Robustness in Complex Systems

Steven D. Gribble
Department of Computer Science and Engineering
The University of Washington
gribble@cs.washington.edu

Abstract

This paper argues that a common design paradigm for systems is fundamentally flawed, resulting in unstable, unpredictable behavior as the complexity of the system grows. In this flawed paradigm, designers carefully attempt to predict the operating environment and failure modes of the system in order to design its basic operational mechanisms. However, as a system grows in complexity, the diffuse coupling between the components in the system inevitably leads to the butterfly effect, in which small perturbations can result in large changes in behavior. We explore this in the context of distributed data structures, a scalable, cluster-based storage server. We then consider a number of design techniques that help a system to be robust in the face of the unexpected, including overprovisioning, admission control, introspection, adaptivity through closed control loops. Ultimately, however, all complex systems eventually must contend with the unpredictable. Because of this, we believe systems should be designed to cope with failure gracefully.

1. Introduction

As the world grows more dependent on complex computing systems (such as scalable web sites, or even the Internet itself), it has become increasingly evident that these systems can exhibit unpredictable behavior when faced with unexpected perturbations to their operating environment. Such perturbations can be small and innocuous, but due to latent flaws in the design of the system combined with widespread coupling between its components, the effects of small perturbations may be large and destructive, possibly rendering the system inoperative.

For example, in [5], Floyd and Jacobson demonstrated that periodic signals (such as router broadcasts) in the Internet tend to become abruptly synchronized, leading to patterns of loss and delays. As another example, in [4], Druschel and Banga demonstrate that with web servers running on traditional interrupt-driven operating systems, a slight increase in load beyond the capacity of the server can drive the server into a persistent state of livelock, drastically reducing its effective throughput. As a third example, in [1], Arpaci-Dusseau et al. demonstrate that with conventional software architectures, the difference in performance resulting from placing data on the inner tracks vs. outer tracks of a single disk can affect the global throughput of an eight node cluster of workstations by up to 50%. A final example is that of BGP "route flap storms" [11, 12]: under conditions of heavy routing instability, the failure of a single router can instigate a storm of pathological routing oscillations. According to [11], there have been cases of flap storms that have caused extended Internet outages for millions of network customers.

By their very nature, large systems operate through the complex interaction of many components. This interaction leads to a pervasive coupling of the elements of the system; this coupling may be strong (e.g., packets sent between adjacent routers in a network) or subtle (e.g., synchronization of routing advertisements across a wide area network). A well-known implication of coupling in complex systems is the butterfly effect [14]: a small perturbation to the system can result in global change.

Avoiding Fragility

A common goal that designers of complex systems strive for is robustness. Robustness is the ability of a system to continue to operate correctly across a wide range of operational conditions, and to fail gracefully outside of that range. In this paper, we argue against a seemingly common design paradigm that attempts to achieve robustness by predicting the conditions in which a system will operate, and then carefully architecting the system to operate well in those (and only those) conditions. We claim that this design technique is akin to *precognition*: attempting to gain knowledge of something in advance of its actual occurrence.

As argued above, it is exceedingly difficulty to completely understand all of the interactions in a complex sys-

tem *a priori*. It is also effectively impossible to predict all of the perturbations that a system will experience as a result of changes in environmental conditions, such as hardware failures, load bursts, or the introduction of misbehaving software. Given this, **we believe that any system that attempts to gain robustness solely through precognition is prone to fragility**.

In the rest of this paper, we expore this hypothesis by presenting our experiences from building a large, complex cluster-based storage system. We show that although the system behaved correctly when operating within its design assumptions, small perturbations sometimes led to the violation of these assumptions, which in turn lead to system-wide failure. We then describe several design techniques that can help systems to avoid this fragility. All of these techniques have existed in some form in previous systems, but our goal in this paper is to consolidate these techniques as a first step towards the design of more robust systems.

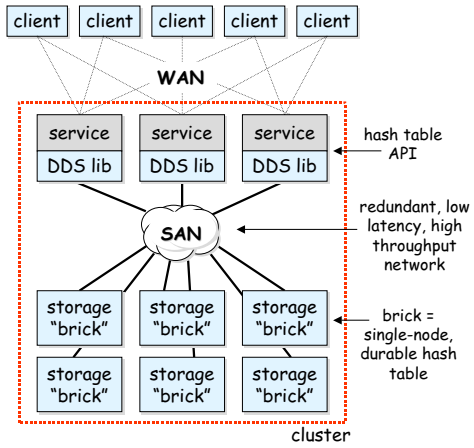

Figure 1. DDS architecture: each box in the diagram represents a software process. In the simplest case, each process runs on its own physical machine in a cluster, however there is nothing preventing processes from sharing physical machines.

2. DDS: A Case Study

In [7], we presented the design and implementation of a scalable, cluster-based storage system called a *distributed data structure (DDS)*. A DDS, shown in Figure 1, is a high-capacity, high-throughput virtual hash table that is partitioned and replicated across many individual storage nodes called bricks. DDS clients (typically Internet services such as web servers) invoke operations on it through a library that acts as a two-phase commit coordinator across replicas affected by the operation. These two-phase commits are used to achieve atomicity of all operations and one-copy equivalence across the entire cluster.

The design philosophy we used while building the DDS was to choose a carefully selected set of reasonable operational assumptions, and then to build a suite of mechanisms and an architecture that would perform robustly, scalably, and efficiently given our assumptions. Our design strategy was essentially predictive: based on extensive experience with such systems, we attempted to reason about the behavior of the software components, algorithms, protocols, and hardware elements of the system, as well as the workloads it would receive. In other words, we largely relied on precognition while designing mechanisms and selecting operating assumptions to gain robustness in our system.

Within the scope of our assumptions, the DDS design proved to be very successful. We were able to scale the number of nodes in the system across two orders of magnitude, and we observed a corresponding linear scaling in performance. We also demonstrated fault-tolerance by deliberately inducing faults in the system and showing that the storage remained available and consistent. However, as we operated the system for a period of more than a year, we observed several very unexpected performance and behavioral anomalies. In all cases, the anomalies arose because of an unforeseen perturbation to the system that resulted in the violation of one of our operating assumptions; the consequences of these violations were usually severe.

In this section of the paper, we describe several of the more interested and unexpected behavioral anomalies that we encountered over the one or two years' worth of experience we had with this system. Some may choose to consider these anomalies simply as bugs in the design of the system, arising from lack of foresight or naïvety on the part of its designers. We argue, however, that these "bugs" all shared similar properties: they were extremely hard to predict, they arose from subtle interactions between many components or layers in the system, and they bugs led to severe implications in our system (specifically, the violation of several operating assumptions which in turn led to system unavailability or data loss).

2.1. Garbage Collection Thrashing and Bounded Synchrony

Various pieces in the DDS relied on timeouts to detect the failure of remote components. For example, the two-phase commit coordinators used timeouts to identify the deaths of subordinates. Because of the low-latency (10-100 μs), redundant network in the cluster, we chose to set our timeout values to several seconds, which is four orders of magnitude higher than the common case round trip time of messages in the system. We then assumed that components that didn't respond within this timeout had failed: we assumed *bounded synchrony*.

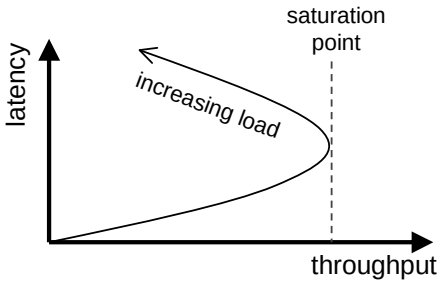

Figure 2. Performance with GC thrashing: this graph depicts the (parametric) curve of latency and throughput as a function of load. As load increases, so does throughput and latency, until the system reaches a saturation point. Beyond this, additional load results in GC thrashing, and a decrease in throughput with a continued latency increase. After saturating, the system falls into a hole out of which it must "climb".

The DDS was implemented in Java, and therefore made use of garbage collection. The garbage collector in our JVM was a mark-and-sweep collector; as a result, as more active objects were resident in the JVM heap, the duration that the garbage collector would run in order to reclaim a fixed amount of memory would increase. If the DDS were operating near saturation, slight (random) fluctuations in the load received by bricks in the system would increase the pressure on their garbage collector, causing the effective throughput of these bricks to drop. Because the offered load to the system is independent of this effect, this would cause the degraded bricks to "fall behind" relative to its peers, leading to more active objects in its heap and a further degradation in performance. This catastrophe leads to a performance response of the system as shown in Figure 2.

Once the system was pushed past saturation, the catastrophe would cause the affected node to slow down until its latency exceeded the timeouts in the system. Thus, the presence of garbage collection would cause the system to violate the assumption of bounded synchrony as it approached and then exceeded saturation.

2.2. Slow Leaks and Correlated Failure

We used replication in the DDS to gain fault-tolerance: by replicating data in more than one location, we gained the ability to survive the faults of individual components. We further assumed that *failures would be independent*, and therefore the probability that multiple replicas would simultaneously fail is vanishingly small.

For the most part, this assumption was valid. We only encountered two instances of correlated failure in our DDS. The first was due to blatant, naive bugs that would cause bricks to crash; these were quickly fixed. However, the second was much more subtle. Our bricks had a latent race condition in their two-phase commit handling code that didn't affect correctness, but which had the side-effect of a causing a memory leak. Under full load, the rareness of this race condition caused memory to leak at the rate of about 10KB/minute. We configured each brick's JVM to limit its heap size to 50MB. Given this leak rate, the bricks' heaps would fill after approximately 3 days.

Whenever we launched our system, we would tend to launch all bricks at the same time. Given roughly balanced load across the system, all bricks therefore would run out of heap space at nearly the same time, several days after they were launched. We also speculated that our automatic failover mechanisms exacerbated this situation by increasing the load on a replica after a peer had failed, increase the rate at which the replica leaked memory.

We did in fact observe this correlated failure in practice: until we isolated and repaired the race condition, our bricks would fail predictably within 10-20 minutes of each other. The uniformity of the workload presented to the bricks was itself the source of coupling between them; this coupling, when combined with a slow memory leak, lead to the violation of our assumption of independent failures, which in turn caused our system to experience unavailability and partial data loss.

2.3. Unchecked Code Dependencies and Fail-Stop

As mentioned above, we used timers in order to detect failures in our system. If a timer expired, we assumed that the corresponding entity in the system had crashed; therefore, in addition to assuming bounded synchrony, we also assumed nodes would behave in a *fail-stop* manner (i.e., a node that failed to respond to one message would never again respond to any message).

To gain high performance from our system given the highly concurrent workload, we implemented our bricks using an event-driven architecture: the code was structured as a single thread executing in an event loop. To ensure the liveness of the system, we strove to ensure that all long-latency operations (such as disk I/O) were performed asynchronously. Unfortunately, we failed to notice that portions of our code that implemented a network session layer made use of blocking (synchronous) socket `connect()` routines in the Java class library. This session layer was built to attempt to automatically reinstantiate a network connection if it was broken. The main event-handling thread therefore could be surreptitiously borrowed by the session layer to forge transport connections.

On several occasions, we noticed that some of our bricks would seize inexplicably for a multiple of 15 minutes (i.e., 15 minutes, 30 minutes, 45 minutes, ...), and then resume execution, egregiously violating our fail-stop assumption. After much investigation, we traced this problem down to

a coworker that was attempting to connect a machine that was behind a firewall to the cluster. The firewall was silently dropping incoming TCP syn packets, causing session layers to block inside the `connect()` routine for 15 minutes for each connection attempt made to that machine.

While this error was due to our own failure to verify the behavior of code we were using, it serves to demonstrate that the low-level interaction between independently built components can have profound implications on the overall behavior of the system. A very subtle change in behavior (a single node dropping incoming SYN packets) resulted in the violation of our fail-stop assumption across the entire cluster, which eventually lead to the corruption of data in our system.

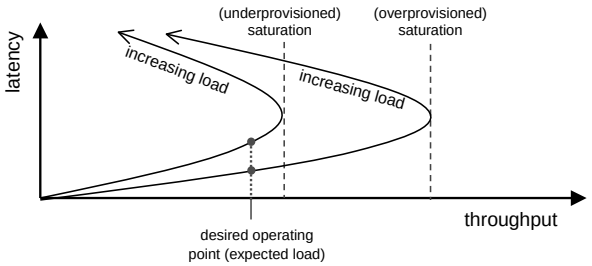

Figure 3. An overprovisioned system: by overprovisioning relative to the expected load, the system has slack: it can withstand unexpected bursts of load without falling into the "hole" associated with operating beyond saturation.

3. Towards Robust Complex Systems

The examples in the previous section served to illustrate a common theme: small changes to a complex, coupled system can result in large, unexpected changes in behavior, possibly taking the system outside of its designers' expected operating regime. In this section, we outline a number of design strategies that help to make systems more robust in the face of the unexpected. None of these strategies are a panacea, and in fact, some of them may add significant complexity to a system, possibly introducing more unexpected behavior. Instead, we present them with the hope of stimulating thought in the systems community for dealing with this increasingly common problem: we believe that an important focus for future systems research is building systems that can adapt to unpredictably changing environments, and that these strategies are a useful starting point for such investigation.

Systematic overprovisioning: as exemplified in Section 2.1, systems tend to become less stable when operating near or beyond the threshold of load saturation. As a system approaches this critical threshold, there is less "slack" in the system to make up for unexpected behavior: as a result, the system becomes far less forgiving (i.e., fragile). A simple technique to avoid this is to deliberately and systematically overprovision the system; by doing so, the system is ensured to operate in a more forgiving regime (Figure 3).

Overprovisioning doesn't come for free; an overprovisioned system is underutilizing its resources, and it is tempting to exploit this underutilization instead of adding more resources as the load on the system grows. In fact, it is only when the system nears saturation that many well-studied problems (such as load balancing) become interesting. However, we believe it is usually better to have a well-behaved, overprovisioned system than a poorly behaved, fully utilized one, especially given that computing resources are typically inexpensive relative to the cost of human designers.

However, overprovisioning contains the implicit assumption that the designers can accurately predict the expected operating regime of the system. As we've argued in Section 1, this assumption is often false, and it can lead to unexpected fragility.

Use admission control: given that systems tend to become unstable as they saturate, a useful technique is to use admission control to reject load as the system approaches the saturation point. Of course, to do this requires that the saturation point is identifiable; for large systems, the number of variables that contribute to the saturation point may be large, and thus statically identifying the saturation point may be difficult. Admission control often can be added to a system as an "orthogonal" or independent component. For example, high throughput web farms typically use layer 5 switches for both load balancing and admission control.

To reject load still requires resources from the system; each incoming task or request must be turned back, and the act of turning it back consumes resources. Thus, we view systems that perform admission control as having two classes of service: normal service, in which tasks are processed, and an extremely lightweight service, in which tasks are rejected. It is important to realize that the lightweight service has a response curve similar to that shown in Figure 2: a service, even if it is performing admission control, can saturate and then collapse. This effect is called livelock, and it is described in [4]. Admission control simply gives a system the ability to switch between two response curves, one for each class of service.

Build introspection into the system: an introspective system is one in which the ability to monitor the system is designed in from the beginning. As argued in [2], by building measurement infrastructure into a system, designers are much more readily able to monitor, diagnose, and adapt to aberrant behavior than in a black-box system. While this may seem obvious, consider the fact that the Internet and many of its protocols and mechanisms do not include the ability to introspect. As a result, researchers have often

found it necessary to subvert features of existing protocols [9, 15], or to devise cunning mechanisms to deduce properties of network [13]. We believe that introspection is a necessary property of a system for it to be both managable and for its designers and operators to be able to help it adapt to a changing environment.

Introduce adaptivity by closing the control loop: the usual way for systems to evolve over time is for their designers and operators to measure its current behavior, and then to correspondingly adapt its design. This is essentially a control loop, in which the human designers and operators form the control logic. This loop operates on a very long timescale; it can take days, weeks, or longer for humans to adapt a complex system.

However, an interesting class of systems are those which include internal control loops. These systems incorporate the results of introspection, and attempt to adapt control variables dynamically to keep the system operating in a stable or well-performing regime. This notion of adaptation is important even if a system employs admission control or overprovisioning, because *internal* as well as external perturbations can affect the system. For example, modern disks occasionally perform thermal recalibration, vastly affecting their performance; if a system doesn't adapt to this, transient periods of poor performance or even instability may result.

Closed control loops for adaptation have been exploited in many systems, including TCP congestion control, online adaptation of query plans in databases [8, 10], or adaptive operating systems that tuning their policies or run-time parameters to improve performance [16]. All such systems have the property that the component performing the adaptation is able to hypothesize somewhat precisely about the effects of the adaptation; without this ability, the system would be "operating in the dark", and likely would become unpredictable. A new, interesting approach to hypothesizing about the effects of adaptation is to use statistical machine learning; given this, a system can experiment with changes in order to build up a model of their effects.

Plan for failure: even if a system employs all of the above strategies, as it grows sufficiently complex, unexpected perturbations and failure modes inevitably will emerge. *Complex systems must expect failure and plan for it accordingly*.

Planning for failure might imply many things: systems may attempt to minimize the damage caused by the failure by using robust abstractions such as transactions [6], or the system may be constructed so that losses are acceptable to its users (as is the case with the web). Systems may attempt to minimize the amount of time in which they are in a failure state, for example by checkpointing the system in known good states to allow for rapid recovery. In addition, systems may be organized as several weakly coupled compartments, in the hope that failures will be contained within a single compartment. Alternatively, systems may stave off failure by proactively "scrubbing" their internal state to prevent it from accumulating inconsistencies [3].

4. Summary

In this paper, we have argued that a common design paradigm for complex systems (careful design based on a prediction of the operating environment, load, and failures that the system will experience) is fundamentally fragile. This fragility arises because the diffuse coupling of components within a complex systems makes them prone to completely unpredictable behavior in the face of small perturbations. Instead, we argue that a different design paradigm needs to emerge if we want to prevent the ever-increasing complexity of our systems from causing them to become more and more unstable. This different design paradigm is one in which systems are given the best possible chance of stable behavior (through techniques such as overprovisioning, admission control, and introspection), as well as the ability to adapt to unexpected situations (by treating introspection as feedback to a closed control loop). Ultimately, systems must be designed to handle failures gracefully, as complexity seems to lead to an inevitable unpredictability.

In the future, we hope to explore the rich design space associated with robust, complex systems. Our plans include evaluating and extending the techniques identified in this paper in the context of adaptive, wide-area information delivery systems, such as caching hierarchies, content distribution networks, and peer-to-peer content sharing systems.

References

[1] R. H. Arpaci-Dusseau, E. Anderson, N. Treuhaft, D. E. Culler, J. M. Hellerstein, D. A. Patterson, and K. Yelick. Cluster I/O with River: Making the Fast Case Common. In *Proceedings of the Sixth Workshop on I/O in Parallel and Distributed Systems (IOPADS '99)*, May 1999.

[2] A. Brown, D. Oppenheimer, K. Keeton, R. Thomas, J. Kubiatowicz, and D. A. Patterson. ISTORE: Introspective storage for data intensive network services. In *Proceedings of the 7th Workshop on Hot Topics in Operating Systems (HotOS-VII)*, March 1999.

[3] G. Candea and A. Fox. Reboot-based High Availability. In *Presented in the WIP Session of the 4th Symposium for Operating System Design and Implementation (OSDI 2000)*, San Diego, CA, October 2000.

[4] P. Druschel and G. Banga. Lazy Receiver Processing: A Network Subsystem Architecture for Server Systems. In *Proceedings of the USENIX 2nd Symposium on Operating System Design and Implementation (OSDI '96)*, Seattle, WA, USA, October 1996.

[5] S. Floyd and V. Jacobson. The synchronization of periodic routing messages. *ACM Transactions on Networking*, 2(2):122–136, April 1994.

[6] J. Gray. The Transaction Concept: Virtues and Limitations. In *Proceedings of VLDB*, Cannes, France, September 1981.

[7] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable, Distributed Data Structures for Internet Service Construction. In *Proceedings of the 4th USENIX Symposium on Operating System Design and Implementation (OSDI 2000)*, San Diego, California, USA, October 2000.

[8] Z. G. Ives, M. Friedman, D. Florescu, A. Y. Levy, and D. Weld. An Adaptive Query Execution System for Data Integration. In *Proceedings of SIGMOD 1999*, June 1999.

[9] V. Jacobsen. Traceroute. `ftp://ftp.ee.lbl.gov/traceroute.tar.Z`, 1989.

[10] N. Kabra and D. J. DeWitt. Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans. In *Proceedings of SIGMOD 1998*, Seattle, WA, June 1998.

[11] C. Labovitz, G. R. Malan, and F. Jahanian. Internet routing instability. In *Proceedings of ACM SIGCOMM '97*, Cannes, France, September 1997.

[12] C. Labovitz, G. R. Malan, and F. Jahanian. Origins of internet routing instability. In *Proceedings of IEEE INFOCOMM '99 Conference*, New York, New York, March 1999.

[13] K. Lai and M. Baker. Measuring Link Bandwidths Using a Deterministic Model of Packet Delay. In *Proceedings of the 2000 ACM SIGCOMM Conference*, Stockholm, Sweden, August 2000.

[14] E. N. Lorentz. *The Essence of Chaos*. University of Washington Press, Seattle, WA, 1993.

[15] S. Savage. Sting: a TCP-based Network Measurement Tool. In *Proceedings of the 1999 USENIX Symposium on Internet Technologies and Systems (USITS '99)*, October 1999.

[16] M. Seltzer and C. Small. Self-monitoring and self-adapting operating systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VII)*, May 1997.