Appears in *4th Usenix Symposium on Internet Technologies and Systems (USITS '03)*, 2003.

Why do Internet services fail, and what can be done about it?

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

Abstract

In 1986 Jim Gray published his landmark study of the causes of failures of Tandem systems and the techniques Tandem used to prevent such failures [6]. Seventeen years later, Internet services have replaced fault-tolerant servers as the new kid on the 24x7-availability block. Using data from three large-scale Internet services, we analyzed the causes of their failures and the (potential) effectiveness of various techniques for preventing and mitigating service failure. We find that (1) operator error is the largest cause of failures in two of the three services, (2) operator error is the largest contributor to time to repair in two of the three services, (3) configuration errors are the largest category of operator errors, (4) failures in custom-written front-end software are significant, and (5) more extensive online testing and more thoroughly exposing and detecting component failures would reduce failure rates in at least one service. Qualitatively we find that improvement in the maintenance tools and systems used by service operations staff would decrease time to diagnose and repair problems.

1. Introduction

The number and popularity of large-scale Internet services such as Google, MSN, and Yahoo! have grown significantly in recent years. Such services are poised to increase further 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.

Although 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 could prevent failures. In an attempt to answer the question "Why do Internet services fail, and what can be done about it?" we have studied over a hundred post-mortem reports of user-visible failures from three large-scale Internet services. In this paper we

- identify which service components are most failure-prone and have the highest Time to Repair (TTR), so that service operators and researchers can know what areas most need improvement;
- discuss in detail several instructive failure case studies;
- examine the applicability of a number of failure mitigation techniques to the actual failures we studied; and
- highlight the need for improved operator tools and systems, collection of industry-wide failure data, and creation of service-level benchmarks.

The remainder of this paper is organized as follows. In Section 2 we describe the three services we analyzed and our study's methodology. Section 3 analyzes the causes and Times to Repair of the component and service failures we examined. Section 4 assesses the applicability of a variety of failure mitigation techniques to the actual failures observed in one of the services. In Section 5 we present case studies that highlight interesting failure causes. Section 6 discusses qualitative observations we make from our data, Section 7 surveys related work, and in Section 8 we conclude.

2. Survey services and methodology

We studied a mature online service/Internet portal (*Online*), a bleeding-edge global content hosting service (*Content*), and a mature read-mostly Internet service (*ReadMostly*). Physically, all of these services are housed in geographically distributed colocation facilities and use commodity hardware and networks. Architecturally, each site is built from a load-balancing tier, a stateless front-end tier, and a back-end tier that stores persistent data. Load balancing among geographically distributed sites for performance and availability is achieved using DNS redirection in *ReadMostly* and using client cooperation in *Online* and *Content*.

Front-end nodes are those initially contacted by clients, as well as the client proxy nodes used by *Content*. Using this definition, front-end nodes do not store per-

service characteristic	Online	ReadMostly	Content
hits per day	~100 million	~100 million	~7 million
# of machines	~500, 2 sites	> 2000, 4 sites	~500, ~15 sites
front-end node architecture	Solaris on SPARC and x86	open-source OS on x86	open-source OS on x86
beck-end node architecture	Network Appliance filers	open-source OS on x86	open-source OS on x86
period of data studied	7 months	6 months	3 months
component failures	296	N/A	205
service failures	40	21	56

Table 1: Differentiating characteristics of the services described in this study.

sistent data, although they may cache or temporarily queue data. *Back-end nodes* store persistent data. The "business logic" of traditional three-tier system terminology is part of our definition of front-end, because these services integrate their service logic with the code that receives and replies to client requests.

The front-end tier is responsible primarily for locating data on back-end machine(s) and routing it to and from clients in *Content* and *ReadMostly*, and for providing online services such as email, newsgroups, and a web proxy in *Online*. In *Content* the "front-end" includes not only software running at the colocation sites, but also client proxy software running on hardware provided and operated by *Content* that is physically located at customer sites. Thus *Content* is geographically distributed not only among the four colocation centers, but also at about a dozen customer sites. The front-end software at all three sites is custom-written, and at *ReadMostly* and *Content* the back-end software is as well. Figure 1, Figure 2, and Figure 3 show the service architectures of *Content*, *Online*, and *ReadMostly*, respectively.

Operationally, all three services use primarily custom-written software to administer the service; they undergo frequent software upgrades and configuration updates; and they operate their own 24x7 System Operations Centers staffed by operators who monitor the service and respond to problems. Table 1 lists the primary characteristics that differentiate the services. More details on the architecture and operational practices of these services can be found in [17].

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 (which we call a *service 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]. Service failures are caused by component failures that are not masked.

Our base dataset consisted of 296 reports of component failures from *Online* and 205 component failures from *Content*. These component failures turned into 40 service failures in *Online* and 56 service failures in *Content*. *ReadMostly* supplied us with 21 service failures (and two additional failures that we considered to be

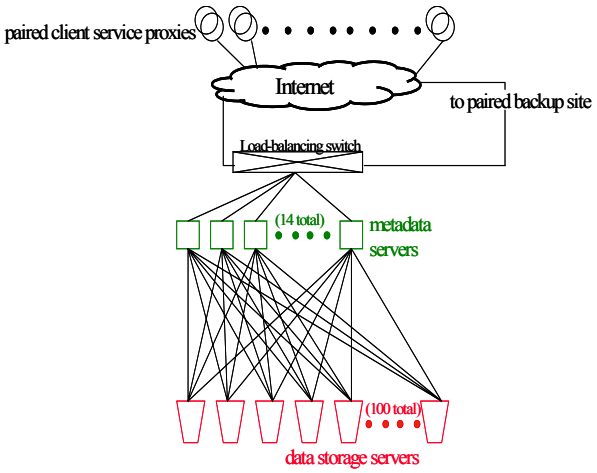

Figure 1: The architecture of one site of *Content*. Stateless metadata servers provide file metadata and route requests to the appropriate data storage servers. Persistent state is stored on commodity PC-based storage servers and is accessed via a custom protocol over UDP. Each cluster is connected to its twin site via the Internet.

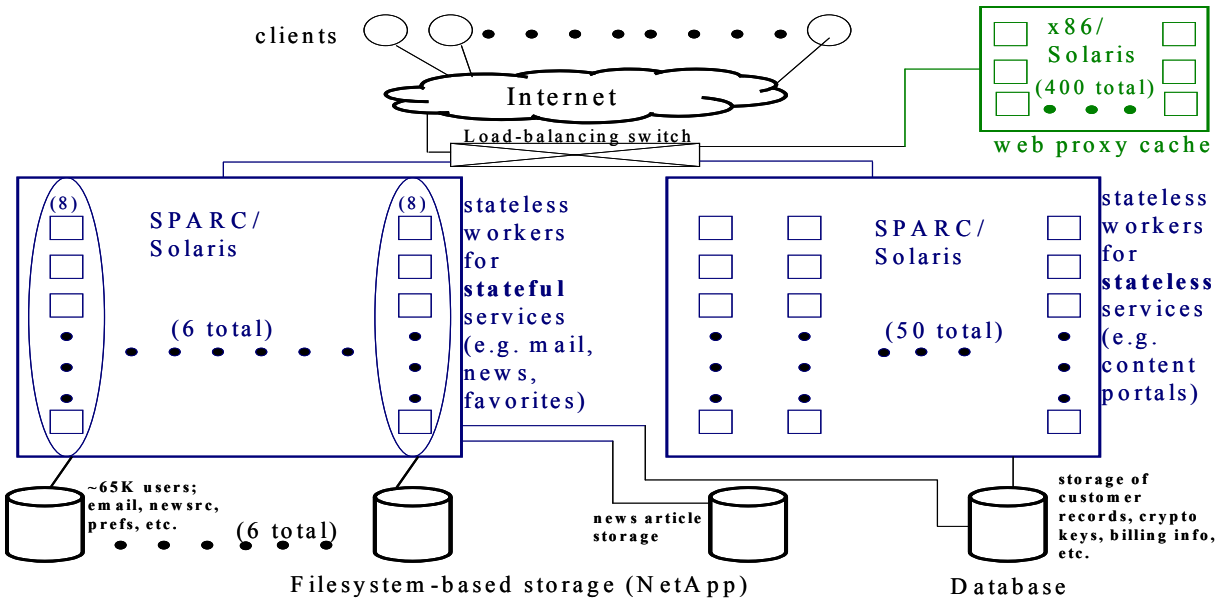

Figure 2: The architecture of one site of *Online*. Depending on the particular feature a user selects, the request is routed to any one of the web proxy cache servers, any one of 50 servers for stateless services, or any one of eight servers from a user's "service group" (a partition of one sixth of all users of the service, each with its own back-end data storage server). Persistent state is stored on Network Appliance servers and is accessed by worker nodes via NFS over UDP. This site is connected to a second site, at a collocation facility, via a leased network connection.

below the threshold to be deemed a service failure). These problems corresponded to 7 months at *Online*, 6 months at *ReadMostly*, and 3 months at *Content*. In classifying problems, we considered operators to be a component of the system; when they fail, their failure may or may not result in a service failure.

We attributed the cause of a service failure to the first component that failed in the chain of events leading up to the service failure. The *cause* of the component failure was categorized as node hardware, network hardware, node software, network software *(e.g.,* router or switch firmware), environment (*e.g.,* power failure), operator error, overload, or unknown. The *location* of that component was categorized as front-end node, back-end node, network, or unknown. Note that the underlying flaw may have remained latent for some time, only to cause a component to fail when the component was used in a particular way for the first time. Due to inconsistencies across the three services as to how or whether security incidents (*e.g.,* break-ins and denial of service attacks) were recorded in the problem tracking

[1] "Significantly degrades a user-visible aspect of system performance" is admittedly a vaguely-defined metric. It would be preferable to correlate failure reports with degradation in some aspect of user-observed Quality of Service, such as response time, but we did not have access to an archive of such metrics for these services. Note that even if a service measures and archives response times, such data is not guaranteed to detect all user-visible failures, due to the periodicity and placement in the network of the probes. In sum, 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.

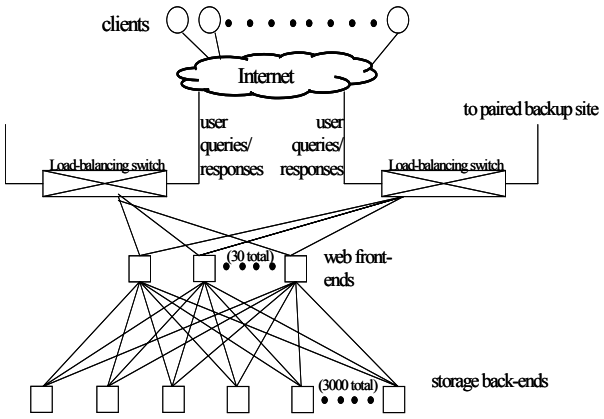

Figure 3: The architecture of one site of *ReadMostly*. A small number of web front-ends direct requests to the appropriate back-end storage servers. Persistent state is stored on commodity PC-based storage servers and is accessed via a custom protocol over TCP. A redundant pair of network switches connects the cluster to the Internet and to a twin site via a leased network connection.

databases, we ignored security incidents.

Most problems were relatively easy to map into this two-dimensional cause-location space, except for wide-area network problems. Network problems affected the links among colocation facilities for all services, and, for *Content*, also between client 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 the location as "network" and the cause as "unknown."

3. Analysis of failure causes

We analyzed our data on component and service failure with respect to four properties: how many component failures turn into service failures (Section 3.1); the relative frequency of each component and service failure root cause (Section 3.2); and the MTTR for service failures (Section 3.3).

3.1. Component failures to service failures

The services we studied all use redundancy in an attempt to mask component failures. That is, they try to prevent component failures from turning into end-user visible failures. As indicated by Figure 4 and Figure 5, this technique generally does a good job of preventing hardware, software, and network component failures from turning into service failures, but it is much less effective at masking operator failures. A qualitative analysis of the failure data suggests that this is because operator actions tend to be performed on files that affect the operation of the entire service or of a partition of the service, *e.g.,* configuration files or content files. Difficulties in masking network failures generally stemmed from the significantly smaller degree of network redundancy compared to node redundancy. Finally, we also observed that *Online*'s non-x86-based servers appeared to be less reliable than the equivalent, less expensive x86-based servers. Apparently more expensive hardware isn't always more reliable.

3.2. Service failure root cause

Next we examine the source and magnitude of service failures, categorized by the root cause location and component type. We augmented the data set presented in the previous section by examining five more months of data from *Online*, yielding 21 additional service failures, thus bringing our total to 61 for that service. (We did not analyze the component failures that did not turn into service failures from these five extra months, hence their exclusion from Section 3.1.)

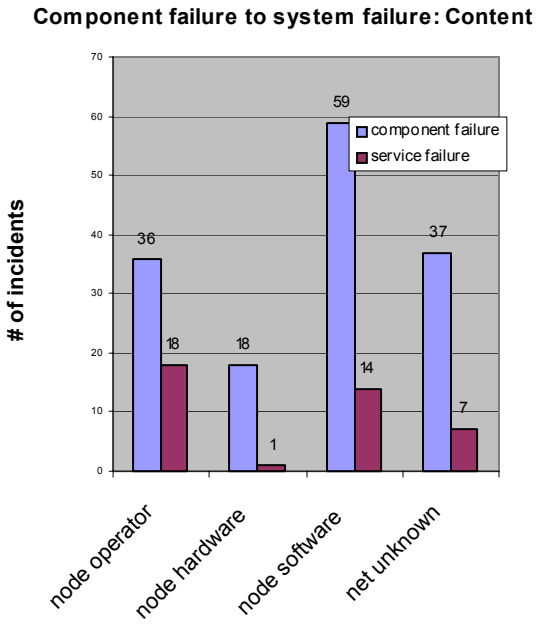

Figure 4: Number of component failures and resulting service failures for *Content*. Only those categories for which we classified at least six component failures (operator error related to node operation, node hardware failure, node software failure, and network failure of unknown cause) are listed. The vast majority of network failures in *Content* were of unknown cause because most network failures were problems with Internet connections between colocation facilities or between customer proxy sites and colocation facilities. For all but the "node operator" case, 24% or fewer component failures became service failures. Fully half of the 36 operator errors resulted in service failure, suggesting that operator errors are significantly more difficult to mask using the service's existing redundancy mechanisms.

Table 2 shows that contrary to conventional wisdom, front-end machines are a significant source of failure--in fact, they are responsible for more than half of the service failures in *Online* and *Content*. This fact was largely due to operator configuration errors at the application or operating system level. Almost all of the problems in *ReadMostly* were network-related; we attribute this to simpler and better-tested application software at that service, fewer changes made to the service on a day-to-day basis, and a higher degree of node redundancy than is used at *Online* and *Content*.

Table 3 shows that operator error is the leading cause of service failure in two of the three services.

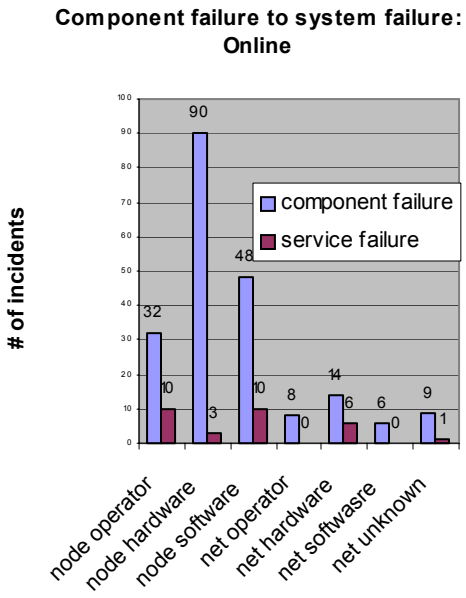

Figure 5: Number of component failures and resulting service failures for *Online*. Only those categories for which we classified at least six component failures (operator error related to node operation, node hardware failure, node software failure, and various types of network failure) are listed. As with *Content*, operator error was difficult to mask using the service's existing redundancy schemes. Unlike at *Content*, a significant percentage of network hardware failures became service failures. There is no single explanation for this, as the customer-impacting network hardware problems affected various pieces of equipment.

Operator error in all three services generally took the form of misconfiguration rather than procedural errors (*e.g.,* moving a user to the wrong fileserver). Indeed, for all three services, more than 50% (and in one case nearly 100%) of the operator errors that led to service failures were configuration errors. In general, operator errors arose when operators were making changes to the system, *e.g.,* scaling or replacing hardware, or deploying or upgrading software. A few failures were caused by operator errors during the process of fixing another problem, but those were in the minority--most operator errors, at least those recorded in the problem tracking databases, arose during normal maintenance.

	Front-end	**Back-end**	**Net-work**	**Un-known**
Online	77%	3%	18%	2%
Content	66%	11%	18%	4%
Read-Mostly	0%	10%	81%	9%

Table 2: Service failure cause by location. Contrary to conventional wisdom, most failure root causes were components in the service front-end.

Networking problems were a significant cause of failure in all three services, and they caused a surprising 76% of all service failures at *ReadMostly*. As mentioned in Section 3.1, network failures are less often masked than are node hardware or software failures. An important reason for this fact is that networks are often a single point of failure, with services rarely using redundant network paths and equipment within a single site. Also, consolidation in the collocation and network provider industries has increased the likelihood that "redundant" network links out of a collocation facility will actually share a physical link fairly close (in terms of Internet topology) to the data center. A second reason why networking problems are difficult to mask is that their failure modes tend to be complex: networking hardware and software can fail outright or more gradually, *e.g.,* become overloaded and start dropping packets. Combined with the inherent redundancy of the Internet, these

	Operator node	**Operator net**	**H/W node**	**H/W net**	**S/W node**	**S/W net**	**Unknown node**	**Unknown net**	**Environment**
Online	31%	2%	10%	15%	25%	2%	7%	3%	0%
Content	32%	4%	2%	2%	25%	0%	18%	13%	0%
Read-Mostly	5%	14%	0%	10%	5%	19%	0%	33%	0%

Table 3: Service failure cause by component and type of cause. The component is described as node or network, and failure cause is described as operator error, hardware, software, unknown, or environment. We excluded the "overload" category because of the very small number of failures caused.

failure modes generally lead to increased latency and decreased throughput, often experienced intermittently--far from the "fail stop" behavior that high-reliability hardware and software components aim to achieve [6].

Colocation facilities were effective in eliminating "environmental" problems--no environmental problems, such as power failure or overheating, led to service failure (one power failure did occur, but geographic redundancy saved the day). We also observed that overload (due to non-malicious causes) was insignificant.

Comparing this service failure data to our data on component failures in Section 3.1, we note that as with service failures, component failures arise primarily in the front-end. However, hardware and/or software problems dominate operator error in terms of component failure causes. It is therefore not the case that operator error is more frequent than hardware or software problems, just that it is less frequently masked and therefore more often results in a service failure.

Finally, we note that we would have been able to learn more about the detailed causes of software and hardware failures if we had been able to examine the individual component system logs and the services' software bug tracking databases. For example, we would have been able to break down software failures between operating system *vs*. application and off-the-shelf *vs*. custom-written, and to have determined the specific coding errors that led to software bugs. In many cases the operations problem tracking database entries did not provide sufficient detail to make such classifications, and therefore we did not attempt to do so.

3.3. Service failure time to repair

We next analyze the average Time to Repair (TTR) for service failures, which we define as the time from problem detection to restoration of the service to its pre-failure Quality of Service[1]. Thus for problems that are repaired by rebooting or restarting a component, the TTR is the time from detection of the problem until the reboot is complete. For problems that are repaired by replacing a failed component (*e.g.,* a dead network switch or disk drive), it is the time from detection of the problem until the component has been replaced with a functioning one. For problems that "break" a service functionally and that cannot be solved by rebooting (*e.g.,* an operator configuration error or a non-transient software bug), it is the time until the error is corrected, or until a workaround is put into place, whichever happens first. Note that our TTR incorporates both the time needed to diagnose the problem and the time needed to repair it, but not the time needed to detect the problem (since by definition a problem did not go into the problem tracking database until it was detected).

We analyzed a subset of the service failures from Section 3.2 with respect to TTR. We have categorized TTR by the problem root cause location and type. Table 4 is inconclusive with respect whether front-end failures take longer to repair than do back-end failures. Table 5 demonstrates that operator errors often take significantly longer to repair than do other types of failures; indeed, operator error contributed approximately 75% of all Time to Repair hours in both *Online* and *Content*.

We note that, unfortunately, TTR values can be misleading because the TTR of a problem that requires operator intervention partially depends on the priority the operator places on diagnosing and repairing the problem. This priority, in turn, depends on the operator's judgment of the impact of the problem on the service. Some problems are urgent, *e.g.,* a CPU failure in the machine holding the unreplicated database containing the mapping of service user IDs to passwords. In that case repair is likely to be initiated immediately. Other problems, or *even the same problem when it occurs in a different context*, are less urgent, *e.g.,* a CPU failure in one of a hundred redundant front-end nodes is likely to be addressed much more casually than is the database CPU failure. More generally, a problem's priority, as judged by an operator, depends on not only purely technical metrics such as performance degradation, but also on business-oriented metrics such as the importance of the customer(s) affected by the problem or the importance of the part of the service that has experienced the problem (*e.g.,* a service's email system may be considered to be more critical than the system that generates advertisements, or vice-versa).

	Front-end	**Back-end**	**Network**
Online	9.4 (16)	7.3 (5)	7.8 (4)
Content	2.5 (10)	14 (3)	1.2 (2)
Read-Mostly	N/A (0)	0.2 (1)	1.2 (16)

Table 4: Average TTR by part of service, in hours. The number in parentheses is the number of service failures used to compute that average.

[1] As with our definition of "service failure," restoration of the service to its pre-failure QoS is based not on an empirical measurement of system QoS but rather on inference from the system architecture, the component that failed, and the operator log of the repair process.

	Operator node	Operator net	H/W node	H/W net	S/W node	S/W net	Unknown node	Unknown net
Online	8.3 (16)	29 (1)	2.5 (5)	0.5 (1)	4.0 (9)	0.8 (1)	2.0 (1)	N/A (0)
Content	1.2 (8)	N/A (0)	N/A (0)	N/A (0)	0.2 (4)	N/A (0)	N/A (0)	1.2 (2)
Read-Mostly	0.2 (1)	0.1 (3)	N/A (0)	6.0 (2)	N/A (0)	1.0 (4)	N/A (0)	0.1 (6)

Table 5: Average TTR for failures by component and type of cause, in hours. The component is described as node or network, and failure cause is described as operator error, hardware, software, unknown, or environment. The number in parentheses is the number of service failures used to compute that average. We have excluded the "overload" category because of the very small number of failures due to that cause.

4. Techniques for mitigating failures

Given that user-visible failures are inevitable despite these services' attempts to prevent them, how could the service failures that we observed have been avoided, or their impact reduced? To answer this question, we analyzed 40 service failures from *Online*, asking whether any of a number of techniques that have been suggested for improving availability could potentially

- prevent the original component design flaw (fault)
- prevent a component fault from turning into a component failure
- reduce the severity of degradation in user-perceived QoS due to a component failure (*i.e.,* reduce the degree to which a service failure is observed)
- reduce the Time to Detection (TTD): time from component failure to detection of the failure
- reduce the Time to Repair (TTR): time from component failure detection to component repair. (This interval corresponds to the time during which system QoS is degraded.)

Figure 6 shows how these categories can be viewed as a state machine or timeline, with component fault leading to component failure, possibly causing a user-visible service failure; the component failure is eventually detected, diagnosed, and repaired, returning the system to its failure-free QoS.

The techniques we investigate for their potential effectiveness were

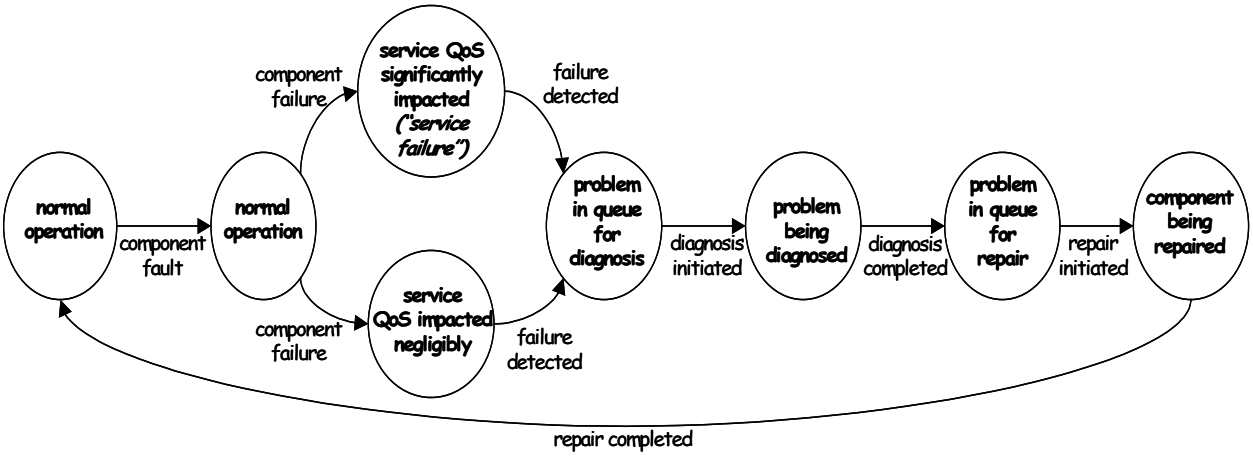

Figure 6: Timeline of a failure. The system starts out in *normal operation*. A *component fault*, such as a software bug, an alpha particle flipping a memory bit, or an operator misunderstanding the configuration of the system he or she is about to modify, may or may not eventually lead the affected component to fail. A *component failure* may or may not significantly impact the service's QoS. In the case of a simple component failure, such as an operating system bug leading to a kernel panic, the component failure may be automatically *detected* and *diagnosed* (*e.g.*, the operating system notices an attempt to twice free a block of kernel memory), and the *repair* (initiating a reboot) will be automatically initiated. A more complex component failure may require operator intervention for detection, diagnosis, and/or repair. In either case, the system eventually returns to normal operation. In our study, we use TTR to denote the time between "failure detected" and "repair completed."

- **correctness testing:** testing the system and its components for correct behavior before deployment or in production. Pre-deployment testing prevents component faults in the deployed system, and online testing detects faulty components before they fail during normal operation. Online testing will catch those failures that are unlikely to be created in a test situation, for example those that are scale- or configuration-dependent.
- **redundancy:** replicating data, computational functionality, and/or networking functionality [5]. Using sufficient redundancy often prevents component failures from turning into service failures.
- **fault injection and load testing:** testing error-handling code and system response to overload by artificially introducing failure and overload, before deployment or in the production system [18]. Pre-deployment, this aims to prevent components that are faulty in their error-handling or load-handling capabilities from being deployed; online, this detects components that are faulty in their error-handling or load-handling capabilities before they fail to properly handle anticipated faults and loads.
- **configuration checking:** using tools to check that low-level (*e.g.,* per-component) configuration files meet constraints expressed in terms of the desired high-level service behavior [13]. Such tools could prevent faulty configurations in deployed systems.
- **component isolation:** increasing isolation between software components [5]. Isolation can prevent a component failure from turning into a service failure by preventing cascading failures.
- **proactive restart:** periodic prophylactic rebooting of hardware and restarting of software [7]. This can prevent faulty components with latent errors due to resource leaks from failing.
- **exposing/monitoring failures:** better exposing software and hardware component failures to other modules and/or to a monitoring system, or using better tools to diagnose problems. This technique can reduce time to detect, diagnose, and repair component failures, and it is especially important in systems with built-in redundancy that masks component failures.

Of course, in implementing online testing, online fault injection, and proactive restart, care must be taken to avoid interfering with the operational system. A service's existing partitioning and redundancy may be exploited to prevent these operations from interfering with the service delivered to end-users, or additional isolation might be necessary.

Table 6 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 failure states we consider feasibly addressed by the corresponding technique. Because our analysis is made in retrospect, we tried to be particularly careful to assume a *reasonable* application of each technique. For example, using a trace of past failed and successful user requests as input to an online regression testing mechanism would be considered reasonable after a software change, whereas creating a bizarre combination of inputs that seemingly incomprehensibly triggers a failure would not.

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,

Technique	System state or transition avoided/ mitigated	instances potentially avoided/ mitigated
Online correctness testing	component failure	26
Expose/monitor failures	component being repaired	12
Expose/monitor failures	problem being diagnosed	11
Redundancy	service failure	9
Config. checking	component fault	9
Online fault/load injection	component failure	6
Component isolation	service failure	5
Pre-deployment fault/load injection	component fault	3
Proactive restart	component fail	3
Pre-deployment correctness testing	component fault	2

Table 6: Potential benefit from using in *Online* various proposed techniques for avoiding or mitigating failures. 40 service failures were examined, taken from the same time period as those analyzed in Section 3.3. Those techniques that *Online* is already using are indicated in italics; in those cases we evaluate the benefit from using the technique more extensively.

preventing a component fault prevents the fault from turning into a failure, a degradation in QoS, and a need to detect, diagnose, and repair the failure. Note that techniques that reduce time to detect, diagnose, or repair component failure reduce overall service loss experienced (*i.e.,* the amount of QoS lost during the failure multiplied by the length of the failure).

From Table 6 we observe that online testing would have helped the most, mitigating 26 service failures. The second most helpful technique, more thoroughly exposing and monitoring for software and hardware failures, would have decreased TTR and/or TTD in more than 10 instances. Simply increasing redundancy would have mitigated 9 failures. Automatic sanity checking of configuration files, and online fault and load injection, also appear to offer significant potential benefit. Note that of the techniques, *Online* already uses some redundancy, monitoring, isolation, proactive restart, and pre-deployment and online testing, so Table 6 underestimates the effectiveness of adding those techniques to a system that does not already use them.

Naturally, all of the failure mitigation techniques described in this section have not only benefits, but also costs. These costs may be financial or technical. Technical costs may come in the form of a performance degradation (*e.g.,* by increasing service response time or reducing throughput) or reduced reliability (if the complexity of the technique means bugs are likely in the technique's implementation). Table 7 analyzes the proposed failure mitigation techniques with respect to their costs. With this cost tradeoff in mind, we observe that the techniques of adding additional redundancy and better exposing and monitoring for failures offer the most significant "bang for the buck," in the sense that they help mitigate a relatively large number of failure scenarios while incurring relatively low cost.

Clearly, better online correctness testing could have mitigated a large number of system failures in *Online* by exposing latent component faults before they turned into failures. The kind of online testing that would have helped is fairly high-level self-tests that require application semantic information (*e.g.,* posting a news article and checking to see that it showed up in the newsgroup, or sending email and checking to see that it is received correctly and in a timely fashion). Unfortunately these kinds of tests are hard to write and need to be changed every time the service functionality or interface changes. But, qualitatively we can say that this kind of testing would have helped the other services we examined as well, so it seems a useful technique.

Online fault injection and load testing would likewise have helped *Online* and other services. This observation goes hand-in-hand with the need for better exposing failures and monitoring for those failures--online fault injection and load testing are ways to ensure that component failure monitoring mechanisms are correct and sufficient. Choosing a set of representative faults and error conditions, instrumenting code to inject them, and then monitoring the response, requires potentially even more work than does online correctness testing. Moreover, online fault injection and load testing require a performance- and reliability-isolated subset of the production service to be used, because of the threat they pose to the performance and reliability of the production system. But we found that, despite the best intentions, offline test clusters tend to be set up slightly differently than the production cluster, so the online approach appears to offer more potential benefit than does the offline version.

5. Failure case studies

In this section we examine in detail a few of the more instructive service failures from *Online*, and one failure from *Content* related to a service provided to the operations staff (as opposed to end-users).

Our first case study illustrates an operator error affecting front-end machines. In that problem, an operator at *Online* accidentally brought down half of the front-end servers for one service group (partition of users) using the same administrative shutdown com-

Technique	Implementation cost	Potential reliability cost	Performance impact
Online-correct	medium to high	low to moderate	low to moderate
Expose/monitor	medium	low (false alarms)	low
Redundancy	low	low	very low
Online-fault/load	high	high	moderate to high
Config	medium	zero	zero
Isolation	moderate	low	moderate
Pre-fault/load	high	zero	zero
Restart	low	low	low
Pre-correct	medium to high	zero	zero

Table 7: Costs of implementing failure mitigation techniques described in this section.

mand issued separately to three of the six servers. Only one technique, redundancy, could have mitigated this failure: because the service had neither a remote console nor remote power supply control to those servers, an operator had to physically travel to the colocation site and reboot the machines, leading to 37 minutes during which users in the affected service group experienced 50% performance degradation when using "stateful" services. Remote console and remote power supply control are a redundant control path, and hence a form of redundancy. The lesson to be learned here is that improving the redundancy of a service sometimes cannot be accomplished by further replicating or partitioning existing data or service code. Sometimes redundancy must come in the form of orthogonal redundancy, such as a backup control path.

A second interesting case study is a software error affecting the service front-end; it provides a good example of a cascading failure. In that problem, a software upgrade to the front-end daemon that handles username and alias lookups for email delivery incorrectly changed the format of the string used by that daemon to query the back-end database that stores usernames and aliases. The daemon continually retried all lookups because those looks were failing, eventually overloading the back-end database, and thus bringing down all services that used the database. The email servers became overloaded because they could not perform the necessary username/alias lookups. The problem was finally fixed by rolling back the software upgrade and rebooting the database and front-end nodes, thus relieving the database overload problem and preventing it from recurring.

Online testing could have caught this problem, but pre-deployment component testing did not, because the failure scenario was dependent on the interaction between the new software module and the unchanged back-end database. Throttling back username/alias lookups when they started failing repeatedly during a short period of time would also have mitigated this failure. Such a use of isolation would have prevented the database from becoming overloaded and hence unusable for providing services other than username/alias lookups.

A third interesting case study is an operator error affecting front-end machines. In this situation, users noticed that their news postings were sometimes not showing up on the service's newsgroups. News postings to local moderated newsgroups are received from users by the front-end news daemon, converted to email, and then sent to a special email server. Delivery of the email on that server triggers execution of a script that verifies the validity of the user posting the message. If the sender is not a valid *Online* user, or the verification otherwise fails, the server silently drops the message. A service operator at some point had configured that email server not to run the daemon that looks up usernames and aliases, so the server was silently dropping all news-postings-converted-into-email-messages that it was receiving. The operator accidentally configured that email server not to run the lookup daemon because he or she did not realize that proper operation of that mail server depended on its running that daemon.

The lessons to be learned here are that software should never silently drop messages or other data in response to an error condition, and perhaps more importantly that operators need to understand the high-level dependencies and interactions among the software modules that comprise a service. Online testing would have detected this problem, while better exposing failures, and improved techniques for diagnosing failures, would have decreased the time needed to detect and localize this problem. Online regression testing should take place not only after changes to software components, but also after changes to system configuration.

A fourth failure we studied arose from a problem at the interface between *Online* and an external service. *Online* uses an external provider for one of its services. That external provider made a configuration change to its service to restrict the IP addresses from which users could connect. In the process, they accidentally blocked clients of *Online*. This problem was difficult to diagnose because of a lack of thorough error reporting in *Online*'s software, and poor communication between *Online* and the external service during problem diagnosis and when the external service made the change. Online testing of the security change would have detected this problem.

Problems at the interface between providers is likely to become increasingly common as composed network services become more common. Indeed, techniques that could have prevented several failures described in this section--orthogonal redundancy, isolation, and understanding the high-level dependencies among software modules--are likely to become more difficult, and yet essential to reliability, in a world of planetary-scale ecologies of networked services.

As we have mentioned, we did not collect statistics on problem reports pertaining to systems whose failure could not directly affect the end-user experience. In particular, we did not consider problem reports pertaining to hardware and software used to support system administration and operational activities. But one incident merits special mention as it provides an excellent example of multiple related, but non-cascading, component failures contributing to a single failure. Ironically, this problem led to the destruction of *Online*'s entire problem tracking database while we were conducting our research.

Content's problem tracking database was stored in a commercial database. The data was supposed to be backed up regularly to tape. Additionally, the data was remotely mirrored each night to a second machine, just as the service data itself was remotely mirrored each night to a backup datacenter. Unfortunately, the database backup program had not been running for a year and a half because of a configuration problem related to how the database host connects to the host with the tape drive. This was considered a low-priority issue because the remote mirroring still ensured the existence of one backup copy of the data. One night, the disk holding the primary copy of the problem tracking database failed, leaving the backup copy as the only copy of the database. In a most unfortunate coincidence, an operator re-imaged the host holding the backup (and now only) copy of the database later that evening, before it was realized that the primary copy of the problem tracking database had been destroyed, and that the re-imaging would therefore destroy the last remaining copy.

We can learn several lessons from this failure. First, lightning *does* sometimes strike twice--completely unrelated component failures can happen simultaneously, leading a system with one level of redundancy to fail. Second, categorizing failures, particularly operator error, can be tricky. For example, was reimaging the backup machine after the primary had failed really an operator error, if the operator was unaware that the primary had failed? Was intentionally leaving the tape backup broken an operator error? (We do consider both to be operator errors, but arguably understandable ones.) Third, it is vital that operators understand the current configuration and state of the system and the architectural dependencies and relationships among components. Many systems are designed to mask failures--but this can prevent operators from knowing when a system's margin of safety has been reduced. The correct operator behavior in the previous problem was to replicate the backup copy of the database before reimaging the machine holding it. But in order to know to do this, the operator needed to know that the primary copy of the problem tracking database had been destroyed, and that the machine he or she was about to reimage held the backup copy of that database. Understanding *how* a system will be affected by a change is particularly important before embarking on destructive operations that are impossible to undo, such as reimaging a machine.

6. Discussion

In this section we describe three areas currently receiving little research attention that we believe could help substantially to improve the availability of Internet services: better operator tools and systems; creation of an industry-wide failure repository; and adoption of standardized service-level benchmarks. We also comment on the representativeness of the data we have presented.

6.1. Operators as first-class users

Despite the huge contribution of operator error to service failure, operator error is almost completely overlooked in designing high-dependability systems and the tools used to monitor and control them. This oversight is particularly problematic because as our data shows, operator error is the most difficult component failure to mask through traditional techniques. Industry has paid a great deal of attention to the end-user experience, but has neglected tools and systems used by operators for configuration, monitoring, diagnosis, and repair.

As previously mentioned, the majority of operator errors leading to service failure were misconfigurations. Several techniques could improve this situation. One is improved operator interfaces. This does not mean a simple GUI wrapper around existing per-component command-line configuration mechanisms--we need fundamental advances in tools to help operators understand existing system configuration and component dependencies, and how their changes to one component's configuration will affect the service as a whole. Tools to help visualize existing system configuration and dependencies would have averted some operator errors (configuration-related and otherwise) by ensuring that an operator's mental model of the existing system configuration matched the true configuration.

Another approach is to build tools that do for configuration files what *lint* [8] does for C programs: to check configuration files against known constraints. Such tools can be built incrementally, with support for additional types of configuration files and constraints added over time. This idea can be extended in two ways. First, support can be added for user-defined constraints, taking the form of a high-level specification of desired system configuration and behavior, much as [3] can be viewed as a user-extensible version of *lint*. Second, a high-level specification can be used to automatically generate per-component configuration files. A high-level specification of operator intent is a form of semantic redundancy, a technique that is useful for catching errors in other contexts (types in programming languages, data structure invariants, and so on). Unfortunately there are no widely used generic tools to allow an operator to specify in a high-level way the desired service architecture and behavior, such that the specification could be checked against the existing configuration,

or per-component configurations could be generated. Thus the very wide configuration interface remains error-prone.

An overarching difficulty related to diagnosing and repairing problems relates to collaborative problem solving. Internet services require coordinated activity by multiple administrative entities, and multiple individuals within each of those organizations, for diagnosing and solving some problems. These entities include the operations staff of the service, the service's software developers, the operators of the collocation facilities that the service uses, the network providers between the service and its collocation facilities, and sometimes the customers. Today, this coordination is handled almost entirely manually, via telephone calls to contacts at the various points. The process could be greatly improved by sharing a unified problem tracking/bug database among all of these entities and deploying collaboration tools for cooperative work.

Besides allowing for collaboration, one of the most useful properties of a problem tracking database is that it gives operators a *history* of all the actions that were performed on the system and an indication of why the actions were performed. Unfortunately this history is human-generated in the form of operator annotations to a problem report as they walk through the steps of diagnosing and repairing a problem. A tool that, in a structured way, expresses the history of a system--including configuration and system state before and after each change, who or what made the change, why they made the change, and exactly what changes they made--would help operators understand how a problem evolved, thereby aiding diagnosis and repair.

Tools for determining the root cause of problems across administrative domains, *e.g., traceroute,* are rudimentary, and these tools generally cannot distinguish between certain types of problems, such as end-host failures and network problems on the network segment where a node is located. Moreover, tools for fixing a problem once its source is located are controlled by the administrative entity that owns the broken hardware or software, not by the site that determines that the problem exists. These difficulties lead to increased diagnosis and repair times. The need for tools and techniques for problem diagnosis and repair that work effectively across administrative boundaries, and that correlate system observations from multiple network vantage points, is likely to become even greater in the age of composed network services built on top of emerging platforms such as Microsoft's .NET and Sun's SunONE.

Finally, the systems and tools operators use to administer services are not just primitive and difficult to use, they are also brittle. Although we did not collect detailed statistics about failures in systems used for service operations, we observed many reports of failures of such components. At *Content*, more than 15% of the reports in the problem tracking database related to administrative machines and services. A qualitative analysis of these problems across the services reveals that organizations do not build the operational side of their services with as much redundancy or pre-deployment quality control as they do the parts of the service used by end-users. The philosophy seems to be that operators can work around problems with administrative tools and machines if something goes wrong, while end-users are powerless in the face of service problems. Unfortunately the result of this philosophy is increased time to detect, diagnose, and repair problems due to fragile administrative systems.

6.2. A worldwide failure data repository

Although analyzing failure data seems at first straightforward, our initial expectation turned out to be far from the truth. Because the Internet services we studied recorded component failures in a database, we expected to be able to simply write a few database queries to collect the quantitative data we have presented in this paper. Unfortunately, we found that operators frequently filled out the database forms incorrectly--for example, fields such as problem starting time, problem ending time, root cause, and whether a problem was customer impacting (*i.e.,* a "service failure" as opposed to just a "component failure"), often contradicted the timestamped operator narrative of events that accompanied the problem reports. The data presented in this paper was therefore gathered by reading the operator narrative for each problem report, rather than accepting the pre-analyzed database data on blind faith. Likewise, insufficiently detailed problem reports sometimes led to difficulty in determining the actual root cause of a problem or its time to repair. Finally, the lack of historical end-user-perceived service QoS measurements prevented us from rigorously defining a "service failure" or even calculating end-user-perceived service availability during the time period corresponding to the problem reports we examined.

We believe that Internet services should follow the lead of other fields, such as aviation, in collecting and publishing detailed industry-wide failure-cause data in a standardized format. Only by knowing why real systems fail, and what impact those failures have, can researchers and practitioners know where to target their efforts. Many services, such as the ones we studied, already use databases to store their problem tracking information. It should be possible to establish a standard schema, per-

haps extensible, for Internet services, network providers, colocation facilities, and related entities, to use for recording problems, their impact, and their resolutions. We have proposed one possible failure cause and location taxonomy for such a schema in this paper. A standard schema would benefit not only researchers, but also these services themselves, as establishing and sharing such databases would help to address the coordination problem described in Section 6.1. Finally, we note that services that release such data are likely to want the publicly-available version of the database to be anonymized; automating the necessary anonymization is non-trivial, and is a research question unto itself.

6.3. Performability and recovery benchmarks

In addition to focusing dependability research on real-world problem spots, the failure data we have collected can be used to create a fault model (or, to use our terminology, component failure model) for *service-level performability 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 performability benchmarks, a small-scale replica (or a physically or virtually isolated partition) of a service is created, and QoS 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 events by either the relative frequency with which the different classes of component failure occur in the service being benchmarked, or using the proportions we found in our survey. Important metrics to measure in addition to QoS impact of injected failures, are time to detect, diagnose, and repair the component failure(s), be it automatically or by a human. As suggested in [2], the workload should include standard service administrative tasks. A recent step towards this type of benchmark is described in [16].

6.4. Representativeness

While our data was taken from just three services, we feel that it is representative of large-scale Internet services that use custom-written software to provide their service. Most of the "giant scale" services we informally surveyed use custom-written software, at least for the front-end, for scalability and performance reasons. Based on this information, we feel that our results do apply to many Internet services.

On the other hand, our data is somewhat skewed by the fact that two of our three sites (*Content* and *Read-Mostly*) are what we would call "content-intensive," meaning that the time spent transferring data from the back-end media to the front-end node is large compared to the amount of time the front-end node spends processing the request and response. Sites that are less "content intensive" are less likely to use custom-written back-end software (as was the case for *Online*). Additionally, at all three services we studied, user requests do not require transactional semantics. Sites that require transactional semantics (*e.g.,* e-commerce sites) are more likely to use database back-ends rather than custom-written back-end software. In theory both of these factors should tend to decrease the failure rate of back-end software, and indeed *Online*, the one site that used off-the-shelf back-end software, was also the site with the lowest fraction of back-end service failures.

7. Related work

Our work adds to a small body of existing studies of, and suggestions for, Internet service architectures [1] [14]. We are not aware of any studies of failure causes of such services.

A number of studies have been published on the causes of failure in various types of computer systems that are not commonly used for running Internet sites, and in operational environments unlike those of Internet services. Gray is responsible for the most widely cited studies of computer system failure data [5] [6]. In 1986 he found that operator error was the largest single cause of failure in deployed Tandem systems, accounting for 42% of failures, with software the runner-up at 25%. We found strikingly similar percentages at *Online* and *Content*. In 1989, however, Gray found that software had become the major source of outages (55%), swamping the second largest contributor, system operations (15%). We note that Gray attributed a failure to the *last* component in a failure chain rather than the root cause, making his statistics not directly comparable to ours, although this fact only matters for cascading failures, of which we found few. Though we did not quantitatively analyze the length of failure chains, Gray found longer chains than we did: 20% of failure chains were of length three or more in Gray's study, with only 20% of length one, whereas we found that almost all were of length one or two.

Kuhn examined two years of data on failures in the Public Switched Telephone Network as reported to the

FCC by telephone companies [10]. He concluded that human error was responsible for more than 50% of failure incidents, and about 30% of customer minutes, *i.e.,* number of customers impacted multiplied by number of minutes the failure lasted. Enriquez extended this study by examining a year's worth of more recent data and by examining the "blocked calls" metric collected on the outage reports [4]. She came to a similar conclusion--human error was responsible for more than 50% of outages, customer-minutes, and blocked calls.

Several studies have examined failures in networks of workstations. Thakur examined failures in a network of 69 SunOS workstations but divided problem root cause coarsely into network, non-disk machine problems, and disk-related machine problems [21]. Kalyanakrishnam studied six months of event logs from a LAN of Windows NT workstations used for mail delivery, to determine the causes of machines rebooting [9]. He found that most problems were software-related, and that average downtime was two hours. In a closely related study, Xu examined a network of Windows NT workstations used for 24x7 enterprise infrastructure services, again by studying the Windows NT event log entries related to system reboots [22]. Unlike the Thakur or Kalyanakrishnam studies, this one allowed operators to annotate the event log to indicate the reason for reboot; thus the authors were able to draw conclusions about the contribution of operator failure to system outages. They found that planned maintenance and software installation and configuration caused the largest number of outages, and that system software and planned maintenance caused the largest amount of total downtime. They were unable to classify a large percentage of the problems (58%). We note that they counted reboots after installation or patching of software as a "failure." Their software installation/configuration category therefore is not comparable to our operator failure category, despite its being named somewhat similarly.

A number of researchers have examined the causes of failures in enterprise-class server environments. Sullivan and Chillarege examined software defects in MVS, DB2, and IMS [19]. Tang and Iyer conducted a similar study for the VAX/VMS operating system in two VAXclusters [20]. Lee and Iyer categorized software faults in the Tandem GUARDIAN operating system [12]. Murphy and Gent examined causes of system crashes in VAX systems between 1985 and 1993; in 1993 they found that hardware caused about 10% of failures, software about 20%, and system management a bit over 50% [15].

Our study is similar in spirit to two studies of network failure causes: Mahajan examined the causes of BGP misconfigurations [13], and Labovitz conducted an earlier study of the general causes of failure in IP backbones [11].

8. Conclusion

From a study of more than 500 component failures and dozens of user-visible failures in three large-scale Internet services, we observe that (1) operator error is the leading cause of failure in two of the three services studied, (2) operator error is the largest contributor to time to repair in two of the three services, (3) configuration errors are the largest category of operator errors, (4) failures in custom-written front-end software are significant, and (5) more extensive online testing and more thoroughly exposing and detecting component failures would reduce failure rates in at least one service.

While 100% availability is almost certainly unattainable, our observations suggest that Internet service availability could be significantly enhanced with proactive online testing; thoroughly exposing and monitoring for component failures; sanity checking configuration files; logging operator actions; and improving operator tools for distributed, collaborative diagnosis and problem tracking. To the extent that tools for these tasks already exist, they are generally *ad hoc* and are retrofitted onto existing systems. We believe it is important that service software be built from the ground up with concern for testing, monitoring, diagnosis, and maintainability in mind--in essence, treating operators as first-class users. To accomplish this, APIs for emerging services should allow for easy online testing, fault injection, automatic propagation of errors to code modules and/or operators that can handle them, and distributed data-flow tracing to help detect, diagnose, and debug performance and reliability failures. Finally, we believe that research into system reliability would benefit greatly from an industry-wide, publicly-accessible failure database, and from service-level performability and recovery benchmarks that can objectively evaluate designs for improved system availability and maintainability.

Acknowledgements

We extend our sincere gratitude to the operations staff and management at the Internet services who provided us data and answered our questions about the data. This work was only possible because of their help. We regret that we cannot thank these services and individuals by name due to our confidentiality agreements with

them. We also thank our shepherd Hank Levy and the anonymous reviewers for their very helpful feedback on earlier versions of this paper, and the members of the Berkeley/Stanford ROC group for contributing to the ideas presented in this paper.

This work was supported in part by the Defense Advanced Research Projects Agency of the Department of Defense, contract no. DABT63-96-C-0056, the National Science Foundation, grant no. CCR-0085899, NSF infrastructure grant no. EIA-9802069, Alocity, Hewlett-Packard, Microsoft, and the California State MICRO Program. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

References

[1] E. Brewer. Lessons from giant-scale services. *IEEE Internet Computing,* vol. 5, no. 4, 2001.

[2] A. Brown and D. A. Patterson. To Err is Human. In *Proceedings of the First Workshop on Evaluating and Architecting System dependabilitY* (EASY '01), 2001.

[3] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. *4th Symposium on Operating Systems Design and Implementation (OSDI)*, 2000.

[4] P. Enriquez, A. Brown, and D. A. Patterson. Lessons from the PSTN for dependable computing. *Workshop on Self-Healing, Adaptive and Self-Managed Systems*, 2002.

[5] J. Gray. A census of Tandem system availability between 1985 and 1990. Tandem Computers Technical Report 90.1, 1990.

[6] J. Gray. Why do computers stop and what can be done about it? *Symposium on Reliability in Distributed Software and Database Systems,* 1986.

[7] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton. Software rejuvenation: analysis, models, and applications. *25th symposium on fault-tolerant computing,* 1995.

[8] S. C. Johnson. Lint, a C program checker. Bell Laboratories computer science technical report, 1978.

[9] M. Kalyanakrishnam, Z. Kalbarczyk, and R. Iyer. Failure data analysis of a LAN of Windows NT based computers. *18th IEEE Symposium on Reliable Distributed Systems,* 1999.

[10] D. R. Kuhn. Sources of failure in the public switched telephone network. *IEEE Computer* 30(4), 1997.

[11] C. Labovitz, A. Ahuja, and F. Jahanian. Experimental study of Internet stability and backbone failures. *Fault-Tolerant Computing Symposium (FTCS)*, 1999.

[12] I. Lee and R. Iyer. Software dependability in the Tandem GUARDIAN system. *IEEE Transactions on Software Engineering,* 21(5), 1995.

[13] R. Mahajan, D. Wetherall, and T. Anderson. Understanding BGP misconfiguration. *SIGCOMM '02*, 2002.

[14] Microsoft TechNet. Building scalable services. http://www.microsoft.com/technet/treeview/default.asp?url=/TechNet/itsolutions/ecommerce/deploy/projplan/bss1.asp, 2001.

[15] B. Murphy and T. Gent. Measuring system and software reliability using an automated data collection process. *Quality and Reliability Engineering International,* vol 11, 1995.

[16] K. Nagaraja, X. Li, R. Bianchini, R. P. Martin, and T. D. Nguyen. Using fault injection and modeling to evaluate the performability of cluster-based services. *4th USENIX Symposium on Internet Technologies and Systems (USITS '03)*, 2003.

[17] D. Oppenheimer and D. A. Patterson. Architecture, operation, and dependability of large-scale Internet services: three case studies. *IEEE Internet Computing,* September/October, 2002.

[18] D. A. Patterson., A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, N. Treuhaft. Recovery-Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies. Technical Report UCB//CSD-02-1175, 2002.

[19] M. S. Sullivan and R. Chillarege. A comparison of software defects in database management systems and operating systems. *Proceedings of the 22nd International Symposium on Fault-Tolerant Computing,* 1992.

[20] D. Tang and R. Iyer. Analysis of the VAX/VMS error logs in multicomputer environments--a case study of software dependability. *Proceedings of the Third International Symposium on Software Reliability Engineering,* 1992.

[21] A. Thakur and R. Iyer. Analyze-NOW--an environment for collection adn analysis of failures in a network of workstations. *IEEE Transactions on Reliability,* R46(4), 1996.

[22] J. Xu, Z. Kalbarczyk, and R. Iyer. Networked Windows NT system field failure data analysis. *Proceedings of the 1999 Pacific Rim International Symposium on Dependable Computing,* 1999.