Green team paper:
Falling off the cliff: when systems go nonlinear

Yvonne Coady, Russ Cox, John DeTreville, Peter Druschel, Joseph Hellerstein, Andrew Hume, Kimberly Keeton, Thu Nguyen, Christopher Small, Lex Stein, Andrew Warfield[1]

Abstract

As the systems we build become more complex, understanding and managing their behavior becomes more challenging. If the system's inputs are within an acceptable range, it will behave predictably. However, the system may "fall off the cliff" if input values are outside this range. This nonlinear behavior is undesirable, because the system no longer behaves predictably: it may not be possible to use, control or even recover the system. In this paper, we describe what it means for a system to fall off the cliff. We outline methods for detecting and predicting these modes of nonlinear behavior, and propose several approaches for designing systems to cope with these instabilities, or to avoid them altogether. We conclude by outlining open research questions for investigation by the systems community.

1. Introduction

A system behaves *nonlinearly* when the same input or environmental change produces different system behavior at light loads and at heavy loads. For instance, an increased demand at light loads might produce *linear* behavior, where the amount of work the system performs is proportional to the load, and the system quickly recovers from perturbations. Systems may operate nonlinearly under heavier loads, producing poor or unpredictable performance and perhaps even resulting in a partial or complete system collapse.

Systems respond to input changes like increased load in different ways. Some employ various techniques to reduce load and bring the system back to a "safe" mode of operation. Alternately, they may gracefully degrade performance or otherwise reduce the quality of service provided for each request. In the absence of such coping mechanisms, systems may degrade gracelessly, resulting in a loss of predictability, recoverability, or controllability.

The real world is full of systems that exhibit each of these strategies. One example is the US telephone network. To reduce capital costs, local telephone switches are configured to handle some limited number of concurrent calls that is far below the theoretical limit. As a result, an emergency that causes a sudden spike in call attempts can swamp local systems. Customers who do not receive service promptly often hang up and retry repeatedly, nonlinearly increasing the system load and lengthening the period during which they do not receive service.

The telephone network incorporates a number of effective techniques to handle overload more gracefully. For example, although call attempts are normally handled in FIFO order, during overload they are handled in LIFO order, increasing the fraction of customers who receive prompt service and thereby reducing retries. These techniques have evolved over a period of decades, producing a stable telephone system, but this lengthy evolutionary path is not available to more modern systems.

Another example of graceful degradation is the reaction of the CNN.com web service to increased demand after the terrorist attacks on September 11, 2001 [9]. In the span of fifteen minutes, page request demand increased by an order of magnitude: peak demand rose to 1.8 million hits per minute, which is 20X of normal demand. The organization employed several techniques to handle the load spikes. First, they dynamically reprovisioned servers from other web services (e.g., cartoons or entertainment) to reinforce their news service. They also added additional server capacity to the system. Most interestingly, they chose to reduce the com-

[1] Author affiliations and email addresses: University of Victoria, *ycoady@cs.uvic.ca*; MIT, *rsc@mit.edu*; Microsoft Research, *johndetr@microsoft.com*; Rice University, *druschel@cs.rice.edu*; IBM Research, *hellers@us.ibm.com*; AT&T Research, *andrew@research.att.com*; HP Labs, *kimberly.keeton@hp.com*; Rutgers University, *tdnguyen@cs.rutgers.edu*; Vanu, Inc., *chris@vanu.com*; Harvard University, *stein@eecs.harvard.edu*; University of Cambridge, *andrew.warfield@cl.cam.ac.uk*

plexity of the pages they serviced, by removing advertisements and pictures, and focusing on the text.

The real world also provides examples of graceless degradation. On August 14, 2003, a series of operator errors caused an electrical power grid in northern Ohio to stop tracking failures caused by sagging wires on a hot day [2]. Power lines sag as they carry power, and sometimes fail after contacting trees, forcing reallocation of power through other wires, and making them fail, too. Later, as the outage spread, overload sensors in other areas produced false positives and shut down more of the system, eventually affecting 50 million people in the United States and Canada. Power was not fully restored in some areas for over a week.

We would like our systems to behave more like the telephone network or CNN.com, rather than the power grid on that hot August day. Can we design and build systems whose behavior we can understand and that do not behave unexpectedly under unusual circumstances? Can we build systems that work all of the time, not just much of the time? Or are complex systems inherently unpredictable under unusual loads?

2. Defining graceless degradation

Graceless degradation describes the situation in which a small change in a system's input or environment causes a large degradation in system behavior. We take a broad view of change, including an increase in load, a modification of configuration, or the installation or upgrade of a system component. Similarly, we define degradation broadly to include a loss of *predictability, recoverability,* and *controllability*. This section characterizes these types of degradation and discusses why they occur.

Types of graceless degradation

Probably the most familiar form of degradation is the loss of predictable performance for a service. For instance, in the management of virtual memory, a small increase in the multiprogramming level can result in highly variable response times if not all working sets can fit into memory. Alternately, in the configuration of database management systems, a small increase in buffer pool size can rapidly degrade throughput if it results in a change in query plans. In these examples, a small change in concurrency level, load, or data volume results in a tremendous change in system performance.

A common cause for a loss of predictable service under load is the use of load adaptation mechanisms that introduce feedback loops into the system. An example of such feedback is *thrashing* – the situation in which a virtual memory system is so constrained in satisfying the physical memory requirements of a set of concurrent applications that it spends the majority of its time moving pages of memory to and from disk rather than making forward progress. Still another example is TCP's congestion control mechanism. TCP interprets packet loss as an indication of congestion and halves a connection's transmission rate in response; this behavior results in poor performance on even moderately lossy links [4]. This congestive response has been shown to be vulnerable to attacks on TCP flows, especially where TCP implementations use constant retry timeouts; well-timed bursts of data have been shown to render the TCP connections on a link useless [7, 8].

A second type of graceless degradation is the loss of recoverability of the critical resources being used or provided by a system. In a storage system, the data being stored is a critical resource. As the underlying storage system degrades (e.g., as physical disks fail), this data is itself in a degraded state: it is less capable of surviving further failures, and may not be provided at as high a throughput as in the fully-functional system. After sufficient degradation, the resources are irrecoverably lost. This form of degradation results in a compromise of the system's capacity to provide its service.

A final type of graceless degradation is a loss of controllability. Often, as a system degrades, so does the ability to intervene to prevent further decline. A naïve example is that of a UNIX system experiencing a *fork-bomb*, in which a malicious process alternates between consuming system resources and forking copies of itself. An administrator wanting to recover the system needs to kill the forking processes, but as time progresses must kill more and more processes, with less and less resources available to recover.

Why does graceless degradation happen?

As system designers, we hope to build systems that are stable and predictable under all possible (or at least all specified) operating conditions. We now consider a set of specific characteristics of existing systems that lead to graceless degradation. This list is hardly comprehensive, but rather an attempt to identify some key factors of concern.

Renewable resource exhaustion: systems that allow over-subscription of renewable resources (CPU, memory, and network connections) are susceptible to over-

load. While overloaded, the system will have to provide some means of dividing the limited resources across consumers until the load returns to an acceptable state. Although overloading of renewable resources is not necessarily a cause of graceless degradation, the mechanisms for dealing with it frequently are.

Persistent resource exhaustion: As discussed with respect to the loss of recoverability above, the persistent resources of a system may themselves be compromised. This situation may occur due to failure of the underlying devices, system or application software, operator mistakes, or malicious attacks. Systems may be designed to be resistant against this form of degradation by employing various forms of redundancy.

Feedback-induced degeneration: Adaptation mechanisms within a system that feedback into the system's operational behavior may enter states of oscillation or related instability, and thus prevent the system from getting useful work done.

Removal from expected operating regime: A superset of the previous example, a system may be forced into an unexpected mode of operation. Such a phase change may result in the execution of poorly-tested code paths and compromise the stability of the system as a whole.

Degradation of operating state over time: Small, non-performance-critical problems may accumulate over the course of a system's lifetime. These state permutations may result in difficulties much later. Consider the management of applications on modern operating systems, in which *OS rot* may eventually result in the inability to update a system, requiring that the OS be re-installed from scratch.

Error conditions or exception logic: Exception logic is rarely invoked and often poorly tested. Thus, when it is invoked, it is common for degradation or even failures to result. Often exception logic is invoked as a result of a small increment in load that causes a buffer pool to overflow or too many file handles to be acquired. Thus, while the change in workload appears to be small, the change in the execution path is substantial.

Unintended software reuse: Modular software design encourages the reuse of components, as well as the construction of hierarchical systems from existing components. Although this approach can reduce costs, it can also lead to successful systems being used in ways and environments never imagined. Components can behave predictably in their intended environment, but unpredictably in others.

Unclear usage semantics: The premise of this section is that a small change results in a large degradation in performance. But sometimes it is difficult to quantify "small." For example, is it a small change to increase a buffer pool size by 1KB in a database management system with 1GB of memory? While this is small in terms of the fraction of memory affected, it may be huge in terms of the impact on query plans. To address this case and the previous one, it may be valuable to specify constraints on how software components can be safely used, and to verify the satisfaction of these constraints before deployment or during execution.

3. Detecting graceless degradation

Computer systems susceptible to graceless degradation should be built to detect such graceless degradation and substitute a more graceful alternative. The first step in such an approach is monitoring the system to detect when graceless degradation occurs or is about to occur.

If we view the system as a black box with inputs (e.g., request load, hardware failures) and outputs (e.g., throughput, latency, correctness, and other application-specific performance measurements), then a basic detection strategy is to characterize the safe operating ranges for the inputs or the outputs (or both) and detect when the system has moved outside the safe operating range.

Some operating constraints can be derived from the design of the system. For instance, a decision to use erasure codes places a hard limit on the number of fragments that must be available. Other constraints might be derived from more general requirements: a web server should respond to a request within a minute or else the response is likely to be ignored by the web browser – either the program or the human, both of which are likely to have timed out. The most precise constraints can only be derived from testing the system and determining what operating conditions keep it performing as desired.

Testing a large computer system may be non-trivial: the CNN.com web servers reached over one million page views per minute following both the 2000 U.S. elections and the 2001 terrorist attacks. Generating such conditions during testing requires a significant test framework. Computer systems designers might take solace in the fact that, unlike physical structures such as bridges, computer systems are usually not destroyed by being tested beyond their limits.

System load is not the only interesting parameter during testing. For example, testing a web server farm might also mean checking how many server failures the farm can tolerate simultaneously without entering a cascading failure scenario. Parameters are often interrelated: in the previous example, offered load certainly affects the number of server failures that can be tolerated.

Black-box testing may be insufficient for applications that are expected to run for long periods of time, as it is difficult to identify inputs and environmental conditions that can drive an application into unsafe regimes. Currently, some researchers are exploring the alternative *white-box* testing approach. For example, if source code is available (or byte code for Java applications), the compiler may be able to help. Compiler analyses can aid in the coverage testing of *uncommon code paths* such as recovery code [6]. Compiler analyses and instrumentations may also help applications and the runtime system/OS to track resource usage and detect when an application may be approaching the cliff.

Testing should focus on determining safe output parameters, as well. For example, if a web server can respond to all requests within five seconds under expected conditions, then significantly longer response times in a real deployment are indicative of unexpected behavior, possibly graceless degradation. Safe operating ranges could also be defined in terms of more cumulative statistics. For example, high variance in one-minute average throughput during high load might indicate that the servers are experiencing performance problems.

Once the expected safe operating parameters have been determined, the system must be able to continuously measure and check these parameters, ready to change behavior if graceless degradation is detected or anticipated. Statistical learning techniques may provide a means for understanding the observed data [5].

4. Coping with graceless degradation

There are several ways to cope with and even avoid graceless degradation. Admission control limits the amount of load that can enter a system. Overprovisioning builds a buffer of extra resources into a system. Reprovisioning dynamically adds resources as needed. Load shedding drops or scales back processing when resource over-commitment is detected.

Admission control conditions system load to try to avoid load spikes. Unlike physical systems, which often have implicit capacity-based admission controls, computer systems cannot depend on physical space or fixed environmental conditions to impose limits. As a result, computer systems must explicitly control admission. Examples of admission control in computer systems include circuit signaling in computer networks or user login. However, many computer systems (e.g., IP networks or web servers) use very little admission control. Admission control and overprovisioning are duals. An ideal admission control scheme conditions load so that it can never take a system out of its safe region. Overprovisioning makes the safe region so vast that the cliff is over the horizon.

Computer systems tend to underprovision for efficiency, rather than overprovision for safety. When compared with bridges, buildings, or other physical systems, most computer systems are designed with few excess resources. Overprovisioning presents two challenges. First, it is expensive. Second, it is difficult to know how much to overprovision each resource. Overprovisioning to handle load spikes smoothly means that most resources will be idle most of the time. Statistical multiplexing increases resource utilization by gambling on uncorrelated load. When the requests become correlated, the system receives a burst, and the gamble has been lost. At this point, the system is approaching the cliff and has to choose a strategy for coping. It can try to reprovision resources to move the cliff farther away, or it can use short-term approaches, such as load shedding, to back away from the cliff.

While software complexity may make it difficult to define a safe region *a priori*, the flexibility of software control provides a means to rescue systems that are leaving their safe region and heading for the cliff. Software can reprovision and reorganize system resources in real-time. For example, many storage systems use a virtualization layer to make capacity addition and failure events transparent to applications. In contrast to overprovisioning, where resources are statically allocated to absorb peak load, reprovisioning either changes the mix of resources or includes resources from an external source. For example, resources could be incorporated from a pool shared across many systems. In this case, reprovisioning is an attempt to statistically multiplex overprovisioning across independent systems. Systems that share a resource pool should have uncorrelated needs for the pool to remain solvent.

Reprovisioning is particularly important for situations where falling off the cliff implies loss of recoverability. For example, consider data redundancy for availability and durability. If the data is replicated using erasure coding, when the number of fragments drops below a

critical threshold, then the data becomes unrecoverable. Consider an erasure code-based P2P storage system. If replicas are failing and some data are approaching their critical threshold, then it becomes necessary to reprovision storage nodes, even at the expense of handling incoming load. Incoming load could be throttled down through admission control or the load shedding approach discussed below. This example illustrates that coping mechanisms can be combined effectively. TotalRecall is an example of such a system; it automatically measures and estimates the availability of host components and calculates and enforces the appropriate redundancy mechanisms and repair policies [1].

A final approach for avoiding graceless degradation is load-shedding. The simplest approach to shedding load is to drop requests from the tail of a FIFO queue. Another approach is to prioritize and postpone work where possible. One example of this approach is soft updates, which stabilize file system performance under heavy load by tracking the dependencies between block I/Os to postpone disk updates until the system calms. In the extreme case of the load shedding approach, a system might choose to avoid a cliff by resetting its state and starting fresh through either a full or partial reboot [3].

Load shedding may provoke feedback from the higher-level systems that issued the dropped requests. If the feedback is poorly behaved, it threatens to further aggravate an already struggling system: consider the phone retry example from Section 1. Synchronization is a danger because it correlates load and neutralizes statistical multiplexing. Randomization can help avoid synchronization. Exponential backoff can also reduce feedback problems by progressively delaying consistently problematic retries [10]. Negative acknowledgments (nacks) avoid generating load on an overloaded system by using acks for success cases and not sending any reply for errors, letting the higher layer time out. These approaches have their limits, however: they assume that the upper layer is a trusted and logical system, which may not always be the case.

Many systems are designed under the assumption of particular environmental parameters. Using randomization is one way to immunize a system against fluctuations in these parameters. The system will not behave optimally under some conditions, but at least it will not perform terribly under others. Randomized file system layout has been shown to provide stable file system performance across storage system virtualization parameters [11]. For routing in a hypercube, sending first to a random neighbor has been shown to improve performance by balancing messages across queues [12].

5. Summary and open research questions

Catastrophic failures have forced us to consider what should be done to better understand and manage system software. Avoidance and detection strategies require that we not only clearly define where the cliffs are, but also identify trends that force systems towards them. Key future challenges thus revolve around identifying a meaningful set of system constraints to describe safe operating regions, effectively capturing information about the system's operational state, and responding to cliff-inducing conditions in a timely fashion.

System constraints must be holistic to be meaningful. Some set of local system constraints may be known *a priori*, while potentially global constraints must be dynamically derived from specifics of system configuration and execution environment. Open questions include how best to identify and represent constraints or safe modes of operation, how to expose the right parameters for local constraints and how to dynamically derive context-specific holistic constraints.

Testing the system may help to discover operating constraints. Required advancements in this area include trace collection of heavy load scenarios, workload generators to synthetically generate load or to replay collected traces, and development of large-scale simulation and/or emulation environments.

The process of capturing and mining system state introduces several challenges. Given the vast amount of shared system state and increasing variability of configuration options, research challenges include how to manage state collection carefully, how to selectively monitor state according to global/local information needs, and how to quantify critical tradeoffs in safety and performance.

Responding to potentially cliff-inducing conditions requires an appropriate coping strategy. Further research is required to define new approaches for enforcing safe modes of operation and for gracefully degrading system behavior, and to understand the conditions under which each strategy may be appropriate.

Given the nature of this problem and the dramatic increase in its importance, we as a research community must collectively commit to better understanding and managing the systems we build.

6. References

[1] R. Bhagwan, K. Tati, Y. Cheng, S. Savage, and G. M. Voelker. "TotalRecall: systems support for automated availability management," *Proc. of ACM/USENIX Symp. on Networked Systems Design and Implementation (NSDI)*, San Francisco, CA, March 2004.

[2] Canada-U.S. Power System Outage Task Force. *Final report on the August 14, 2003 blackout in the United States and Canada: causes and recommendations*. April 2004. Available from http://www.nrcan-rncan.gc.ca/media/docs/final/finalrep_e.htm.

[3] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox, "Microreboot - a technique for cheap recovery," *Proc. of 6th ACM/USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, San Francisco, CA, December 2004.

[4] R. Chakravorty, J. Cartwright, and I. Pratt. "Practical experience with TCP over GPRS," *Proc. of IEEE GLOBECOM*, Taipei, Taiwan, November 2002.

[5] I. Cohen, M. Goldszmidt, T. Kelly, J. Symons, and J. Chase. "Correlating instrumentation data to system states: a building block for automated diagnosis and control," *Proc. of 6th OSDI*, San Francisco, CA, December 2004.

[6] C. Fu, B. Ryder, A. Milanova, and D. Wonnacott. "Robustness testing of Java server applications," *IEEE Trans. on Software Engineering*, Vol. 31, No. 4, April 2005.

[7] M. Guirguis, A. Bestavros, and I. Matta. "Exploiting the transients of adaptation for RoQ attacks on Internet resources," *Proc. of Intl. Conf. on Network Protocols*, Berlin, Germany, October 2004.

[8] A. Kuzmanovic and E. Knightly. "Low-rate TCP-targeted denial of service attacks. (The shrew vs. the mice and elephants)," *Proc. of ACM SIGCOMM*, Karlsruhe, Germany, August 2003.

[9] W. Lefebvre. "CNN.com: facing a world crisis," Invited talk at *USENIX 15th Systems Administration Conference (LISA)*, San Diego, CA, December 2001. Summary available from http://www.usenix.org/publications/library/proceedings/lisa2001/lisa2001confrpts.pdf.

[10] R. Metcalfe and D. Boggs, "Ethernet: distributed packet switching for local computer networks", *Communications of the ACM*, Vol. 19, No. 5, July 1976, pp. 395 - 404.

[11] L. Stein. "Stupid file systems are better," *Proc. of ACM/USENIX 10th Workshop on Hot Topics in Operating Systems (HotOS X)*, Santa Fe, NM, June 2005.

[12] L. Valiant. "A scheme for fast parallel communication," *SIAM Journal on Computing*, Vol. 11, No. 2, 1982, pp. 350 - 361.