Uncertain Infrastructures

MARK BURGESS

Mark Burgess is the CTO and Founder of CFEngine, formerly professor of Network and System Administration at Oslo University College, and the principal author of the Cfengine software. His the author of numerous books and papers on topics from physics, Network and System Administration, to fiction.
mark.burgess@cfengine.com

To our shame, one of the rarely voiced complaints one could level at the IT service industry is that we don't know how to make promises we can keep. In fact, we have not designed technology to keep anything; the main focus lies in building, tweaking, and fire-fighting, all within an increasingly fast-moving and disposable culture.

Against the backdrop of this uncertainty, we've invested in our reliance on technology. Smart devices enable and enhance our personal freedom in ways that are just too seductive to forego, and they interface with services that lie behind the scenes. In the new age of IT-powered commerce, the continuity of that lifestyle has come centre-stage. We used to talk about business continuity and disaster recovery, now we talk about continuous delivery of products to market, as well as continuous availability of services. We are starting to realize that the modern world is always on, and we will not accept anything less.

The builders and custodians of today's infrastructure have designed technology to be managed by direct commands or remote control, replacing manual error with amplified manual error. Errors are reported by independent monitors and shot down by further manual intervention: errors slain like dragons in a gaming experience. But this will not do for mission critical infrastructure.

Systems Thinking: Clockwork Uncertainty

We can really only promise a tiny number of things about the vastly complex environments we build. This does not give cause for complete certainty, but it can help to set expectations. We can promise certain aspects of behavior, albeit with margins for error, but we should also be clear: We build systems and essentially hope for the best; nothing we do can fully determines whether a system crosses a threshold into instability.

Over the years I've had the privilege to work with many smart people at installations of varying scale and complexity, often through the lens of CFEngine, and I've seen the issues first hand and been able to learn some lessons from them. There is growing recognition that systems are composed of both humans and automated processes, and that systemic complexity amplified by scale is the main cause of uncertainty. But few voices have invested in a science to understand and describe such complexity.

There are big ideas here, far too large to fit into this brief comment; so, I decided that it was time to write them into a book. In Search of Certainty: The Science of our Information Infrastructure is my new book [1], a popular science account of what I learned over the past 20 years.

Smart but Resilient

Infrastructure is getting smarter. Why? Because we want to get at stuff faster. The less work we have to do, the more accessible marketplaces for "stuff" are, the happier we seem. That means embedded computation.

References

[1] Burgess, Mark. In Search of Certainty: The Science of our Information Infrastructure. XtAxis Press, 2013. http://mark-burgess.org/certainty.html

[2] Gordon, J.E. The New Science of Strong Materials: Or Why We Don't Fall Through the Floor. Princeton University Press, 2006.

Uncertain Infrastructures

We've seen a lot of progressive thinking over the past few years under the banner of software defined systems. There has been talk about "anti-fragility," with chaos monkeys prodding systems to make them fail. Of course it is not the breaking of systems that makes them stronger, but adaptive processes behind the scenes that no one is really talking about. This is where the science lies.

The thrust of my activism in system administration has been that we have to start thinking more scientifically. Computer science is weak in the broad traditions of science as a tool of explanation. Its roots lie in deductive reasoning, which is only a small part of a post hoc picture. Managing systems is not just a case of test-based development, or of analyzing big data. Test-based development is like trying to pin the tail on a mule, unless you have some guiding principles and empirical foundations on which to home in on your design.

Semantics and Dynamics = Dev and Ops

I argue that there are two aspects to systems. I shall call these semantics and dynamics. Semantics are about purpose and intent. Dynamics are about behavior and performance. In some ways, these two aspects map on the dev (development) and ops (operations) in the the current parlance. Developers think mainly about purpose and intent, whereas operations engineers have to deal with actual behavior. DevOps tries to teach the message that you need to understand both of these aspects together in a unified way if you want to understand IT services beyond a trivial scale and complexity.

In fact, there is deep science here—and not just the signal lambda calculus that has gained the unfair attention of a small crowd of developers—the study of behavior is known to us as physics and it spans a plethora of different methods and issues. I don't have time to talk about them here, but I've tried to describe the key ideas in my book.

If Only Systems Were Deterministic...

If only systems would do as they're told, developers would have their way. Many people I meet still believe that systems are deterministic. But this Newtonian dogma was shattered in the 20th century. The history of scientific thinking tells us: The world is non-deterministic, get over it.

Computer science does itself a disservice by ignoring the main lesson of 20th century science, namely that the world is non-deterministic in fundamental ways. There is not even a well-developed theory of bugs. The push-button, imperative, API remote control approaches we use to instigate action today do not bring certainty. They offer a comfortable industrialization of process, but ultimately, by trying to remote control, we merely throw stuff over the wall and hope for the best. The only way to approach system reliability is to embrace the notion of indeterminism once and for all. It is about best effort.

Some things can help us here, such as building systems that are weakly coupled. System dependencies lead to strong coupling. If one thing fails, the system immediately transmits the failure to the next component. A weakly coupled system is fault tolerant.

Artificial Criticality

We escape from criticality by diversifying systems through redundancy. We never control systems, we merely keep their forces in balance. The knife edge of if-else programming is the radiation or asbestos of the software world. We stuff the walls full of this potentially dangerous automated reasoning, believing that it is there to protect us, when in fact it exposes us to an instability by the myriad pinpricks of a jostling environment.

Trying to conclude true or false from a highly complex environment is found to be the main cause of software unreliability. The reasoning for this is explained in the book.

Our thinking is still incredibly primitive, if we are expecting to scale reliable systems. We have given little evidence that we've understood the key issues of system automation in IT. Manufacturing and electronics have come a lot further. It's not only about how resilient the pieces are, but also about how they are put together. In several of the examples I've shown, the presence of regular maintenance could have prevented the gradual failure of the system.

The great pioneer of material science J.E. Gordon wrote [2] that: "The history of attempts to prevent cracks from spreading or evade their consequences is almost the history of engineering."

In the Comet airline disaster of 1954, microscopic cracks precipitated an avalanche failure that was so fast nothing could have prevented it from happening. In physical terms we would say that the rate of reaction dominated any process capable of preventing it. When there is a mismatch of dynamical scales like this, maintaining equilibrium is not possible. You are balancing on a knife edge. Semantics of design always give way to the dynamics of underlying reality.

There are two "answers" to this kind of failure: avoid stress concentrations, bottlenecks, and other points of failure; and use materials that catch the stress automatically by design, like the storm drain, like embedded glass and carbon fibers and alloys that spread load by deforming plastically. In IT terms, you want load balancing and failover without failure as part of the design, not as a late fire-fight.

Uncertain Infrastructures

External Intervention vs Embedded Smart Infrastructure

Why do we continue to make remote control systems that make the worst use of both humans and machines? Because we believe it's the only way to do it. But take a look at this picture of responding to a crisis.

- You wait for a crisis.
- You bring in a manual response (too late).
- You scale up the human operation by bringing power tools.

Now think about how simple drainage prevents most flooding as an entirely automated embedded system. We are obsessed by the manual intervention. It is a sign of technological immaturity. It is even more apparent in the way we attempt to orchestrate systems, using simplistic flow-chart thinking as a model of a highly parallel and distributed environment.

There are three phases to the system lifecycle that we need to rein in. We think very differently about each.

- **Planning:** Here we tend to think in terms of broad block semantics (boxes with arrows between) or workflows.
- **Operations:** A highly dynamic and parallel phase, where overt flow thinking is a hindrance / bottleneck.
- **Evaluation:** Here we look for artificial and misleading hindsight narratives about successes and failures.

Rimsky Korsakov would have rolled his eyes at contemporary descriptions of orchestration. Orchestration of total systems lies in the planning of highly parallel operations. We might only remember a specific storyline in hindsight—perhaps a good or a bad experience. This is how we usually describe the complex system, but it is not a true representation of it.

We have the opportunity to make introspective systems that merge semantics and dynamics into a unified picture. That will only happen when we remove the artificial distinction between development, configuration, operation, and monitoring.

In Search of Certainty

What does it mean to be certain about something? How do we make a reliable infrastructure for society?

Absolute certainty and determinism are myths. We can only do our best. As small forces in an environment that permits us islands of temporary calm, we must try to understand the bigger picture. There are three main issues: scale, complexity, and lack of knowledge.

Twenty years after I began CFEngine and my own research into these matters, it seemed time to tell the story of the thinking that went into it. My own interest has meandered around many topics within the scope of IT operations, and I have tried to describe how these pieces fit together in the book, but the main core of it can be understood easily as a simple-minded quest of a physicist to understand a system.

What I hope is that my book starts a discussion that shows how to apply some of the traditions of science to a subject that has ridden mainly on the coat-tails of engineering. How do we make promises we can keep? By understanding the nature of certainty itself.

If we take certainty seriously, we need to think carefully about how software is designed. We can't just throw software logic over the wall for operations to catch. We need to build for intrinsic stability from the outset through true automation. And, even then, we'll need to perform continuous maintenance, just to be sure(ish).