MARK BURGESS

configuration management: models and myths

PART 3: A SHOCKING LACK OF AD-HOCRACY

Mark Burgess is professor of network and system administration at Oslo University College, Norway. He is the author of cfengine and many books and research papers on system administration.

Mark.Burgess@iu.hio.no

IN HIS 1970 BESTSELLING BOOK *Future Shock* [1], writer Alvin Toffler predicted the demise of bureaucracy. Toffler was a writer emerging from the 1960s, on the tail end of the hippie revolution. They were going to make the world right, optimism was in the air, and everyone saw the pace of technological change as a force for good. Today, we are less enamoured by progress and have fallen back into a stagnant economic tumble-drier of selling and consuming that seems to have no vision or direction. Perhaps that is why Toffler's vision of the demise of bureaucracy never really came about.

Toffler predicted that centralized power structures, with their rigid procedures for decision-making and management, designed for a slower age, an age of little change, would collapse under their own sluggishness—buckling under the force of a cultural and technological deluge. Bureaucracies would be replaced by lean, mean decision machines, guided by simple principles, and so agile that they would win over traditional leviathans, much like mammals sticking out their tongues at the sauropods. Moreover, people like me, working in government organizations, would be freed from the slavery of application-report-archive to live productive lives full of choice and measured reflection. He called this state of affairs ad-hocracy. Toffler wrote:

> Faced by relatively routine problems, [Man] was encouraged to seek routine answers. Unorthodoxy, creativity, venturesomeness were discouraged . . . rather than occupying a permanent, cleanly-defined slot and performing mindless routine tasks in response to orders from above, [Man] must [now] assume decision-making responsibility—and must do so within a kaleidoscopically changing organizational structure built upon highly transient human relationships.

That is what Toffler said about the human workplace in 1970. This well-meaning sermon has admittedly not taken the human world by any great storm, as we attest from experience (though, if we are being fair, it has indeed made inroads). What I find ironic is that we are now reliving an almost identical discussion in a different sphere. Today, we are struggling to accept the same wis-

dom in the area of computer management. It will take the next two parts of this series to do this subject justice.

Strategies in the War Against Tera

What is a good strategy or algorithm for computer management? Few would argue against the idea that the sheer size of systems today practically necessitates automated tools. (Recall Ken's law: Always let your tool do the work.) Certainly I believed this in 1993 when I started writing cfengine, and today IBM certainly believes it and flags it with its Autonomic Computing initiative. Toffler pointed out that automation does not necessitate production-line thinking, in which one mass-produced identical copies—a world in which one can have any color as long as it's black. On the contrary, he argued that "As technology becomes more sophisticated, the cost of introducing variations declines."

But in the management of the information technology itself, we are still hearing about "ways to mass-produce 1000 workstations, all identical, from a common source"—golden master servers that are to be worshipped by hundreds, perhaps thousands, of clones. Ad-hocracy is not the default doctrine in computer administration.

Ever since the late 1980s, the telecommunications companies have had their own vision of computer resource management, borrowing from tried and trusted inventory systems, for warehouse and personnel management, and trying to modify them to cope with the computing age. In time they borrowed ideas from software engineering (e.g., object-oriented database models).

Industry standards organizations such as the renamed Telemanagement Forum (TMF) and Internet Engineering Task Force (IETF) have continued to develop models for managing computing equipment that are essentially bureaucratic. What they perhaps failed to anticipate was the pace at which the technology would develop (something akin to the rate at which device drivers have to be written on PCs). Trying to keep up with the schema-centric definitions for all new products has led to a classic "Tortoise versus Achilles" race between the development of new technology and the struggle to document the growing zoological inventory. (Cisco's IOS is surely the winner of this race.)

For the telecoms, Operational Support Systems (OSS) and Business Support Systems (BSS) were the order of the day. The idea was simply to document every device and human procedure exhaustively in a huge database so that help-desk staff would be able to see an overview. Later came tools that could interact with the devices via a "management console" in order to write certain values to routers and switches, and even to workstations and PCs. Today, the legacy of these approaches is still with us; they still cling to life, even today in the largest corporations, but they are still wailing (or yawning) from their tar pits.

The complexity of those systems is legendary. No sane engineer, in his or her right GHz CPU, would seriously try to build such a monster. Yet, in the wake of these support systems, designed for the telephone era, the same knowledge engineers attempted to create the new generation of forms and processes that would manage the computing age. Among today's species:

- *SNMP/MIB:* A hierarchical table-based data structure (the Management Information Base) that is mapped into a linear set of machine-readable

identifiers. The values associated with these identifiers are simply read or pushed into place by the SNMP read/write protocol. The algorithmic complexity is very low. The data complexity is a simple regular approximation to a context-free language.

- *SID:* Shared Information and Data model. This is an information model that is used in both NGOSS and DEN-ng. It includes services and organizational containers in an object-oriented framework.
- *CIM:* Common Information Model. An information model that provides an exhaustive replacement for MIB.
- *NGOSS:* The TMF describes this as a "comprehensive, integrated framework for developing, procuring and deploying operational and business support systems and software." It includes the SID and eTOM standards. It is a complete organization map.
- *DEN(-ng):* Directory Enabled Networks. This is a model that is complementary to SID. It focuses on modeling network elements and services, using an interpretation of policy-based management. DEN-ng products and locations are subsets of the SID.

I challenge readers to look up the data models on the Web to see just how complex they truly are. The DEN-ng and SID initiatives are trying to move away from a MIB-like catalog of device attributes to an overview of an organization and its resources. In particular, the notion of services is an important addition.

Even equipped with these big guns for pattern description, and having the most eager blue-collar beavers to register all of this information, the efforts of these engineers ultimately seem to have fallen on deaf ears. No one really seems to want these systems—not even their key designers. Why? As the Soviet Union or European Union or even the State of the Union will testify, bureaucracy is just too expensive.

What's on the Yellow Brick Road?

The data models mentioned here have sufficient linguistic complexity to describe the patterns we would expect to manage in an organization, just as we predicted in the last issue's episode, but something is wrong. Toffler's warning is ringing in our ears. We seem to be missing a vital part of the story. Configuration management is not merely about brick-laying and form-filling.

Configuration management (a pretty low-level animal in the administrative phylogeny) has become the topic de jour in the UNIX world, perhaps because it is a technological problem, which tech folks love. But it is not the beginning or end of any story that we really care about. We have no real interest in what the configuration of a system looks like. What we really care about is how to represent the goals of our organizations and applications using patterns in order to lead to a predictable pattern of behaviors. This leads us to a hypothesis, which, as far as I know, has not been convincingly proven:

> Hypothesis: There is a direct association between a "correctly configured computer" and a "correctly behaving computer," where "correct" means "policy or specification compliant."

The essence of this hypothesis is shown in Figure 1. It is not just a matter of configuring a computer but one of solving the problem of achieving the correct behavior. Configuration is a static thing; behavior is a dynamic consequence, but not a fully predictable one.

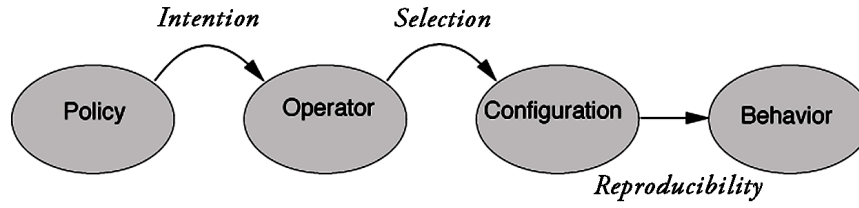

FIGURE 1: THE STAGES FROM POLICY TO BEHAVIOR. A STORY QUICKLY FORGOTTEN?

There are three parts to the story depicted in Figure 1::

- Planning the intended behavioral policies for all parts of a system.
- Mapping this to a configuration that can lead to the correct behavior.
- Implementing the change in configuration reliably.

How do we know that we can complete this manifesto? Is it doable? If so, can it be done reliably? Well, in 2003, I proved a limited version of this hypothesis [2], showing only that it is possible to define the meaning of "policy" in terms of configuration changes so as to lead to predictable behavior on average. This is not quite the same as the hypothesis posed here; what it says is that there a restricted language that maps directly to behavioral consequences, so if we restrict ourselves to that, we are okay. The part about "on average" is general, and it says that no configuration management scheme can guarantee that a host will always be correctly configured, unless the machine is never used.

You Say Tomato and I Say . . . Semantics

According to the first two parts of this series, we have a reasonable account of how to manage patterns of data (with or without the monstrous data models that pepper the procedures with structural complexity). These procedures might be messy, but they are essentially just bureaucratic spaghetti, somewhat irrelevant to the deeper issues.

If we fix a bit string, such as a file mode, using a numerical value, there is little ambiguity in the procedure. It seems like a straightforward problem to write some configuration to a computing device: This is like stamping out molds from a production line (e.g., chmod 755 filename). It is straightforward, easy, and not complex—much like SNMP without MIB. Any complexity lies in the patterns themselves—in the coding of the instructions, and in understanding what the behavioral consequences of these changes are. Thus, this is not where the problem lies.

But now consider the expression of policy itself. If we wish to describe an operation in terms of a high-level procedure (e.g., "InstallPackage(ssh)") then this is no longer straightforward, because it is describing the configuration coding only at a medium level, not all the way down to the bits. This is like saying, "Make me prettier!" It is not a uniquely defined or reproducible goal. Someone might say that it is their policy to make you prettier, but you cannot guarantee their behavior from this assertion. (You might trust them more, if they told you about what end result they were going to guarantee—see the following.) If we take only a shell of patterns such as InstallPackage, there can be several (even many) nonequivalent ways of defining the internal procedures within the language of the low-level configuration. Consider the following two interpretations of an InstallPackage command, which are inspired by real examples:

```
InstallPackage(foo)
        Check dependencies
        Check if package README exists
         if (!exists)
                copy package
                unpack
                run local script

InstallPackage(foo)
        Check if existing binary is executable
        if (!exists_and_exectuable)
                Check dependencies
                Copy packages
                unpack all
                copy files to /usr/bin
```

The resulting patterns are described and implemented in terms of language syntax, as we have already noted, and computing is obsessed by syntax today—but if the complete syntax is missing from the explanation, the call for InstallPackage is meaningless. Several of the big data models mentioned here boast a specification written in the Unified Modeling Language (UML), which is based on an object-oriented syntax (i.e., hierarchical class structures). Thus it is fundamentally built as a bureaucracy of types. Moreover, XML has become the bureaucratic memo-paper of choice. XML is no more than an empty syntax "desperately seeking semantics."

This is pretty much what happens in configuration management tools. By attempting to be user-friendly and high-level, many configuration tools sacrifice operational clarity for human readability. Trying to define configuration in terms of such vague high-level precepts is like trying to tell a story like this:

> A man (motion-verb) into a (drinking-place-noun) and (communication-verb) a drink.

We can fill in many alternatives that lead to grammatically correct sentences, i.e., which obey a pattern language that is recognized by our system. But the patterns all mean quite different things, or perhaps nothing at all. There is no clear way to say that what we meant was "a man walks into a bar."

If we are to successfully govern systems, either externally or autonomically, we need to be able to complete the chain from top-level goals, to a clear and reproducible set of operations, to a definite configuration that leads to predictable behavior. This is not an impossible task, but it is far from guaranteed.

How to Say What You Mean

At the 2001 cfengine workshop (later followed up by Paul Anderson and opened to a wider community, becoming the configuration management workshop), a discussion almost became an argument. My friend Steve Traugott, bless him, told me I was wrong. Thunderclaps sounded, screams were heard. Tempers were enraged. In the meantime, Alva Couch and I were quietly interested in Steve's point as others were doing battle over it. I thought, "Clearly, I was not wrong; I am surely never wrong," and yet Steve pressed his point, which has since been studied in detail by Alva Couch and which I have come to understand better as I have pondered the matter using different reasoning. Of course, neither of us was wrong, but, importantly, something was learned.

The matter concerned two design strategies that have been discussed for constructing configuration management schemes:

- We specify the final state and leave it up to the program to figure out the details of getting there.
- We specify the starting point and a specific program of steps to take.

For reasons we won't go into yet, these were labeled "convergence" and "congruence," respectively. To borrow Alva Couch's terminology, we can rather refer to these as precondition-based and postcondition-based specifications.

Ultimately, I believe that the first of these is preferable for a number of reasons, including parsimony, consistency, and aesthetics (stay tuned), but the real difficulties associated with configuration management are present in both cases. They cannot be avoided simply by choosing one.

In both cases there is the matter of how it is possible to change from the old state to the new state. Suppose a computing device is in a state that is not consistent with policy. We require a procedure, whether that means a static bureaucratic procedure or a lean-mean entrepreneur procedure, to fix it. In the first case (postcondition), we define this procedure generically, like a template, once and for all (i.e., we define what we want to get out of "make me prettier"). In the latter, we define the procedure in each case, making it potentially inconsistent. Steve said we can still achieve consistency by always starting from a known state and following a precise chain of preconditioned actions, meaning that if a computer gets messed up, you wipe it clean and start over. This is a reasonable approach to take if one thinks in production-line terms about configuration management, but this is not my vision.

Mass Production Undone

Production-line factory thinking requires a chain of preconditions. When you create a chain of operations that depends on previous operations, each step is preconditioned on what came before. If one step fails to be implemented, all subsequent steps fail (e.g., "I'm sorry, sir; I can't make you prettier; your nose is in the way.").

This is fair enough—we just have to figure out how to get it right without getting stuck. That might be possible, but in fact it is harder than in the postconditional case, because the compositional complexity of the approach has to be dealt with in one go, whereas it only has to be dealt with for one operation with postconditions. But the real problem with preconditions is that the approach fails to easily support a wide variety of different adaptations. It takes us back to Toffler's fear of the totalitarian-commie nightmare of mass production of a single unvarying model.

Oddly, in system administration, many still worship the totalitarian gods of mass production. The god of small things, to paraphrase Arundati Roy, is still being trampled by the heavy boots of bureaucratic thinking.

Suppose we assume that the postcondition model is possible (cfengine uses this approach, so it works at least in some limited capacity). Then we can (at least try to) never base an operation on something that came before. Then the order no longer matters, and only the final state is significant. Now, although this approach is achievable, in principle, it is also beset with problems. Its chief selling points are:

- Consistent semantics.
- Specification of the final state is often simpler than specification of the steps needed to get there.

- You do not have to wipe out a machine if something goes wrong; the system can adapt in real time.

Its main problem is a residual ordering ambiguity caused by creation and deletion and competitive adaptation.

Black Boxes and Closures

The inner workings of bureaucracy are generally opaque, but for reliable administration this is not necessarily a bad thing. Black boxes are a mainstay in computing because they hide inner complexity and also protect inner details from outside corruption.

As Alva Couch and his students have pointed out, the computer science black-box notion of closure gives us a level of predictability, by locking out the environment that generally confounds predictability. This is the same environment that can screw up chains of preconditions, as Alva's work has taken some pains to model in detail. The trouble is that, although closure is easily implemented for things such as database transactions, it is quite difficult to implement in the area of system administration, because systems are constantly being exposed to the environment by uncontrollable backdoors. Moreover, they often share an operational state (routing tables, databases, etc.) that breaks open closures.

The story of order-independent operations is also rather nontrivial and is based on a very low-level approach to operational semantics. With cfengine, the focus has been on this approach (some think too much so), and hence it often fails to provide higher-level expressivity, which other projects such as Luke Kanies' Puppet are trying to remedy (hopefully keeping the low-level semantics intact). Paul Anderson has long told me that he sees cfengine as a low-level language that one compiles down to. This seems sensible to me. In the meantime, together with Alva Couch, I am developing a more precise theoretical model for these low-level semantics that will eventually be incorporated into cfengine 3.

Even if a configuration is reachable without any ordering problems, there are some features of behavior that depend on the order. This has to do with the fact that creation and deletion are catastrophic state-destroying operations that break commutativity on present-day operating systems. It is conceivable that one could build an operating system that did not have this property, but it would be quite difficult. A fair approximation would not be too hard to build, however, so we could have commuting operations and the order of procedures would be entirely irrelevant to the final state.

The King Is Dead: Long Live the Laissez-Faire Army

Humans beings have a remarkable capacity to view the world in terms of subordination, and system administrators are no exception. You'd think we'd all done military service or were trying to establish ourselves as king or emperor by conquering fourth-world tribes of disorganized computers and sending them for Pygmalion execution lessons on how to behave in the Kingdom of the Data Center.

In the 1990s, as telephonic empires were crumbling, small-business entrepreneurship invaded this turf and took computer management in a different direction. Small furry businesses started making it up as they went along, thanks to tools such as small computers, UNIX, and ad hoc solutions of Windows and Macintosh. With an excitement for progress rather than control, mammals evolved and dinosaurs were left floundering. The

UNIX world has bothered itself little with the data models mentioned here: cfengine, Isconf, LCFG, and of course every site's home-brew scripts have been much more ad hoc in their approaches—with almost devil-may-care informality. And yet they work. Why?

In *Future Shock*, Toffler related an important insight, an insight that it is appropriate for us to relearn. His point was this: In the 1960s, as we remained scared of the looming presence of communism, it was assumed that the industrial age and the rise of technology meant a future that was mass-produced, in which everything was the same—there was no variation and no choice, just an overwhelming amount of factory produce, because the duplication of fixed pattern was marching to the tune and beat of industrial nations' sternest baton.

What Toffler realized was that better technology allows one to manage more variety, greater diversity, and, importantly, greater choice. We do not have to fear diversity. What, after all, is the point of information technology if not to manage the complex array of specially tailored blueprints? What is the reason for improving management of productivity if not to cater for the whims and desires of minorities and special interests?

The weight of bureaucratic constraints just to maintain a large information model is overwhelming. It is too slow. If you are attached to a fleet of steel balls by a cat's cradle of elastic bands, your best career choice is not that of acrobat.

A Bearable Lightness of Being

There is a myth that, if you do not control something, the result will be chaos. There is a belief that, if you do control something, its behavior will be in accordance with your wishes.

I believe that there is some linguistic confusion at the heart of this debate. The word we want is not "control," because that is a word of hubris. You can tame a horse but you will never control it. There is a world of difference between control and management. Toffler pointed out the answers in 1970. We are fighting the wrong battles.

> Rising novelty renders irrelevant the traditional goals of our chief institutions. . . . Acceleration produces a faster turnover of goals. Diversity or fragmentation leads to a relentless multiplication of goals. Caught in this churning, goal-cluttered environment, we stagger, future shocked, from crisis to crisis, pursuing a welter of conflicting and self-cancelling purposes.

The real measure of intellectual achievement is to take something complex and make it simpler—by suitable abstraction. Anyone can make excruciating syntax, an exhaustive list, or a database of every possible detail. There is absolutely no evidence that tight bureaucracy leads to greater predictability. What can lead to predictability is clearer semantics—perhaps with a lighter touch.

In the next episode, I want to dispel a related myth: why centralization is not the necessity that has generally been implied.

REFERENCES

[1] *Future Shock*, A. Toffler (Random House, 1970).

[2] "On the Theory of System Administration," M. Burgess, *Science of Computer Programming* 49, 1–46, 2003.