of the software, and if anyone has managed to make it work, we are expected to know exactly how and why. Again, virtualization allows us to synthesize almost any software environment needed by an application, without breaking any other one.

Softening Hard Boundaries

A third trick in the contemporary system administrator's arsenal is to use virtualization to control which attributes of a network are "hard" and which are "soft." A *hard attribute* is an attribute of a system that can only be controlled by a human being, such as the physical location of a machine or the location of the access point to which it binds. A *soft attribute* is one that can be manipulated by setting values of parameters via software and/or automation.

The easiest example of hard and soft boundaries involves the computing power of servers. In a non-virtualized environment, the amount of computing power available to a service is a hard attribute, whereas in a virtualized environment it can be considered a soft attribute (e.g., a configuration parameter of the hypervisor). As another example, virtual LANs make the network to which a host is connected a soft attribute, whereas in non-virtual LANs this is a hard attribute.

The overall purpose of softening a hard boundary is to turn a decision whose implementation requires major work into one requiring setting parameters. For example, consider the example above for php4 and php5. If the two applications are installed on two servers, then changing the response time for one application requires rebuilding the service on another server, but if the applications are virtual instances on one server, changing response time can be expressed as a parameter change in the hypervisor.

In both of these cases, softening does not eliminate entropy; rather, it transforms it into a new form. Even the very best virtualization strategies exact a performance penalty, because sharing resources among more than one operating system takes time (thus invoking the second law).

Edge Cases

Alas, there are always cases in which one cannot straightforwardly eliminate troubleshooting of combinatorial problems. Mostly this is because the user explicitly requires several conflicting services to be co-located on the same device, such as a workstation. Then we are faced with the same old combinatorial problems. What to do?

Fortunately, there are several evolving approaches to this problem, all involving advanced forms of virtualization that the system administrator controls. Operating systems do not represent the only grain at which virtualization can function; one can also virtualize file access, registry access, or library access for different applications running within one operating system instance. Some visionaries in the virtualization community believe one will be able to routinely virtualize the software environment for each application without virtualizing the underlying operating system. The net result of this strategy is the same as before but is much lighter in weight; virtualizing the "open" call has a much lower overhead than virtualizing the whole operating system.

For example, IBM's prototype Progressive Deployment System (PDS) [3] virtualizes library and registry access in Windows without virtualizing the whole operating system. Each application thus executes in a custom environment in which registry or library conflicts cannot occur. This is done without virtualizing the whole operating system, which makes it much less resource-intensive to use.

Understanding Resource Contention

In all of the examples cited so far, we have transformed entropy arising from combinatorial conflicts into entropy arising from resource conflicts. In the first case, we traded speed for combinatorial complexity, preferring a simpler, slower solution to a faster, more complex one. In the second case, we traded customizability for robustness, preferring a mainstream, well-understood solution to a perhaps more customized but less-tested option. In the third case, we traded space and time for flexibility, preferring to control state via software rather than by rebuilding servers. The good news is that a few common forms of system failure, including downtime from configuration conflicts, are "virtually" eliminated.

But in system administration, as in thermodynamics, entropy remains. We have only changed the way it can be expressed. We have ensured that the various and sundry state machines making up our applications have the configurations and environmental conditions that they need to react correctly, but not that resources that they need will be available when they need them.

Addressing resource conflicts is a very different form of troubleshooting from those most system administrators are used to. Resource contention is a "quiet" kind of failure; systems fail "not with a bang, but with a whimper." Failures are subtle and sometimes nearly unnoticeable.

But there is also a subtle value shift involved. Virtualization has explicit performance penalties. In eliminating combinatorial issues, we have already departed from the old rubric of making systems function "as fast as possible," and we are forced to ask ourselves some difficult questions about what performance is "good enough." Once we know what is "good enough," we can ask ourselves the second question, "What changes will provide performance that meets that standard?"

What Is Acceptable Performance?

Old habits die hard. Most of us are used to squeezing the maximum possible performance out of our systems, so that the question of what is appropriate performance never arises. When we use virtualization tricks to invoke independence, social pressure, and softness, we trade optimal performance for robustness. Obviously, it is possible to trade away more performance than is reasonable. But what is "too much to trade"?

First, we need some reasonable definition of what performance actually means. There are several possible definitions, all involving some concept of *response time*. For a Web site, response time is the time it takes from when you send a request to when you receive content. For a shell, response time refers to the time between a key press and the associated change in screen state. In a batch environment (e.g., accounts payable), response time is the elapsed time between job submission and job completion.

Second, we need some way of measuring performance. There are many mechanisms, both direct and indirect. A direct performance measure quantifies what the user sees, whereas an indirect measure is related to, but is not exactly equivalent to, the user's experience. Direct measures include benchmarking and soliciting user feedback; indirect measures include server load, memory utilization, etc. The latter are functionally related to what the user sees, but the relationship is not (usually) easy to describe. For example, we agree that servers with high load averages are "bad," but "just how high is bad" depends upon what the user experiences, and not necessarily what the system administrator sees in the logs.

SLOs and SLAs

The next step is to define *acceptable* performance. Here we can borrow some terms from autonomic computing and outsourcing. A "Service Level Objective" (SLO) is a definition of what directly measured performance is "good enough." This is usually specified in very high-level terms, as end-to-end response time (e.g., "Users should obtain a response from the Web site within one second"). SLOs are determined by economic analysis of the business effects of service delays. For example, a few seconds of delay may be catastrophic for online stock trading, and it is generally accepted that response delays in online shopping lead to lost sales.

An SLO may also set different goals for each kind of service or each kind of client. It is common to refer to clients as "gold," "silver," or "bronze" to denote priorities for performance. For example, in a hospital, doctors need "gold" service levels, but for staff not involved in patient care (e.g., billing personnel), "bronze" response suffices. In the emergency room, an even higher "platinum" service level may be needed.

SLOs can also embody business strategy. Some analysts believe that in a sales situation, *it is better not to respond at all than to respond slowly*. Giving up on customers who have already waited too long diverts computational resources away from customers who represent lower sales potential, to customers who have not yet been made to wait (and thus represent higher sales potential). Such a strategy is sometimes called an *admission control policy*, because one only "admits" customers to one's site that one has the resources to serve in a timely manner and tells the customers whom one expects to experience long response delays to come back later (because, statistically, if one does admit them, they are likely to leave before buying anyway) .

By contrast with an SLO, a Service-Level Agreement (SLA) defines not only desirable objectives but also penalties and incentives in interacting with

some external client. SLAs are common in defining expectations between a business and a hosting service. Whereas an SLO might say, "Response time should be less than one second," an SLA might add, "Response times over one second will be billed to the provider at one cent per instance" or "The provider will be paid one cent more for each request whose response time is less than one-half second." Incentives often vary for different kinds of clients.

The typical use of an SLA is to define interactions between autonomous service providers, but system administrators can utilize the concept as a way of describing service requirements between themselves and their organizations. In this case:

- A service objective is the minimum performance required by management.
- A service penalty (for not meeting the objective) or incentive (for exceeding the objective) can be interpreted in a human context (e.g., raises and promotions).

The Hard Question

In moving into a new territory it helps to understand the objectives and incentives. What is your SLA as a system administrator? If your site is a development site, there may not be a strong business reason for quick response time, so your SLO expectations may be low and your SLA may be undemanding, whereas fluidity and deployment agility may be very important instead. If your site engages in financial trading, there may be sound business reasons for your SLO to include high minimum expectations and high penalties for delays.

This can be a very difficult question to answer, because most managers may not ever have thought about system administration in this way and may not be aware of the thermodynamic principles (as outlined in this article) that give system administrators a *choice* between manageability and performance.

A New World

Virtualization gives us new choices. The profound impact of those choices is to trade one property of a system for another. This allows us to architect systems for robust behavior, effective automation, and autonomic control. But to reap the benefits, we cannot tune a system to run "as fast as possible," and such an objective is now rather meaningless. The job of system administrator has changed, from doing "whatever it takes to make it work," to making choices that are "good enough." Before, we were thinking about cost; now it is time to concentrate on value.

REFERENCES

[1] Roger S. Pressman, "Design Engineering," in *Software Engineering: A Practitioner's Approach,* 6th ed. (New York: McGraw-Hill, 2004), chapter 9.

[2] Steve Beattie, Seth Arnold, Crispin Cowan, Perry Wagle, Chris Wright, and Adam Shostack, "Timing the Application of Security Patches for Optimal Uptime," *Proc. LISA '02*, USENIX Association, 2002.

[3] Bowen Alpern, Joshua Auerbach, Vasanth Bala, Thomas Frauenhofer, Todd Mummert, and Michael Pigott, "PDS: A Virtual Execution Environment for Software Deployment," *Proceedings of the 1st International Conference on Virtual Execution Environments (VEE '05),* ACM Press, 2005.