Studying and using failure data from large-scale Internet services

David Oppenheimer and David A. Patterson
University of California at Berkeley, EECS Computer Science Division
387 Soda Hall #1776, Berkeley, CA, 94720-1776, USA
{davidopp,patterson}@cs.berkeley.edu

Abstract

Large-scale Internet services are the newest and arguably the most commercially important class of systems requiring 24x7 availability. As a result, very little information has been published about their causes of failure. In an attempt to address this deficiency, we have analyzed detailed failure reports from three large-scale Internet services. Our goals are to (1) identify the major factors contributing to user-visible failures, (2) evaluate the (potential) effectiveness of various techniques for preventing and mitigating service failure, and (3) build a fault model for service-level dependability and recovery benchmarks. Our initial results indicate that operator error and network problems are the leading contributors to user-visible failures, that failures in custom-written front-end software are significant, and that online testing and more thoroughly exposing and handling component failures would reduce failure rates in at least one service.

1. Introduction

The number and popularity of large-scale Internet services such as Google, MSN, and Yahoo! have grown significantly in recent years. Moreover, such services are poised to increase in importance as they become the repository for data in ubiquitous computing systems and the platform upon which new global-scale services and applications are built. These services' large scale and need for 24x7 operation have led their designers to incorporate a number of techniques for achieving high availability. Nonetheless, failures still occur.

While the architects and operators of these services might see such problems as failures on their part, these system failures provide important lessons for the systems community about why large-scale systems fail, and what techniques are or would be effective in preventing component failures from causing user-visible service failures. In an attempt to answer the question "Why do Internet services fail and what can be done about it?" we have studied the architecture of, and 62 post-mortem reports of user-visible failures from, three large-scale Internet services. In this paper we describe three initial directions for using this data. First, we identify which service components are most failure-prone, so that service operators and researchers know what areas most need improvement. Second, we examine the applicability of a number of failure mitigation techniques to the actual failures we observed. Third, we suggest using the failure data to derive a fault model for service-level dependability and recovery benchmarks.

2. Survey sites and methodology

We studied an online service/Internet portal (*Online*), a global content hosting service (*Content*), and a high-traffic, read-mostly Internet service (*ReadMostly*). Physically, all of these services are housed in geographically distributed colocation facilitie and use commodity hardware and networks. Architecturally, they are built from a load-balancing tier, a stateless front-end tier, and a back-end tier that stores persistent data; and they use multiple levels of redundancy and load balancing for performance and availability. Operationally, they use primarily custom-written software to provide and administer the service; they undergo frequent software upgrades and configuration updates; and they operate their own 24x7 Systems Operations Centers staffed by operators who monitor the service and respond to problems.

The services we examine differ slightly in their worklods and node hardware. *Online* and *ReadMostly* each receive about 100 million hits per day, while *Content* receives about 7 million. The ratio of writes to reads in *Online* and *Content* is moderate, while that in *ReadMostly* is (as the name implies) low. Finally, all three services use x86-based PCs running open-source operating systems throughout their service, except for *Online* which uses Network Appliance fileservers for back-end storage.

Because we are interested in *why* and *how* large-scale Internet services fail, we studied individual problem reports rather than aggregate availability statistics. The operations staff of all three services use problem-tracking databases to record information about component and service failures. Two of the services (*Online* and *Content*) gave us access to these databases, and one of the services (*ReadMostly*) gave us access to the problem post-mortem reports written after every major user-visible service failure. For *Online* and *Content*, we defined a user-visible failure as one that theoretically prevents an end-user from accessing the service or a part of the service (even if the user is given a reasonable error message) or that significantly degrades a user-visible aspect of system performance[1].

We studied 16 failures from *Online*, and 23 from each of *ReadMostly* and *Content*. These problems corresponded to the user-visible failures during four months at *Online*, six months at *ReadMostly*, and a month at *Content*. Note that it is not fair to compare the services directly, as the functionality of the custom-written software at *Online* is richer than that of *ReadMostly*, and *Content* is more complicated than either of the other two services (*e.g.*, *Content* counts as service failures not only failures of equipment in its colocation sites, but also failures of client proxy nodes it distributes to its users, including situations in which those client proxies cannot communicate with the colocation facilities). But because we studied all user-visible failures for each service, and used approximately the same definition of failure for choosing (or having chosen for us) the problems to examine from each of the services, we believe our conclusions as to relative failure causes are meaningful.

We attributed the cause of a system failure to the *first component that failed* in the chain of events leading up to the service failure, and the nature of that component's failure to *the type of flaw (fault) that was the root cause of the failure*. In particular, the failing component was categorized as **front-end node**, **back-end node**, or **network**, and the underlying cause of the failure as **hardware**, **software**, **environment**, **operator error**, or **unknown** (indeterminable). Note that the underlying flaw may have remained latent for an arbitrary period of time, only to cause a component to fail when another component subsequently failed or the service was used in a particular way for the first time. *Front-end nodes* are those initially contacted by end-user clients, as well as the client proxy nodes used by *Content*. Using this definition, front-end nodes do not store persistent data (though they may cache data), while back-end nodes do store persistent data. The "business logic" of traditional three-tier systems terminology is therefore part of *front-end*, a reasonable decision because these services integrate their service logic with the code that receives and replies to user client requests.

Most problems were relatively easy to map into this two-dimensional component-flaw space, except for wide-area network problems. Such problems affected the links between colocation facilities for all services, and, in the case of *Content*, also between customer sites and colocation facilities. Because the root cause of such problems often lay somewhere in the network of an Internet Service Provider to whose records we did not have access, the best we could do with such problems was to label them as **network** and due to a flaw of **unknown** cause.

3. Analysis of problem causes

Classifying the 62 problems we reviewed has allowed us to make a number of observations about the causes of user-visible service failures. The data from which we make these observations are summarized in Table 1, which breaks down problem causes by the part of the service containing the root cause, and in Table 2, which breaks down problem causes by the component that failed and the underlying cause of the failure.

Table 1 shows that contrary to conventional wisdom, front-end machines can be a significant source of failure. In the services we studied, this was largely due to the complexity of the service software running on those machines and the complexity of configuring and administering them.

Table 2 shows that operator error and networking problems are a significant cause of failure. In *Online and Content*, operator error caused more failures than did node hardware or software failures, while in *ReadMostly* networking problems caused more failures than did node hardware or software problems. Networking failures were prominent because networks tended to be a single point of failure--services often use only one network switch to connect their server racks to the colocation site's network, and colocation facilities often used only one Internet Service Provider (indeed, such facilities are often owned and operated by an ISP). This latter fact means that even colocation sites with multiple physical Internet links may be adversely impacted by a single upstream Internet failure. We also observed that while geographic redundancy tends to reduce the incidence of complete service unavailability, many Internet problems nonetheless become user-visible

	front-end	**back-end**	**network**
Online	63%	25%	13%
Content	57%	17%	26%
RdMostly	4%	9%	87%

Table 1: Failure cause by part of service.

[1] "Significantly degrades a user-visible aspect of system performance" is admittedly a vaguely-defined metric. A more precise definition of failure would involve correlating component failure reports with degradation in some aspect of observed system performance such as response time. But even where these services measured and archived response times for the time period studied, we are not guaranteed to detect all user-visible failures, due to the periodicity and placement in the network of the probes. Thus our definition of *user-visible* is problems that were *potentially* user-visible, *i.e.*, visible if a user tried to access the service during the failure.

because of non-fail-stop failure modes of Internet links, and delays in detecting a problem and then updating global load balancing tables. Colocation facilities did appear effective in eliminating "environmental" problems--only one environmental problem in our study led to a user-visible failure, and that problem was a power failure at one of *Content's* customer sites, not at a colocation site.

4. Techniques for mitigating failure

Given that user-visible failures are inevitable despite these services' attempts to prevent them, how could the failures have been avoided or their impact reduced? To answer this question, we analyzed each of the 62 problem reports, asking whether any of a number of techniques that have been suggested for improving availability could potentially

- prevent the original **component fault** (*e.g.,* a double-bit memory error, a software bug, an error in a configuration file, or an incorrect operator command),
- prevent a component fault from turning into a **component failure**,
- reduce the severity of degradation in user-perceived system quality of service (QoS) due to a component failure (*i.e.,* reduce the degree to which a **system failure** is observed),
- decrease the **TTD:** time from component failure to detection of the failure,
- decrease the **TTR:** time from component failure detection to component repair (*i.e.,* the time during which system QoS is degraded).

The above categories can be viewed as a state machine or timeline, with component fault leading to component failure, causing a user-visible system failure; the component failure is eventually detected and repaired, returning the system to its failure-free QoS.

The techniques we investigated for their potential effectiveness were

- **redundancy:** replicating data, computational functionality, and/or networking functionality [2]
- **isolation/partitioning:** increasing isolation between software components, to reduce failure propagation
- **restart:** periodic rebooting of hardware and restarting of software
- **fault injection and load testing:** explicitly testing failure-handling code and system response to overload by artificially introducing failure and overload scenarios, either into components before deployment or into the production system
- **testing:** testing the system for correct behavior given normal inputs, either in components before deployment or in the production system
- **config:** using tools to check that low-level configuration files meet sanity constraints
- **exposing:** better exposing software and hardware component failure to other modules and/or to a monitoring system

Table 3 shows the number of problems from *Online's* problem tracking database for which use, or more use, of each technique could potentially have prevented the problem that directly caused the system to enter the corresponding failure state. A given technique generally addresses only one or a few system failure states; we have listed only those we consider feasible.

Note that if a technique prevents a problem from causing the system to enter some failure state, it also necessarily prevents the problem from causing the system to enter a subsequent failure state. For example, checking a configuration file might prevent a component fault, which therefore prevents the fault from turning into a system-level failure, a degradation in QoS, a need to detect the failure, and a need to repair the failure. However, our methodology only counts this as preventing a component fault, so as to more precisely pinpoint the effect of the technique. Finally, note that techniques that reduce time to detect or time to repair component failure reduce the overall service

	node op	net op	node hw	net hw	node sw	net sw	node unk	net unk	env
Online	44%	0%	13%	6%	31%	0%	0%	6%	0%
Content	35%	4%	0%	0%	26%	0%	9%	22%	4%
Read-Mostly	13%	9%	0%	17%	0%	26%	0%	35%	0%

Table 2: Failure cause by component and fault type. The component is described as node (node) or network (net), and fault type is described as operator error (op), hardware (hw), software (sw), unknown (unk), or environment. Operator and network failure are the leading causes of service failure.

loss experienced (we define the loose notion of “overall service loss” as the amount of QoS lost during the failure, multiplied by the duration of the failure).

From Table 3 we observe that a large number of the problems *Online* experienced might have been prevented or mitigated by more online testing, increased redundancy, and more thoroughly exposing and reacting to software and hardware failures. Automatic sanity checking of configuration files, and online fault and load injection, also appear to offer significant potential benefit.

Additional results from the three services, including an analysis of time-to-repair for the various types of failures, the causes of non-user-visible failures, and lessons from individual problem case studies, can be found in [3].

technique	system failure state avoided/ mitigated	# of instances potentially avoided
redundancy	system failure	8
isolation/part.	system failure	2
restart	component fail	1
pre-fault/load	component fault	2
online fault/load	component fail	3
pre-testing	component fault	1
online testing	component fail	11
config	component fault	3
expose/monitor	TTD	8
expose/monitor	TTR	9

Table 3: Potential benefit from using in *Online* various proposed techniques for avoiding or mitigating service failures. Nineteen problems were examined (rather than sixteen as in Section 3) because here we have also included problems whose sources were external to the service (*i.e.*, due to failure in an Internet site that *Online* uses to provide part of its service). Pre-fault/load refers to fault injection and load testing prior to system deployment, while online fault/load refers to such testing conducted in a production environment. Pre-testing and online testing having similar meanings, but for correctness testing. Those techniques that *Online* is already using are indicated in bold; in those cases we evaluate the benefit from using the technique more extensively.

5. Fault models for service-level benchmarks

In addition to indicating where to focus efforts for improving availability, and helping to evaluate the potential effectiveness of specific techniques, the failure data we have collected can be used to create a fault (or, to use our terminology, component failure) model for *service-level benchmarks*. Recent benchmarking efforts have focused on component-level dependability by observing single-node application or OS response to misbehaving disks, system calls, and the like. But because we found a significant contribution to service failure of human error (particularly multi-node configuration problems) and network (including WAN) problems, we suggest a more holisitc approach. In *service-level benchmarks,* a small-scale replica (or a physically or virtually isolated partition) of a service is created, and Quality of Service for a representative service workload mix is measured while representative component failures (*e.g.*, those described in this paper) are injected. To simplify this process, one might measure the QoS impact of individual component failures or multiple simultaneous failures, and then weight the degraded QoS response to these recovery events by either the relative frequency with which the different classes of component failure occur in the service being benchmarked, or using the representative proportions we found in our survey. As suggested in [1], the human operator role in both causing and repairing failures, and in conducting normal service administrative tasks, should be included .

6. Conclusion

From a study of 62 user-visible failures in three large-scale Internet services, we observe that front-ends are a more significant problem than is commonly believed, that operator error and network problems are leading contributors to user-visible failures, and that more thoroughly exposing and handling component failures would reduce failure rates in at least one service. Because human error and network problems dominante, we argue for *service-level benchmarks* that replicate a service’s hardware and software architecture, its component dependencies, its workload, its failure modes, and its human operator tasks.

References

[1] Brown, A., L. C. Chung, D. A. Patterson. Including the Human Factor in Dependability Benchmarks. *2002 DSN Workshop on Dependability Benchmarking,* 2002.

[2] J. Gray. Why Do Computers Stop and What Can Be Done About It? *Symposium on Reliability in Distributed Software and Database Systems,* 1986.

[3] D. Oppenheimer. Why do Internet services fail, and what can be done about it? UC Berkeley Technical Report UCB-CSD-02-1185, 2002.