ALVA L. COUCH

from tasks to assurances: redefining system administration

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

AN ALTERNATIVE CONCEPT OF THE JOB of the system administrator leverages existing techniques of systems engineering and provides a foundation for a more synergistic relationship between system administrators and users.

Historically, there has been much confusion about what a "system administrator" is and does. One great success of the past decade is that we managed to define system administration in terms of the tasks the typical system administrator performs. This definition includes a taxonomy of system administration tasks [1], as well as the *Job Descriptions for System Administrators* booklet [2]. So far, these have served well as a de facto definition of the profession of system administration. We have obtained much leverage from this definition, including that the profession of system administration is now included as an option on the 2010 census.

But defining the profession in terms of tasks has a dark side; it invites naive observers to assume—as they do for plumbers and electricians—that our tasks define our profession [3]. In fact, our actual deliverables are much more abstract, including availability, integrity, and security. These are elements of the social contract between system administrators and users. I propose that this social contract, and not the tasks, is the real definition of the profession. What we are is not "what we do" but, rather, "what assurances we provide." Tasks support assurances, but are *not* the essence of the profession.

This is probably obvious to the average system administrator, but not at all obvious to management, who still on average consider system administration to be a task-based profession. We are to some extent "victims of our own success" in defining the profession via tasks. While tasks are easy to understand, social contracts are more abstract. How can we even write down the contract? Is the social contract defined in that ethereal thing called "policy," or something else? In the following, we explore some approaches to documenting the oft invisible and implicit social contract that is—already—a central component of the profession of system administration.

This article arose from teaching requirements analysis to aspiring software engineers last fall. The key principle of requirements analysis is to separate "requirements" from "design," in the sense that what a system should do ("requirements") is separate from how that is accomplished ("design"). *Separating requirements from design has many positive effects, including allowing the designer the freedom to address requirements in creative ways.* I asked myself, "Can these principles be applied to system administration to obtain similar benefits?" I realized that the prevalent definition of the profession in terms of tasks is actually "design," and that we seldom write down requirements in any other form. This article is the result of that train of thought.

This article might be loosely considered the third in a series. In the first article [4], I described the semantic wall between "high-level" and "low-level" specifications of system configuration and concluded that a new way of thinking is necessary to utilize "high-level" specifications. In the second article [5], I challenged the popular definition of system administration as managing system configuration, and redefined the profession as "closing open worlds," i.e., creating zones of predictable system behavior in an otherwise unpredictable world. In this article I take the next step, considering *which* worlds to close. This step comes with its own quandaries: the user wants the administrator to close "every" world, and boundaries must be drawn between what is "supported" and what is not. The decision as to which worlds to close is a social contract between system administrator and user.

From Tasks to Assurances

My first step is to drastically redefine the profession in a subtle but profound way. System administrators do not "perform tasks" or "apply expertise" but, rather, "provide assurances." An assurance is a clear statement of intent to address some user need. The set of assurances that a system administrator provides are part of the *social contract* between administrator and users. In very much the same way, while a plumber or electrician needs the prerequisites of being able to plumb or wire your house, in actuality these professionals honor a social contract that includes requirements for quality and reliability of the work they perform.

Converting between the old task-based definition and the new contract-based definition can be tricky. Sometimes the conversion between tasks and assurances is easy to make. For example, a system administrator is often seen as "managing printing" (a task), while the real job is "assuring that printing works" (a contractual obligation). Sometimes an assurance is based upon ability to perform a hopefully very infrequent task: for example, "recover from disasters" (a task) becomes "assure data integrity" (a contractual obligation). Some simply stated assurances are very difficult to map to tasks, e.g., "provide high-availability file service" requires mastery of many interrelated tasks.

THE SOCIAL CONTRACT

The social contract between administrator and user includes many facets. The most obvious of these are "ethics" and "privacy" assurances, which are now increasingly defined in writing as part of the job. But, at a deeper level, the social contract includes the assurances that the system administrator group makes about system behavior, as well as the priority of each assurance. A high-priority assurance will be addressed before a lower-priority assurance: if both the file server and a user application die at the same time, for example, the file server obviously takes priority.

PRIORITIES

Priorities are almost never documented in practice, and I think they should be! At one time or another, every system administrator gets into the situation of having to assure too much and has to make difficult decisions. What if a crucial program drops out of the user environment at the same time that the file server becomes unreliable? Luckily, a good manager is often there to defend decisions, but the priorities of assurances are often known far in advance. Writing them down changes the job from a "management decision" into "working from a pattern."

FLEXIBILITY

As in systems engineering, the main reason for describing assurances rather than tasks ("requirements" rather than "design") is to give the system administrator flexibility in providing those assurances. Assuring data integrity is a rather complex obligation, involving techniques for backup, recovery, and data security. The beauty of documenting assurances is that when some heretofore unknown technology comes along (e.g., using unused disk space as online backup), the assurance does not change, even though the tasks that provide that assurance might change drastically.

REUSABILITY

Are assurances reusable? The good news is that the kinds of assurances a system administrator makes do not vary much from site to site, that is, the list of assurances is (somewhat) reusable and relatively high-level compared to the task-based description of the job. Some of the most basic are present everywhere, including assurances of ethical behavior and appropriate safeguards for personal privacy. The taxonomy of assurances is really quite simple compared to the taxonomy of tasks. The bad news is that the priorities of various assurances differ greatly based upon the site. For example, empowering the user to do self-directed work might be the highest-priority assurance at an academic site and the lowest-priority assurance at a bank.

ASSURANCES ARE NOT POLICY

One might think that the definition of the job of system administrator arises from that ethereal thing we call "policy." It does not. "Policy" describes what systems and users should do, not who assures them and what forms that assurance takes. Many assurances that a system administrator makes are an implicit part of policy; use of a service implies reliability of the service. The transformation that turns policy into the social contract comes from asking, "What assurances are required to implement policy?"

Assurances and Requirements

At the most basic level, assurances for system administration are a list of system behaviors that should form a set of reasonable expectations on the part of users. At a deeper level, assurances are driven by (and are a proper superset of) user requirements: the things that users need in order to get their work done. The skilled system administrator converts the list of user requirements into a set of assurances by adding the implicit assurances of security, integrity, stability, etc., just as an electrician does not ask a customer whether to make outside wiring waterproof! At the next level, requirements become a set of service level objectives (SLOs) or even service level agreements (SLAs) defining response-time assurances: if and when things go wrong, how long should it take to correct problems? For example, an expectation is that "printing should work" and an SLO for that is that "a malfunctioning printer should be repaired in one day or less."

System administration is a rather unusual profession in that the actual behavioral requirements often take second place to the techniques and practices by which behaviors are assured, and documentation of practices often serves as the sole documentation of requirements. One obvious reason for this is that documentation of practice is currently the *only* common language we have for describing behavior! It is easy in this situation to confuse that documentation with requirements and, when we do that, our practice becomes a parody of satisfying user needs rather than the real thing.

For example, consider the task of managing printing. The "tasks" include doing various things that ostensibly keep printing working, including managing the service, repairing printers, etc. Our documentation of managing printing includes details on how to accomplish these tasks. But these tasks by themselves cannot be converted easily into SLOs. The corresponding assurance, by contrast, is much simpler: "Everyone is able to print in a timely fashion." This is easily converted into an SLO.

We are very lucky that the task of describing behaviors and requirements has been studied in great detail by others. In systems engineering and software engineering practice, this practice is called "requirements analysis" [6]. A "requirement" is something that the managed system should do, some behavior it should exhibit. There are many ways to document requirements, and there are several established techniques for accurately teasing requirements from user desires. One way to describe requirements is through first documenting "use cases," from which we then extract and describe a "requirements model."

USE CASES

Our first step in establishing a language for describing behavior is the same as in software or systems engineering. "Use cases" describe what the user should be able to do: for example, "users should be able to send and receive electronic mail." Note that the use case does not specify how or why any behavior should be assured, and is thus much simpler and broader than a practice for assuring behavior.

Several issues arise immediately when we write down the use cases. First, use cases are not definitive; they describe some things that should be possible, but not absolutely everything. To assure the use cases, we are left to fill in the details of other things that should be possible. Use cases describe mission-critical behavioral objectives but not peripheral objectives that users might desire. For example, "checks should be printable" is included but "personal greeting cards should be printable" is not. Use cases often include SLOs for how quickly something *should* happen, which can even, in some cases, become SLAs on how quickly something *must* happen. There is a big difference, for example, between the use case statements "sales transactions *should* be posted within two seconds" and "sales transactions *must* be posted within two seconds." Finally, use cases should not describe in any way *how* objectives are to be assured.

REQUIREMENTS MODELING

The next step is to abstract the use cases into patterns and concise representations. In software engineering, this phase is called "requirements analysis." Requirements analysis involves determining the classes of users and services (from the use cases) and documenting the relationships between classes of users (e.g., assignment-of-privilege classes to user classes and documenting inheritance between kinds of user and privilege classes). This is commonly referred to as a "modeling step," and the process is called "requirements modeling."

One powerful tool in requirements modeling is to express capabilities in terms of similarities between user roles, using object-oriented modeling. For example, there might be two kinds of users, "doctors" and "nurses," with different privileges. It might be that "doctors" are allowed to do things "nurses" cannot, but "doctors" can do anything "nurses" can do. Regardless of the real-world relationships between doctors and nurses, the behavior of the system in response to their queries is a simple inheritance relationship between behavioral classes: "doctor" system behavior is a subclass of "nurse" system behavior.

HOMOGENEITY AND HETEROGENEITY

In system administration, the terms "homogeneity" and "heterogeneity" usually refer to variation in the operating systems or hardware deployed at a site; we say that a site with a mix of Windows and Linux is "heterogeneous," for example, while a Linux-only site is "homogeneous." In requirements analysis, however, it is the user classes and behaviors that are homogeneous or heterogeneous, and not the managed systems! Users are "heterogeneous" if there are many user classes with different privileges, and "homogeneous" if all users have more or less the same privilege. Behavior is "heterogeneous" if there are different behaviors for each user class and "homogeneous" if not. These are properties of the mission and structure of the organization and not of the hardware on everyone's desks.

Like operating system heterogeneity, requirements heterogeneity costs more to assure. Thus it is prudent to question whether heterogeneity that naturally arises in requirements is necessary. For example, suppose that there is a *requirement* that user George has access to software to which no one else has access. This is going to be expensive, and one should ensure that this is *really* a requirement before proceeding. I believe that in many cases, heterogeneity of requirements is no less expensive than if George had a different *operating system* on his desktop machine.

AMBIGUITY

Once you have written down the explicit requirements, a pattern will emerge that is not unique to system administration. What you do not write down is as important as what you do. In any high-level description of requirements there will be some necessary ambiguity. Whether the requirements are useful at all depends on how we handle this ambiguity.

Suppose, for example, that one requirement is that "George should be able to compile files with gcc." Alas, this just isn't enough to describe precisely what George should be able to do. It does not say which header files should be present, or whether the kernel sources should be present to make kernel headers available. There are a multitude of factors exterior to gcc that might affect whether George can compile *his* files with gcc. George's real requirements are thus ambiguous, based upon your description.

Drift

Ambiguity is not nearly as bad in itself as in its social consequences. Anyone who goes to the trouble of writing down requirements will quickly discover that users are doing things that are outside the requirements—and getting away with them. In a modern computing environment, there is a prevalent idea that anything you are allowed to do is "supported" (or "assured"). But things you just happen to be allowed to do that are not requirements may go away at any time, due to policy changes, side effects of other changes, or simple mistakes.

A few years ago, at the beginning of the anti-spam effort, Tufts' Department of Computer Science closed down all access to SMTP from outside Tufts except for a few designated servers. The result was an outcry from students who had been running their own SMTP servers inside our network. The affected students claimed that our actions were costing them money by prohibiting business communications to their computers. The students were given a polite choice between relocating their business computing outside the Tufts network, and facing disciplinary charges for operating private businesses inside the university network!

This is an extreme example of a more general phenomenon that makes it difficult to specify requirements. Requirements are not what one can manage to do but, rather, what one *should* be able to do. They are not about what users *want* but, instead, about what users *need* in order to accomplish useful work, which is their end of the social contract.

REFINEMENT

Users are fairly good at describing their functional requirements, but less able to voice their requirements for privacy, security, integrity, and availability. Thus, the system administrator must often augment the list of user needs with implicit needs that users usually cannot voice. In requirements analysis, we might call the derived requirements a "refinement" of the basic user requirements.

Refinement is a matter of listing the requirements that are obvious to system administrators but not to the user. A good refinement consists of new requirements that are "obvious once written down." If one is refining correctly, the user's response will be, "Of course I need that."

BASELINING

One useful technique for the system administrator is to define behavioral requirements in terms of a baseline set of behaviors. This is a set of behaviors everyone should have access to in order to get their work done. It is a handy way of distinguishing between what users *need* (baseline behaviors) and what users *want* (non-baseline behaviors).

For example, at Tufts we have placed a limit on what users can expect from the support organization by establishing a "baseline configuration" for a desktop computer. This configuration satisfies a set of requirements necessary for interoperating with Tufts network services. But it has another social function, which is to define and delimit the responsibility of the support organization. Systems that fail to function according to the baseline will be returned to a baseline state, but functions that users desire outside the baseline are not supported.

The distinction between baseline and non-baseline behaviors can lead to major cost savings. Some organizations have reported that deploying thin clients that support only baseline functions (and, for example, prohibit the installation of custom software) results in up to 50% savings in cost of operations. Allowing users to install seemingly innocent software (e.g., MP3 players) can lead to substantially increased support costs.

As another example, in some system administration circles the words "reasonable faculty member" are an oxymoron. My support staff and I have an unusual social contract. I need high volatility of software configuration, much higher than staff can provide. So the staff provides a baseline configuration that I do not touch. I install my own software on top of this baseline, being careful not to change anything in the baseline itself. If I need a baseline change, I ask *them* to make it so that it becomes persistent. In this way my systems are co-managed by myself and my staff in a nearly ideal way, with staff doing what they do best and me doing what I do best. Their side of the social contract is to provide reliability and recovery; my side is not to make their job difficult. They have recovered from complete system failure by building a new system to my baseline requirements, after which I made the few customizations I needed and everything came back up. Thus high synergy can be obtained from proper use of baselining as a basis for a two-way social contract.

REQUIREMENTS AND DESIGN

Another reason that we really need a requirements step in system administration is that specifying requirements clearly leads to better "designs." As in software and systems engineering, in the design step we decide how to configure systems to provide requirements. Design can best satisfy requirements when those requirements are minimally constrained. For example, specifying the hardware composition of a user's workstation in the requirements step is very limiting, especially if the specified machine proves incapable of functions the user requires. It is better to have the option of satisfying requirements by replacing a user's workstation with another physical machine.

Effective design does not just satisfy requirements but also minimizes cost of operations. For example, even if only George needs gcc, it might be easiest to install it everywhere. This is a design decision, while the needs are a requirements decision. This gives other users additional privileges "by design" and not "according to requirements," in order to reduce management cost rather than to satisfy needs. There has been some controversy—especially in defense circles—about providing *any* capabilities to users that they do not need, but in the modern Linux environment, the homogeneity of a common core of software is more or less assumed.

Rethinking the Profession

In summary, I have redefined the system administration process as providing a set of assurances, derived from a refined set of user requirements, augmented with the requirements of our profession, and implemented via baselining and proactive tracking of ambiguities and drift in requirements and assurances. Why go to such trouble?

There are many reasons for documenting requirements. They clarify the actual job of system administrator. They leave one free to assure users of their requirements in the best possible ways. They protect the system administrator from outrageous demands. They appropriately focus discussion upon the mission of the enterprise. Using techniques from systems engineering, they can be used to predict cost of management and suggest an appropriate number of system administrators to hire.

One obvious reason that clear requirements are beneficial is that one can measure objectively whether requirements are met. It has often been said that the better a system administrator is doing, the less people know his or her name. By defining tangible and realistic requirements, rather than broad and sweeping impossibilities, we provide something that can be measured and offer a fairer estimate of system administrator performance than the alternative of remaining anonymous!

In a deep sense, our profession is about "closing open worlds," i.e., creating islands of predictability in which useful work can be accomplished, in an otherwise unpredictable universe. Some islands that we create are due to requirements; others are due to design considerations. Some islands of predictability rise up out of no clear intention on anyone's part! Understanding the landscape of predictability is the real job of the system administrator.

Caveat: That understanding does *not* solve the ongoing and significant problems system administrators have with public relations. A local discount store I frequent has a large sign on the door: "Confusion is our most important product." At present, many users think that this sign describes their system administrators! We need to get to the point where users understand instead that "Peace of mind is our most important product" and that the assurances we make are far more important and crucial than the services we provide.

This article is only a beginning at straightening out some long-term confusion about the profession. We started by defining a taxonomy of tasks, We now must face the harder problem of defining and managing a taxonomy of assurances and expectations. Most important, we have to see "managing systems" for what it is: beating a dead horse. When we can instead "manage assurances," the profession will truly be "at the next level."

REFERENCES

[1] Rob Kolstad et al., "The System Administration Book of Knowledge": http://ace.delos.com/taxongate.

[2] Tina Darmohray, ed., *Job Descriptions for System Administrators*, Short Topics in System Administration 8, USENIX Association, 2001.

[3] Alva Couch, "Should the Root Prompt Require a Road Test?" *;login:*, vol. 32, no. 4, August 2007.

[4] Alva Couch, "From x=1 to (setf x 1): What Does Configuration Management Mean?" *;login:*, vol. 33, no. 1, February 2008.

[5] Alva Couch, "Configuration Management Phenomenology," *;login:*, vol. 35, no. 1, February 2010.

[6] See, e.g., Roger Pressman, *Software Engineering: A Practitioner's Approach*, 7th ed. (McGraw-Hill, 2010), chapters 5–7.