Constraints and Controls

The Sociotechnical Model of Site Reliability Engineering

LAURA NOLAN

Laura Nolan's background is in site reliability engineering, software engineering, distributed systems, and computer science. She wrote the "Managing Critical State" chapter in the O'Reilly Site Reliability Engineering book and was co-chair of SREcon18 Europe/Middle East/Africa. Laura Nolan is a production engineer at Slack.
laura.nolan@gmail.com

MIT's Professor Nancy Leveson gave a talk at SREcon19 EMEA about her research on safety engineering and accident analysis [1]. Leveson's work draws on case studies from military air-traffic control in Iraq, contamination of water supplies, failure to launch a satellite [2], as well as the accidents that resulted from the Therac-25 software-controlled radiation therapy device [3].

One of the examples described in that SREcon talk was a training exercise undertaken by a pair of fighter pilots. The plan was for a pilot to fire a dummy missile at another aircraft. One of the plane's missile tubes was loaded with a dummy, while other tubes contained live missiles. The pilot targeted the other aircraft, selected the tube with the dummy, and fired—a live missile. This wasn't pilot error: it was a systems accident. The plane had a smart missile selection system that would substitute another missile if the tube the pilot selected was blocked, and in this case an antenna was in front of the tube with the dummy.

The thesis of Leveson's talk is that traditional methods of managing risk in systems, such as fault tree analysis and analytic decomposition, do not work in the context of complex systems. These established techniques involve breaking larger systems down into smaller subsystems, reasoning about the likelihood of failure of these components, and calculating overall reliability of the system from there. Unfortunately, this isn't effective: many systems accidents happen because of unanticipated interactions between parts of the system that were working as intended.

We see these kinds of interactions in computer systems all the time. Reddit's outage on August 11, 2016 [4], is a great example: they were performing maintenance on their Zookeeper cluster. Reddit's autoscaler system relies on Zookeeper for input data, so in order to prevent the autoscaler from doing the wrong thing while Zookeeper was under maintenance, they turned it off. Unfortunately, their configuration management system turned the autoscaler back on, and it took their site down. That, of course, isn't as bad as shooting down a friendly aircraft, but the incidents do have elements in common.

In both those examples, no part of the system was broken, but the system overall didn't work as expected. The failure of analytic decomposition is especially acute for systems involving software, because so many software problems arise from unexpected interactions between parts of our systems, not simple component failure. Safety (or reliability, from our perspective) is a property of the entire system, not of the components of the system.

Leveson's approach, STAMP (Systems-Theoretic Accident Model and Processes), has three parts:

- Constraints, or conditions needed for the system to operate safely
- Hierarchical safety control structures, which work to enforce the constraints
- Process models describing the state of a system and how it moves from one state to another

According to STAMP, designing for reliability starts with figuring out what the key system constraints are, then analyzing how candidate designs can be controlled in such a way to be kept within those constraints. One article doesn't afford nearly enough space to do justice to the intricacies of STAMP, so this column will be focused on Leveson's concepts of system constraints and control structures and how they relate to site reliability engineering (SRE).

Hazards, Constraints, and Controls

For Leveson, safety is all about maintaining control of the system. Start by figuring out the hazards around your system. For a public water supply the hazard might be "avoid exposing the public to contaminated water." In a production software environment, the hazards are likely to be things like "keep the error rate under 0.1%," "don't expose web servers directly to the Internet," or "don't lose user data."

From the hazards, you derive a set of constraints. For the water supply system, those might be "water quality must meet standards," and "if water quality falls below standards, steps must be taken to reduce risk of exposure (e.g., boil-water advisories)."

For your production software system, constraints could be things such as "new releases must be canaried to ensure the error rate doesn't increase," "firewall rules must be in place to prevent access to the web servers," or "maintain at least three replicas of critical data," as well as "the system must have enough compute, storage, and bandwidth available to it in datacenters foo and bar," or "service foobaz, on which we depend, must be operating with a 95th percentile latency under 100 milleseconds."

This should look pretty familiar so far: these are more-or-less service level objectives (SLOs) that our system is expected to fulfill and SLOs that our system needs from other systems or infrastructure.

According to Leveson, hazards and constraints are a critically important part of system design, and deriving them needs deep domain expertise. Once you've defined your constraints, you have to figure out how to monitor them and keep your system within them. This means designing the controls that enforce the constraints. If an incident does happen, accident analysis should be focused on finding the failures or gaps in the system controls that allowed the incident to take place.

Controls are not only technical, however; the entire system of humans involved in the development, operation, and oversight of a system are also part of the control system. For some safety-critical systems, like nuclear reactors or food safety, this goes as far as including the government and courts as part of the control system. For SRE, this usually means the team responsible for a given service and the management and leadership structure to which SRE teams report.

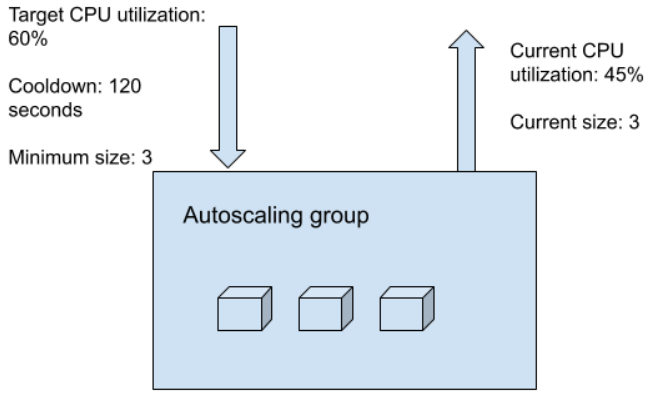

Figure 1: An autoscaling group

Reference and Measuring Channels as SLOs and SLIs

To control a system you need two things: a way to specify the constraints on the system and feedback. Take a simple technical example: an autoscaling group (as provided by the major cloud platforms).

Figure 1 is a model of an autoscaling group from the perspective of the user—an implementor would have a more detailed view of the system internals. The autoscaling group currently contains three instances. It is configured to keep a minimum of three instances running. It'll increase the number of instances if the CPU utilization exceeds 60%. There's a cool-down period of 120 seconds, so the autoscaler won't increase or decrease the number of instances until two minutes have passed since the last scaling action.

In STAMP terminology, the control information is the reference channel (the inward arrow in Fig. 1): this is the information needed to do the job of imposing constraints on the system. The outward arrow, the system metrics, is the measuring channel, which gives information about how the system is behaving—is it within its constraints or not?

The concepts of reference channels and measuring channels map very closely to SLOs and SLIs (service level indicators), respectively. An SLO, or reference channel, is a specification of how you want your system to behave, and an SLI, or measuring channel, shows whether or not your system is achieving its SLO. Control doesn't work without feedback. This, perhaps, is the reason that SLOs and SLIs are so often seen as the essential first step to adopting SRE practices—but they are definitely not the only form of reference and measuring channels needed.

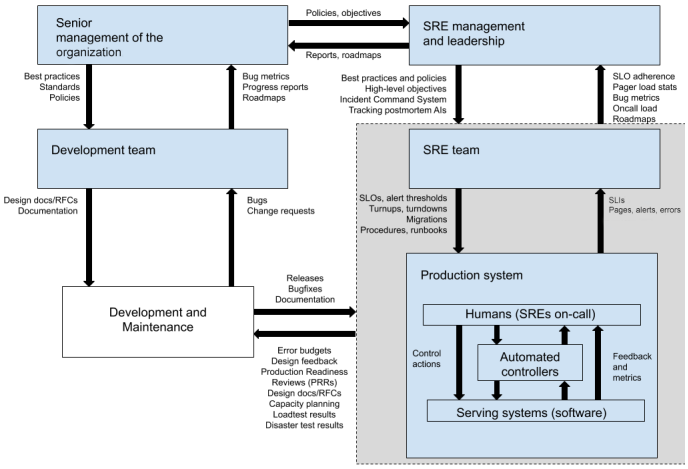

Figure 2: SRE model of sociotechnical control

The SRE Sociotechnical Model

Leveson's SREcon talk really resonated with a lot of people at the conference. The problems of complexity arising from component interactions are our everyday experience, even if our context is with RPCs or data pipelines rather than ballistic missiles or satellite launches. We're very familiar with the need for dynamically controlling the systems that we run and the difficulties that arise from that (we saw some examples in the last instalment of this column when we looked at dynamic control systems and public cloud outages [5]).

The aspect of STAMP that is most relevant to SRE, however, is that it treats the organizational side of system reliability as a first-class citizen. What SREs do at a purely technical level doesn't look much different to software engineering or system administration: we do debugging and performance analysis, and we write C++ or Java or Go or bash scripts or Terraform configs or Prometheus rules, like anyone else in software. The organizational practices aimed at managing and controlling technical complexity, however, make SRE different—and it turns out that many of these practices have close analogues in STAMP.

According to Leveson, safety control structures are hierarchical. Constraints are created at a higher level to control processes at the lower levels of the hierarchy, until you eventually arrive at the operating process itself and its direct control mechanisms.

There are different ways that SRE engagements can be structured organizationally [6], but the classic setup at Google, where SRE originated, is for an SRE team to report to an SRE management function and to collaborate with one or more development teams. The SRE team manages the system in production and uses the experience gained from that to inform its engineering work, which is focused on reliability, scalability, and performance. The development team works on features and collaborates with the SRE team on changes needed to keep the system stable and within its service level objectives.

The SRE organizational model includes a host of different controls and forms of feedback, from error budgets and direct interaction with the production system itself to forms of control generally performed by management, such as setting organization-wide policies and objectives and measurements, like pager load over time. The diagram above is a SRE-specific version of Leveson's general model of sociotechnical control [2].

Constraints, Controls, and the SRE Team

At the SRE team level, the focus is on the technical systems. SRE teams are normally deeply involved in defining SLOs for their systems. Much of our technical work directly involves ensuring the system is kept within SLOs—from design work to monitoring and automation to control the system.

Healthy SRE teams also self-monitor, working at one level of abstraction above the system itself. They're looking at trends in SLIs over longer periods of time, for patterns of incidents, for upcoming problems like hitting scalability limits, for upgrades or migrations that need to be performed, for new kinds of repetitive manual work that may need to be automated.

Teams need control structures to make sure these self-monitoring activities happen regularly. Most SRE teams use a weekly production meeting [7] to review the state of their production systems, and this meeting is the natural site for much of the self-monitoring that teams do. Teams will review service metrics, outages, paging events, and other interrupts such as tickets: all of these are measuring channel activities. As a result of this, teams will make decisions that affect their reference channels: updating runbooks, tweaking alerts. They'll also surface issues that require engineering work, which might be done within the SRE team or become requests to the partner development team, which usually has some representatives in attendance at the production meeting.

SRE and Development Team Collaboration

As well as attending the weekly SRE-run production meeting, SRE teams have several other reference and measurement channels with developer teams. Development and maintenance of systems is a joint activity shared by developers and SREs. Both SREs and developers write design documents (also known as RFCs, or requests for comment) and provide feedback on the other team's designs; this is a very important pair of reference and measurement channels, as each side has its own set of system knowledge and perspectives.

Production readiness reviews (PRRs) [8] are another important channel between developers and SREs. PRRs are generally used when a new service is being onboarded by an SRE team. SRE teams normally evolve a fairly comprehensive team-specific checklist for new services that covers items such as:

- Review of system architecture and dependencies
- Review of the system against the team and the organization's standards
- Review and development of SLOs
- Review and development of monitoring and alerting
- Review of change management practices (such as canarying)
- Developing training that can be delivered to the SRE team

During the PRR process, the SRE team will work through this checklist with the developer team. The PRR process is a reference channel; the SRE team imposes constraints on the standards of the systems they are willing to support.

Error budgets are another well-known reference channel that developer teams and SREs share. Error budgets are defined based on SLOs: how much unavailability can a service have during a given quarter and still be within its SLO? The SRE team monitors a service's SLO and error budget. If the error budget for the quarter has been exhausted, then an SRE team should push back against risky launches and normally will negotiate with the developer team to prioritize reliability-related work.

Monitoring SRE Teams

Leveson says control is hierarchical. We've already seen how SRE teams control and monitor their services. In large organizations, SRE management and leadership should also have a role to play in monitoring the health and efficiency of SRE teams:

- Are their services generally meeting their SLOs?
- Are they getting paged too often?
- Do teams have sufficient staffing to do substantial engineering work as well as operational work?
- Are high priority postmortem action items being done?

This doesn't mean that leadership should micromanage. The feedback loops provided by measurement channels get longer the further up any hierarchy you go, and so control becomes less effective. Management should be concerned with longer-term patterns over multiple quarters.

This should not be a coercive approach, focused on demanding that teams hit their metrics by working unsustainable hours or at the cost of doing the right thing for their service—for instance, teams should be able to prioritize fixing a newly found major risk to their service's stability over low and medium-priority postmortem action items, even if it means that those open postmortem action items will be visible to management in the form of metrics. The approach should be about making sure that teams have resources and organizational support to get their job done effectively and to prioritize the highest impact work. Done right, this should not be a box ticking exercise.

SRE management is also in a great position to increase the effectiveness of the entire SRE organization by spotting places where standard tools and processes can help—these are, of course, reference channels. Examples of this could be introducing a standard process for managing incidents, or kicking off a project to build a production-grade tool for doing deployments or chaos engineering.

Conclusion

This article has just scratched the surface of Leveson's work. Nevertheless the STAMP concepts of reference and measurement channels and hierarchical control systems very closely describe what it is that SREs do. Learning about STAMP gave me a clearer insight into the organizational side of SRE.

The "what is the difference between SRE versus DevOps" debate has been well played out by now, but I'll add my contribution nonetheless: SRE is about the humans that design and control the systems as much as it is about technical considerations.

SREs are in the business of defining objectively which system states are acceptable and which are not. Our job is implementing controls, both technical and organizational, to keep our systems healthy. Our teams are part of those systems too, and also need to be healthy to be effective. Pain is unpleasant, but it is an essential form of feedback—it tells us to stop doing the thing that hurts in order to stay healthy. Far too many teams in operations are in pain, quarter to quarter, year to year. Does your organizational model notice?

References

[1] N. G. Leveson, "A Systems Approach to Safety and Cybersecurity," SREcon19 EMEA: https://www.usenix.org/conference/srecon19emea/presentation/leveson.

[2] N. G. Leveson, *Engineering a Safer World: Systems Thinking Applied to Safety* (MIT Press, 2012).

[3] N. G. Leveson, "Medical Devices: The Therac-25": http://sunnyday.mit.edu/papers/therac.pdf.

[4] "Why Reddit Was Down on Aug 11": https://www.reddit.com/r/announcements/comments/4y0m56/why_reddit_was_down_on_aug_11/.

[5] L. Nolan, "Managing Systems in an Age of Dynamic Complexity Or: Why Does My Single 2U Server Have Better Uptime than GCP?" *;login:*, vol. 44, no. 4 (Winter 2019): https://www.usenix.org/publications/login/winter2019/nolan.

[6] D. Ferguson and P. Labhane, "SRE Team Lifecycles," in B. Beyer, N. R. Murphy, D. K. Rensin, K. Kawahara, S. Thorne, eds., *The Site Reliability Workbook: Practical Ways to Implement SRE* (O'Reilly, 2018).

[7] N. Murphy et al., "Communication and Collaboration in SRE," in B. Beyer, N. R. Murphy, D. K. Rensin, K. Kawahara, S. Thorne, eds., *Site Reliability Engineering: How Google Runs Production Systems* (O'Reilly, 2016).

[8] A. Cruz and A. Bambhani, "The Evolving SRE Engagement Model," in B. Beyer, N. R. Murphy, D. K. Rensin, K. Kawahara, S. Thorne, eds., *Site Reliability Engineering: How Google Runs Production Systems* (O'Reilly, 2016).