Why do Internet services fail, and what can be done about it?[1]

David Oppenheimer
University of California at Berkeley
`davidopp@cs.berkeley.edu`

24 May 2002

Abstract

We describe the architecture, operational practices, and failure characteristics of three very large-scale Internet services. Our research on architecture and operational practices took the form of interviews with architects and operations staff at those (and several other) services. Our research on component and service failure took the form of examining the operations problem tracking databases from two of the services and a log of service failure post-mortem reports from the third.

Architecturally, we find convergence on a common structure: division of nodes into service front-ends and back-ends, multiple levels of redundancy and load-balancing, and use of custom-written software for both production services and administrative tools. Operationally, we find a thin line between service developers and operators, and a need to coordinate problem detection and repair across administrative domains. With respect to failures, we find that operator errors are their primary cause, operator error is the most difficult type of failure to mask, service front-ends are responsible for more problems than service back-ends but fewer minutes of unavailability, and that online testing and more thoroughly exposing and detecting component failures could reduce system failure rates for at least one service.

1. 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, 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.

Section 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.

The architects and operators of these services might see such problems as failures on their part. But 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. To answer the question "Why do Internet services fail and what can be done about it?", we have studied both the architecture and failure reports from three very large-scale Internet services. In this report we describe the lessons we have learned from this study. We observe that services

- are generally housed in geographically distributed colocation facilities and use mostly commodity hardware and networks;
- use multiple levels of redundancy and load balancing for performance and availability;
- are built from a load-balancing tier, a stateless front-end tier, and a back-end tier that stores persistent data;
- use primarily custom-written software to provide and administer the service;
- undergo frequent software upgrades and configuration updates;
- operate their own 24x7 Systems Operations Centers staffed by operators who monitor the service and respond to problems.

Despite redundancy throughout the services and trained operations staff, the services still experience end-user visible service failures. In studying these failures, we discover that

- operator errors are their primary cause;
- operator error is the most difficult type of failure to mask;
- contrary to conventional wisdom, service front-ends are responsible for more problems than service back-ends; and
- online testing and more thoroughly detecting and exposing component failures could reduce system failure rates for at least one service.

Although software architecture, operational practices, and failures of various kinds have been studied in the context of end-user and enterprise software, researchers have thus far paid much less attention to these attributes when it comes to large-scale Internet services. Before presenting the results of our study, we briefly describe the important differences between the Internet service environment and more traditional platforms.

1.1 The emergence of a new application environment

The dominant software platform is evolving from one of shrink-wrapped applications installed on end-user personal computers to one of Internet-based application services deployed in large-scale, globally distributed clusters. Although they share some characteristics with traditional desktop applications and traditional high-availability software, Internet services are built from a unique combination of building blocks, they have different requirements, and they are deployed in a different environment. Table 1 summarizes these differences.

Continuous 24x7 availability is demanded by a globally distributed user base that increasingly sees these services as essential parts of the world's communications infrastructure. As suggested in Table 1, however, large-scale Internet services present an availability challenge because they

Characteristic	**Traditional desktop applications**	**Traditional high-availability applications**	**Large-scale Internet service applications**
Dominant application	productivity applications, games	database, enterprise messaging	email, search, news, e-commerce, data storage
Typical hardware platform	desktop PC	fault-tolerant server or failover cluster	cluster of 100s-1000s of cheap PCs, often geographically distributed
Software model	off-the-shelf, standalone	off-the-shelf, multi-tier	custom-written, multi-tier
Frequency of new software releases	months	months	days or weeks
Networking environment	none (standalone)	enterprise-scale	within cluster, Internet among data centers, Internet to/from user clients
Who operates the software	end-user	corporate IT staff	service operations staff, datacenter/colocation site staff
Typical physical environment	home or office	corporate machine room	colocation facility
Key metrics	functionality, interactive latency	availability, throughput	availability, functionality, scalability, manageability, throughput

Table 1: Internet services vs. traditional applications.

- are typically built from large numbers of inexpensive PCs that lack expensive reliability features;
- undergo frequent scaling, reconfiguration, and functionality changes;
- often run custom-written software with limited testing;
- rely on networks within service clusters, among geographically distributed collocation facilities, and between collocation facilities and end-user clients; and
- are expected to be available 24x7 for access by users around the globe, thus making planned downtime undesirable or impossible.

On the other hand, some features of large-scale services can be exploited to enhance availability:

- plentiful hardware allows for redundancy;
- use of collocation facilities allows controlled environmental conditions and resilience to large-scale disasters via geographic distribution;
- operators can learn the software's inner workings from developers and thus more quickly identify and correct problems, as compared to IT staffs running shrink-wrapped software.

We will examine a number of techniques have been developed to exploit these characteristics.

A challenge closely related to availability is maintainability. Not only are human operators essential to keeping a service running, but the same issues of scale, complexity, and rate of change that makes these services prone to failure also makes them challenging for human operators to manage. Yet existing tools and techniques for configuring systems, visualizing system configurations and the changes made to them, partitioning and migrating persistent data, and detecting, diagnosing and fixing problems, are largely *ad hoc* and often inadequate.

As a first step toward formalizing the principles for building highly available and maintainable large-scale Internet services, we have conducted a survey of the architecture, operational practices, and dependability of existing services. This report describes our observations and is organized as follows. Section 2 describes in detail the architectures of three representative large-scale Internet services. In Section 3 we describe some operational issues involved in maintaining Internet services, and Section 4 summarizes our analysis of failure data from three of these services. In Section 5 we discuss related work, and in Section 6 we conclude and discuss possible future directions.

Section 2

Architecture

To demonstrate the general principles behind the construction of large-scale Internet services, we examined the architectures of a number of representative services. Table 2 highlights some key characteristics of the three services we studied in detail: an online service/internet portal ("*Online*"), a global content hosting service ("*Content*"), and a high-traffic, read-mostly Internet service ("*ReadMostly*"). In keeping with these services' desires to remain anonymous, we have intentionally abstracted some of the information to make it more difficult to identify the individual services.

Architecturally, these services

- are housed in geographically distributed collocation facilities,
- are built from largely commodity hardware but custom-written software,
- use multiple levels of redundancy and load balancing to achieve improved performance and availability, and
- contain a load-balancing tier, a stateless front-end tier, and a stateful back-end tier.

The primary differences among these services are load, read/write ratio, and whether clients use special-purpose software or a standard web browser.

2.1 Geographic server distribution

At the highest level, many services distribute their servers geographically in collocation facilities. For example, *Online* distributes its servers between its headquarters and a nearby collocation facility, *ReadMostly* uses a pair of facilities on the east coast of the United States and another pair on the west coast, and *Content* uses one facility in each of the United States east coast, United States west coast, Asia, and Europe. All three services used this

Characteristic	Online	Content	ReadMostly
Hits per day	~100 million	~7 million	~100 million
# of machines	~500, 2 sites	~500, ~15 sites	> 2000, 4 sites
Hardware	SPARC + x86	x86	x86
Operating system	Solaris	open-source x86 OS	open-source x86 OS
Relative read/write ratio	medium	medium	very high (and very little data is updated by users)

Table 2: Differentiating characteristics of the services described in this study.

geographic distribution for availability, and all but *Content* also used the redundant datacenters to improve performance by sharing in the handling of user requests.

Several mechanisms are available for directing user queries to the most appropriate site when distributed datacenters are used for load sharing. The choice of site generally takes into account the load on each site and its availability. In the case of *Content*, this functionality is pushed to the client, which is pointed to one primary site and one backup site. The two sites work in redundant pairs to reduce administrative complexity--some clients are pointed to one pair of sites, while others are pointed to the other pair. Updates are propagated from a client's primary server site to its secondary server site on a nightly basis. In the case of *Online*, a server at the company's headquarters is responsible for providing clients with an up-to-date list of the best servers to contact whenever the client connects; these servers may be at the company's headquarters or at its collocation facility. *ReadMostly* uses its switch vendor's proprietary global load balancing mechanism, which rewrites DNS responses based on load and health information about other sites collected from the cooperating switches at those sites.

2.2 Single site architecture

Drilling down into the architecture of a single site, we see three tiers: load balancing, front-end servers, and back-end servers. Figures 1, 2, and 3 depict the single-site architecture of *Online*, *Content*, and *ReadMostly*, respectively.

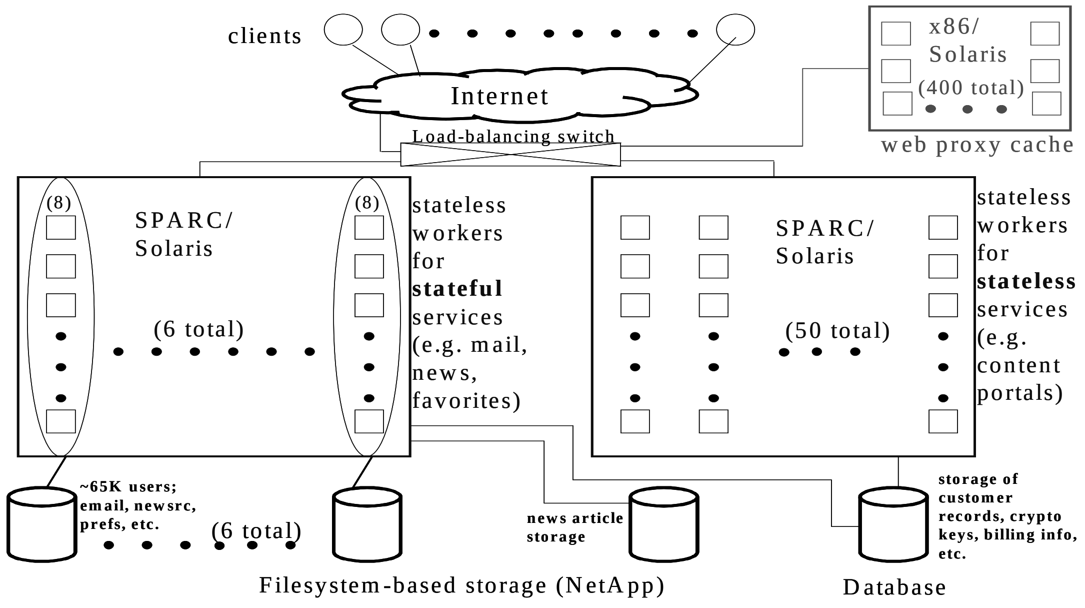
clients
Internet
Load-balancing switch
x86/ Solaris
(400 total)
web proxy cache
(8)
(8)
SPARC/ Solaris
(6 total)
stateless workers for **stateful** services (e.g. mail, news, favorites)
SPARC/ Solaris
(50 total)
stateless workers for **stateless** services (e.g. content portals)
~65K users; email, newsrc, prefs, etc.
(6 total)
news article storage
storage of customer records, crypto keys, billing info, etc.
Filesystem-based storage (NetApp)
Database

Figure 1: 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 servers, 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 cluster nodes via NFS over UDP. This cluster is connected to a second site, at a collocation facility, via a leased network connection.

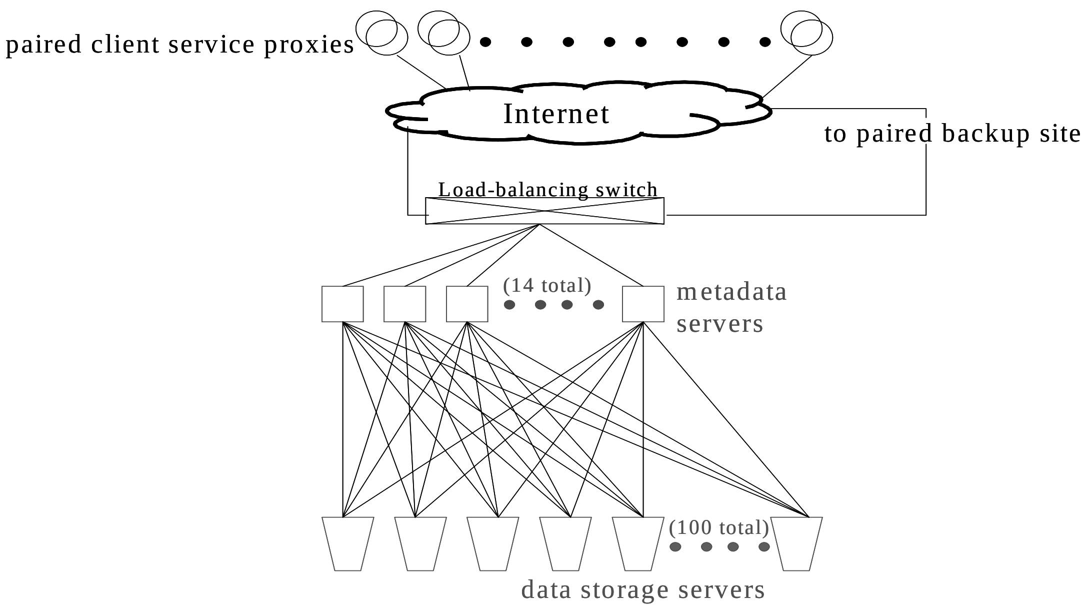

Figure 2: The architecture of one site of *Content*. Stateless metadata servers provide file metadata and route requests to the appropriate data storage servers. Persistent state is stored on commodity PC-based storage servers and is accessed via a custom protocol over UDP. Each cluster is connected to its twin backup site via the Internet.

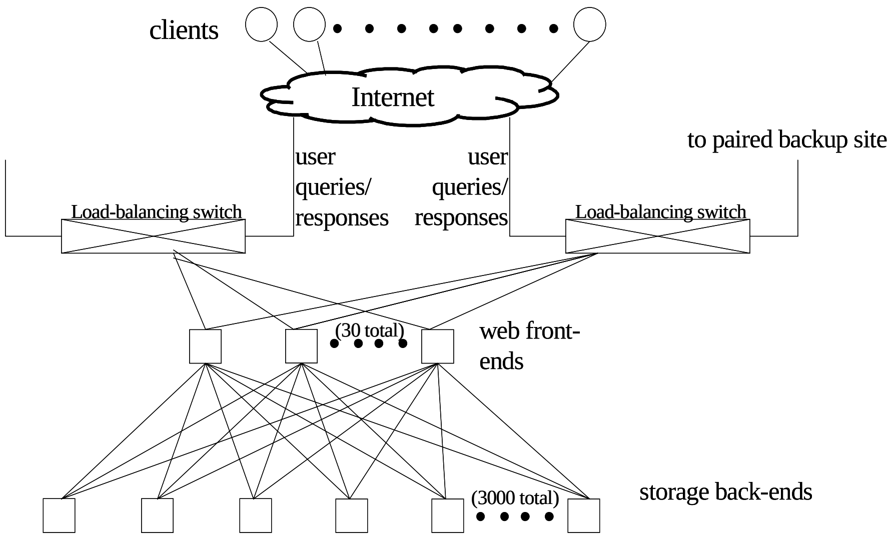

Figure 3: The architecture of one site of *ReadMostly*. A small number of web front-ends direct requests to the appropriate back-end storage servers. Persistent state is stored on commodity PC-based storage servers and is accessed via a custom protocol over TCP. A redundant pair of network switches connects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

2.2.1 Tier 1: Load balancing

A front-end network balanced load by distributing incoming requests to front-end servers based on the load on those servers. Though many modern switches offer “layer 7" switching functionality, meaning they can route requests based on the contents of a user's request, none of the sites we surveyed actually use this switch feature. Instead, they generally use simple DNS round robin or level 3 (IP-level) load distribution to direct clients to the least loaded front-end server. In DNS round robin, multiple IP addresses are advertised for a service and are continuously reprioritized to spread load among the machines corresponding to those addresses. In level 3 load distribution, clients connect to a single IP address and the cluster's switch routes the request to any one of the front-end servers.

Content and *ReadMostly* use IP-level request distribution at each site, as does *Online* for "stateless" parts of its service like the web proxy cache and content portals. *Online* uses IP-level request distribution for the "stateful" parts of its service (*e.g.,* email), but layers a level of stateful front-end load balancing on top of it. In particular, *Online* uses a user's identity (determined when the user "logs in" to the service) to map the user to one of several clusters (called service groups). Each cluster is then further load balanced internally using IP-level load balancing.

We found the theme of multiple levels of load balancing to be a general principle in building sites - load is often balanced among geographically distributed sites, it is always balanced among front-ends in a cluster, and one of multiple clusters at a site may be chosen using a third load balancing decision.

2.2.2 Tier 2: Front-end

Once the load balancing decision is made at the cluster level for a request, the request is sent to a front-end machine. The front-end servers run stateless code that answers incoming user requests, contacts one or more back-end servers to obtain the data necessary to satisfy the request, and in some cases returns that data to the user (after processing it and rendering it into a format suitable for presentation). We say "in some cases" because some sites, such as *Content*, return data to clients directly from the back-end server. The exact ratio of back-end to front-end servers is highly dependent on the type of service and the performance characteristics of the back-end and front-end machines; this ratio ranged from 1:10 for *Online* to 50:1 for *ReadMostly*.

Depending on the needs of the service, the front-end machines may be partitioned with respect to functionality and/or data. *Content* and *ReadMostly* are partitioned with respect to neither characteristic; all front-end machines access data from all back-end machines. *Online*, however, partitions front-end machines with respect to both attributes. With respect to functionality, there are three different types of front-end machines (front-ends for stateful services, front-ends for stateless services, and web proxy caches, which are essentially front-ends to the Internet). With respect to data, the front-ends to the stateful services are further partitioned into service groups, which were described earlier. In all cases, the front-end machines were inexpensive PCs or workstations; the internal architectures for the three sites we spotlight are depicted in Figures 4, 5, and 6. These three services use custom-written front-end software rather than off-the-shelf web servers like Apache or IIS. The services cite functionality and/or performance as reasons for not using commodity software.

2.2.3 Tier 3: Back-end storage

Data needed to satisfy a request is obtained from one or more back-end servers. These servers store persistent data, such as files in the case of *Content* and *ReadMostly*, and email, news articles, and user preferences in the case of *Online*. It is in the back-end that we see

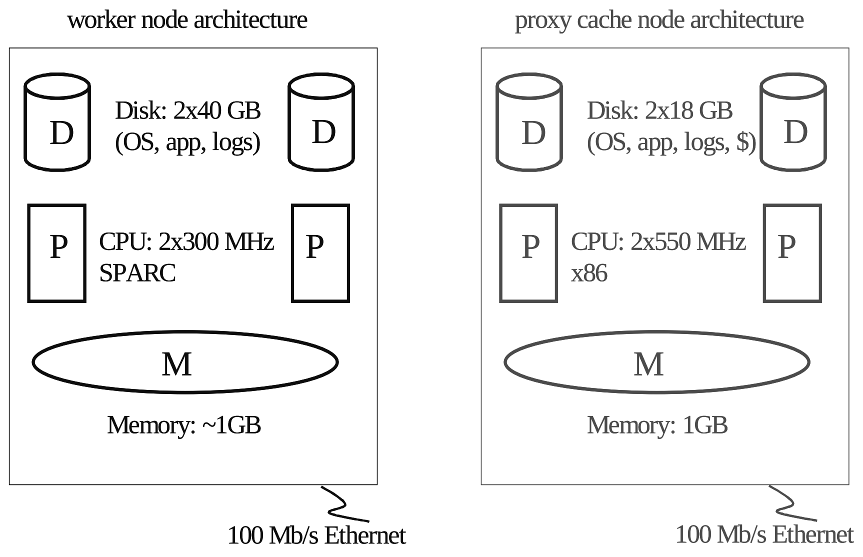

Figure 4: Architecture of worker nodes and proxy cache nodes in *Online*. Nodes contain two disks, two CPUs (SPARC in the case of worker nodes, x86 in the case of proxy cache nodes), about 1 GB of memory, and a 100 Mb/s Ethernet interface.

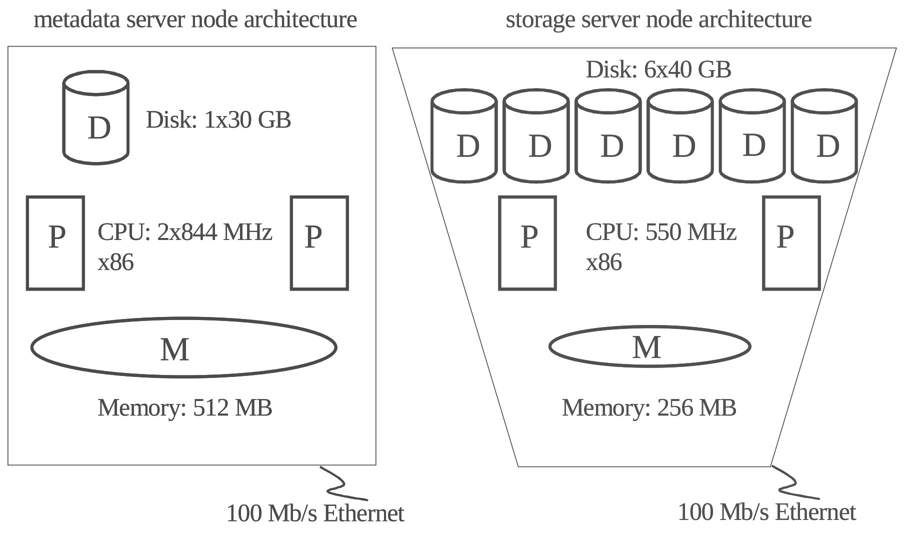

Figure 5: Architecture of metadata server nodes and storage server nodes in *Content*. Metadata servers contain one disk, two CPUs, a 100 Mb/s Ethernet interface, and 512 MB of memory. Storage servers contain six disks, two CPUs, 256 MB of memory, and a 100 Mb/s Ethernet interface.

the greatest variability in node architecture--two sites use inexpensive PCs (*Content* and *ReadMostly*), while one uses a file server designed especially for high availability (*Online*). Likewise, there are a variety of filesystems used by these back-end servers, including custom filesystems (*Content*) and Unix-based filesystems (*Online* and *ReadMostly*).

In the back end, partitioning is key to overall service scalability, availability, and maintainability. Scalability takes the form of scale-out in which more back-end servers are added to absorb growing demands for storage and aggregate throughput. The services we

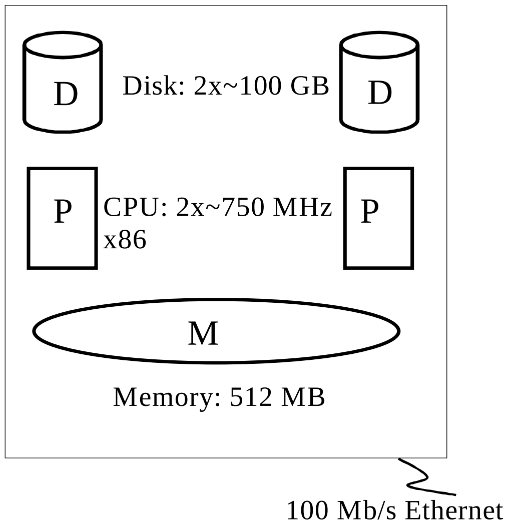

Figure 6: Architecture of service nodes in *ReadMostly*. Nodes contain two disks, two CPUs, 512 MB of memory, and a 100 Mb/s Ethernet interface.

highlight a represent spectrum of redundancy schemes used to improve availability or both performance and availability.

Online partitions users into "service groups" of approximately 65,000 users each, and assigns each group to a single back-end Network Appliance filer. RAID-4 is used within each filer, thereby allowing each service group to tolerate a single back-end disk failure. The drawbacks of this scheme are that the failure of a single filer will cause all users in that service group to lose access to their data, and that the scheme does not improve performance, since only one copy of each piece of data is stored.

Content partitions data among back-end storage nodes. Instead of using RAID within each filer, it replicates each piece of data on a storage server at a twin data center. This scheme offers higher theoretical availability than that used by *Online*, because single disk failure, single storage server failure, and single site failure (due to networking problems, environmental issues, and the like) are all masked. As in *Online*, however, this back-end data redundancy scheme does not improve performance, because only one site is used as the default for retrieving data to service user requests, with the backup twin site used only in case of failure of the first storage server or its site.

Finally, *ReadMostly* uses full replication of each piece of data to achieve high availability and improved performance. Data is replicated among nodes in the cluster and among geographically distributed clusters. Incoming requests from the front-end are load balanced among redundant copies of data stored in the back end. Both back-end availability and performance are improved by a factor proportional to the number of copies of each piece of data stored. It is reasonable that we find this redundancy scheme used by the least write-intensive of the services we studied - the main drawback to replicating data among nodes in a cluster (as opposed to RAID storage attached to a single machine) is that updates are slow since they must be propagated to all nodes storing a particular piece of data. Also, we note that end-users rarely update *ReadMostly*'s dataset, and that there is thus no strong need for users to be able to see their own writes. This allows for data updates to be propagated lazily to all the nodes storing replicas, mitigating the impact of update propagation delays.

2.2.4 Networking

Finally, we note that while the structure of the wide-area networking among server sites varied greatly among the services we surveyed, networking within a single site followed a common pattern. In particular, a collocation facility's network is generally connected to a service's first level switch, which is then connected by gigabit Ethernet to one or more smaller second-level switches per machine room rack. These second-level switches are in turn connected by 100 Mb/s Ethernet to individual nodes. Although it was common for sites to use redundant connections between the first and second level switches, no sites used redundant Ethernet connections between nodes and second-level switches, and only one (*ReadMostly*) used redundant connections between the collocation facility's network and the first-level of switches.

We observed that despite the emergence during the past few years of industry-standard, low-latency, high-bandwidth, system-area networking technologies such as VIA and Infiniband, all of the services use UDP over IP or TCP over IP within their clusters. This is most likely due to cost-a 100BaseT NIC card costs less than $50 (and is often integrated into modern motherboards), while the cost of a PCI card for VIA is on the order of $1000.

2.3 Common principles

To review, the three services we highlight in this article, along with about a half-dozen others that we surveyed is less detail, share a number of characteristics:

- They use large numbers of nodes, distributed geographically among clusters in collocation facilities;
- they use multiple levels of load balancing to derive high throughput and high availability from this large number of nodes;
- they run on commodity PC-based front-end hardware and commodity networks;
- their production service and operations staff use custom-built software;
- they undergo frequent software updates and configuration changes; and
- they maintain a close relationship between operators and developers.

However, we observed differences in a few areas:

- Services sometimes use commodity PCs to store back-end data (as with *ReadMostly* and *Content*), but many use high-end Unix systems or specialized high-availability servers (as with *Online*);
- services sometimes write custom software for back-end data storage (as with *Content*), but most use databases or standard filesystems (as with *Online* and *ReadMostly*); and
- not all services have embraced geographic distribution and collocation facilities. Indeed, two services that are not described in this article and that receive ~100 million hits per day use a single site at their own headquarters. They argue that manageability issues outweigh disaster tolerance and possibly improved performance to some user locations.

Section 3

Service operational issues

The need to develop, deploy, and upgrade services on “Internet time,” combined with the size and complexity of service infrastructures, has placed unique pressures on the traditional software lifecycle. This pressure can be seen particularly in testing, deployment, and operations. The amount of time available for testing has shrunk, the frequency and scale of deployment and upgrade has expanded, and the importance of operational issues such as monitoring, problem diagnosis and repair, configuration and reconfiguration, and general system usability for human operators has grown. The services we studied are addressing these challenges in a variety of ways.

3.1 Testing

All three of the services highlighted in this report test their software before releasing it to their production cluster. They all have development quality assurance (QA) teams, of varying sizes, that test the software before deployment. These groups use traditional testing techniques such as unit tests and regression tests, using both single nodes and using small-scale test clusters that reflect the architecture--though invariably not all of the minute configuration details--of the production cluster.

Although the operations team is involved in deployment to the production cluster in the case of all three services, *Online* involves their operations team in the testing process more than do the others and is therefore worth highlighting. That service uses a three-step testing and deployment process for new software or software features after the unit development QA tests are passed. In the first step, the candidate software is deployed on the development group's test cluster and is run by the developers, but is configured by the operations team. In the second step, the stable version of the software from the first step is taken by the operations group, deployed, and operated on their test cluster. The final steps of software release are alpha and beta releases to the production service; for major releases, a new version of the software is released to just a subset of the users for a week or two before it is rolled out to all customers. This testing and deployment methodology is noteworthy because of the way in which it integrates the operations team from the very beginning, considering operators as first-class users of the software as much as end-users.

3.2 Deployment

Deployment of software to large clusters is a difficult process that requires automation for reasons of both efficiency and correctness. All three sites described here use tools developed in-house to deploy and upgrade production software and configurations for applications and operating systems. All three sites used rolling upgrades [2] for deploying software to their clusters. All three sites used version control during software development, and *Online* and *ReadMostly* use it for configuration management. *Online* always keep two versions of the service installed on every machine, thereby allowing them to revert to an older version in less than five minutes if a new installation goes awry.

The three services differed with respect to the frequency with which they update application software and operating system and application configurations. Back-end software changed the least frequently, front-end software changed more frequently, and machine configurations changed the most often. Of course, content changes on a daily basis in all three services.

3.3 Daily operation

The tasks of service monitoring, problem diagnosis, and repair are significantly more challenging for large-scale Internet services than for traditional desktop applications or enterprise applications. Among the reasons for this are the frequency of software and configuration changes, the scale of the services, and the unacceptability of taking the service down for problem localization and repair. There are also complex interactions among

- application software and configuration,
- operating system software and configuration,
- networks at collocation sites,
- networks connecting customer sites to the main service site,
- networks connecting collocation sites, and
- operators at collocation sites, Internet service providers, and the service's operations center.

Back-end data partitioning is a key maintainability challenge. Data must be distributed among back-end storage nodes so as to balance the load on them and so that no server's storage capacity is exceeded. As the number of back-end machines grows, back-end data must be repartitioned in order to incorporate the new machines into the service. In the current state of the art, partitioning and repartitioning decisions are made by humans, and the repartitioning process itself is automated by tools that are, at best, *ad hoc*.

The scale, complexity, and speed of evolution of Internet services makes operator tasks such as installing, upgrading, diagnosing, and fixing system components frequent events. These tasks require a deeper understanding of the internals of the application software than is required to install and operate slowly evolving, thoroughly documented enterprise or desktop software. There is frequent dialog between operators and developers during upgrades, problem diagnosis, and repair. For example, distinguishing between an application crash caused by a software bug--generally the responsibility of software developers--and an application crash caused by a configuration error--generally the responsibility of the operations team--can require discussion between both groups. As a result, these services are organizationally structured so that the line between “operations” and “software development” is blurry.

Despite automation of some tasks such as low-level monitoring and deployment of software to a cluster, human operators are key to service evolution and availability. Moreover, there is a growing realization that operators are as much a user of Internet service software as are end-users, and that usability by operators is a concern that should be integrated throughout the software development process. Many operational tasks are not fully automated or are not automated at all, such as scaling the service; configuring and reconfiguring software and network equipment; partitioning and repartitioning users and data; identifying the root cause of problems; tuning performance; and responding to hardware, software, security, and networking issues. Conversely, there seems to be much more opportunity for more sophisticated and standardized problem diagnosis and configuration management tools, as all of the sites we surveyed used some combination of their own custom-written tools and manual operator intervention for these tasks.

Unfortunately, building generic tools for these tasks that can be customized for a particular site is difficult. Even a seemingly simple activity such as finding the root cause of an outage can be difficult. For example, in a large network, unreachability of twenty servers might be not be unusual. But if those twenty servers are exactly the set of nodes behind a single switch, then most likely the cause of the problem is not the nodes themselves but rather the switch them connects them to the rest of the network. Ideally the service operator would be told of the switch outage and the alarms for the twenty servers would be suppressed, as they just lead to confusion about the root cause of the problem. But performing even this simple filtering requires an understanding of the topology of the network, which must be part of the monitoring system's configuration. There is a tradeoff between hiring sophisticated programmers to build and configure automated monitoring and maintenance tools, and simply hiring a larger number of less skilled operators to apply their human intelligence to system operation tasks using less sophisticated tools. As the scale of deployed systems and the number of such systems grows, the advantage may shift from the human-centric model to an automation-centric one. But the day of fully automatic system operation seems a long way off, and the prevalent operational model today is clearly that of many operators using relatively unsophisticated tools.

3.4 Common principles

All of the large-scale Internet services we studied are in a state of constant flux with respect to hardware and software configuration. They are also under pressure to rapidly test and deploy new software. In response, the services operate their own 24x7 Systems Operations Centers staffed by operators who monitor the service and respond to problems. These operators also work closely with developers when diagnosing, and sometimes when responding to, problems.

Our architectural study reveals pervasive use of hardware and software redundancy at the geographic, node, and component levels. Nonetheless, outages at large-scale Internet services are frequent. Motivated by this observation, we set out to study the causes of failures in such services.

Section 4

Dependability

This section describes the results of our study of the dependability of the three services outlined in Section 2. We examine the root causes of component and user-visible service failures, and the mean time to repair service failures. We also speculate on the potential effectiveness of a number of techniques for mitigating failures, had they been applied to the failure scenarios experienced by one of the services for which we have extensive data (*Online*).

Our primary conclusions in this section are that

- operator error is the leading contributor to user-visible failures;
- operator error is the most difficult type of component failure to mask;
- contrary to conventional wisdom, front-end software can be a significant cause of user-visible failure, primarily due to configuration errors;
- back-end failures, while infrequent, take longer to repair than do front-end failures;
- more online testing, increased redundancy, and more thoroughly exposing and reacting to software and hardware failures would have reduced service failures in one service studied (*Online*); and
- automatic sanity checking of configuration files, and online fault and load injection, would also offer some potential benefit to that service.

4.1 Survey methodology and terminology

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 *service failure* as one that could (1) prevent an end-user from accessing the service or a part of the service, even if the user is given a reasonable error message; or (2) significantly degrade a user-visible aspect of system performance. We chose this definition of failure because it most closely matched the criteria that *ReadMostly* used to select problems that merited a post-mortem. Note that "user-visible" is implicit in our definition of service failure.

"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 externally observed system Quality of Service such as response time. But even in those cases 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.,* service problems observed by operators and monitoring tools within the service infrastructure, that would have been visible externally if a user had tried to access the service during the failure.

We classified 85 reports of component failures from *Online* and 99 component failures from *Content*. These component failures turned into 18 service failures in *Online* and 20 service failures in *Content*. *ReadMostly* supplied us with 21 service failures (and two additional failures that we considered to be below the threshold to be deemed a service failure). These problems corresponded to four months at *Online*, one month at *Content*, and six months at *ReadMostly*. Table 3 summarizes these characteristics for the data set we examined. In classifying problems, we consider operators to be just another type of component of the system; when they fail, their failure may or may not result in a service failure.

We note that it is not fair to compare these three services directly, as the functionality of the custom-written software at *Online* is richer than that at *ReadMostly*, and *Content* is more complicated than either of the other two services. For example, *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 over a given time period, and we used approximately the same definition of failure for choosing 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 service failure to the first component that failed in the chain of events leading up to the service failure. The *type* of the cause of the failure--its root cause--was categorized as node hardware, network hardware, node software, network software (*i.e.,* router or switch firmware), environment, operator error, overload/disk full, or unknown. The *location* of that component was categorized as front-end node, back-end node, or network. Note that the underlying flaw may have remained latent for a long 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, although they may cache or temporarily queue data. Back-end nodes do store

Service	**# of component failures**	**# of resulting service failures**	**Period covered in problem reports**
Online	85	18	4 months
Content	99	20	1 month
ReadMostly	N/A	21	6 months

Table 3: Characteristics of the failure data problem set examined.

persistent data. The “business logic” of traditional three-tier systems terminology is therefore part of our definition of front-end, a reasonable decision because these services integrate their service logic with the code that receives and replies to client requests.

Most problems were relatively easy to map into this two-dimensional *location-type* space, except for wide-area network problems. Network 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 the location as network and the cause as unknown.

4.2 Analysis of component and service failure causes

We analyze our data on component and service failure with respect to three properties: how many component failures turned into service failures, categorized by location and type (Section 4.2.1); the relative frequency of each component and service failure root cause, categorized by location and type (Section 4.2.2); and the Mean Time To Repair (MTTR) of each root cause location and type (Section 4.2.2)

4.2.1 Component failures resulting in service failures

As described in Section 2, the services we studied use redundancy in an attempt to mask component failures. That is, they try to prevent component failures from turning into service failures. As indicated by Figure 7 and Figure 8, these techniques generally do a good job of preventing hardware, software, and network component failures from turning into service failures, but they are much less effective at preventing operator failures from becoming service 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. Node hardware and software failures are masked relatively well. Network failures generally are not masked as effectively because of the smaller degree of network redundancy as compared to node redundancy. This masking issue is discussed in more detail in the next section.

4.2.2 Failure root causes and time to repair

Next we examine the source and magnitude of service failures, categorized by the root cause location and type of component failure. Table 4 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 error in configuration and administration of front-end software (both application and operating system). Almost all of the problems in *ReadMostly* were network-related; this was likely due to simpler and better-tested application software at that service, less operator “futzing” with the service on a day-to-day basis, and a higher degree of hardware redundancy than is used at *Online* and *Content*.

Table 5 shows that operator error is the largest single cause of service failures in *Online* and *Content*. Operator error in all three services tended to take the form of configuration errors (errors in setting up configuration files) rather than procedural errors made during

	Front-end	Back-end	Network	Unknown
Online	72%	0%	28%	0%
Content	55%	20%	20%	5%
ReadMostly	0%	10%	81%	9%

Table 4: Failure cause by location. Contrary to conventional wisdom, most failure root causes were components in the service front-end.

an interactive session (*e.g.,* moving files or users). Indeed, 100%, 89%, and 50% of the operator errors that led to service failures in *Online*, *Content*, and *ReadMostly*, respectively, were due to configuration errors. In general, these failures arose when operators were making changes to the system, *e.g.,* changing software configurations, 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 database, arose during the course of normal maintenance.

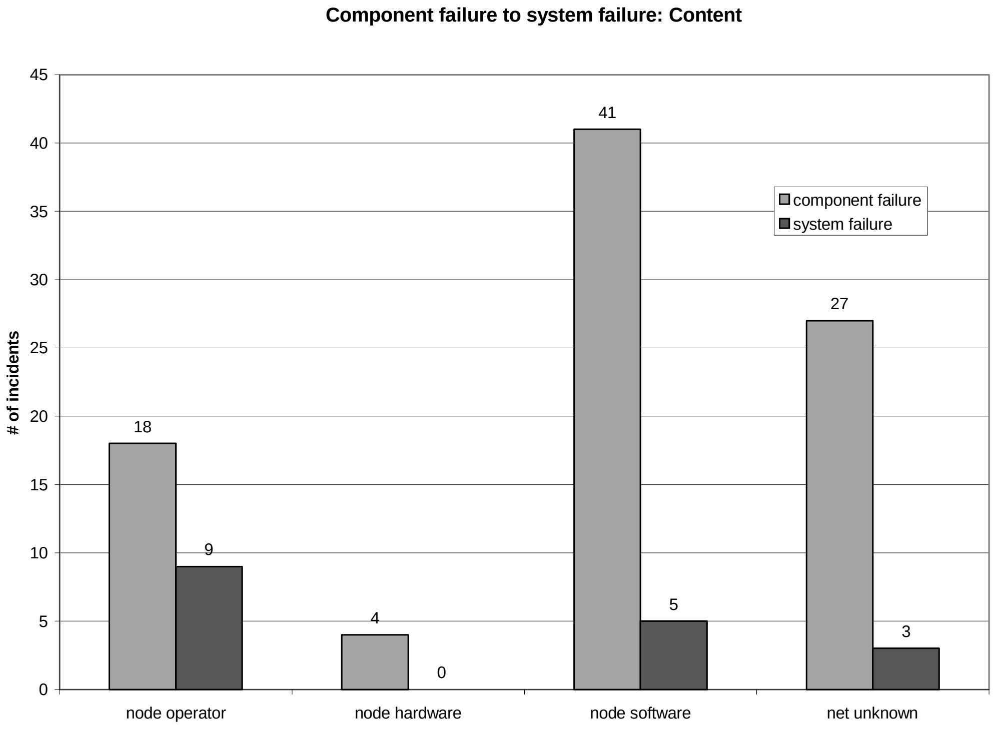

Figure 7: Number of component failures and resulting service failures for *Content*. Only those categories for which we classified at least three component failures (operator error related to node operation, node hardware failure, node software failure, and network failure of unknown cause) are listed. 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, 11% or fewer component failures became service failures. Fully half of the 18 operator errors resulted in service failure, suggesting that operator errors are significantly more difficult to mask using the service's existing redundancy mechanisms.

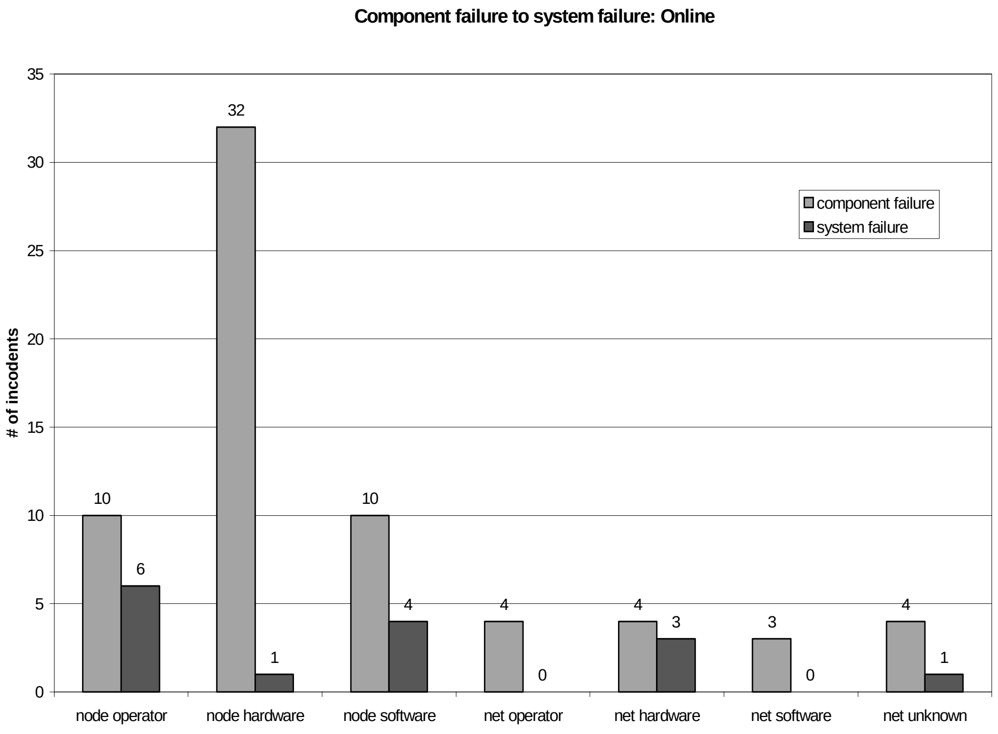

Figure 8: Number of component failures and resulting service failures for *Online*. Only those categories for which we classified at least three component failures (operator error related to node operation, node hardware failure, node software failure, and various types of network failure) are listed. As with *Content*, operator error was difficult to mask using the service's existing redundancy and recovery schemes. Unlike content, a significant percentage of node software and network hardware failures became service failures. There is no single explanation for this, as the three customer-impacting network hardware problems affected different pieces of equipment.

Networking problems were the largest cause of failure in *ReadMostly* and, as mentioned in Section 4.2.1, somewhat more difficult to mask than node hardware or software component failures. An important reason for these facts is that networks are often a single point of failure. For example, *ReadMostly* was the only site we examined that used redundant first and second-level switches within clusters. 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. Moreover, network failure modes tend to be complex: networking hardware and software can fail outright or become overloaded and start dropping packets. Combined with the inherent redundancy of the Internet, these 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 aims to achieve. Finally, because most services use a single network operations center, operators sometimes cannot see the network problems among their collocation sites or between their collocation sites and customers; these problems may thus be at the root of a mysterious performance problem or outage.

Colocation facilities did appear effective in eliminating "environmental" problems--no environmental problems, such as power, temperature, and so forth, led to service failure.

	Node op	Net op	Node hw	Net hw	Node sw	Net sw	Node unk	Net unk	Env
Online	33%	0%	6%	17%	22%	0%	0%	6%	0%
Content	45%	5%	0%	0%	25%	0%	0%	15%	0%
Read-Mostly	5%	14%	0%	10%	5%	19%	0%	33%	0%

Table 5: Failure cause by component and type of cause. The component is described as node or network (net), and type is described as operator error (op), hardware (hw), software (sw), unknown (unk), or environment (env). We have excluded the "disk full/overload" category because of the very small number of failures caused.

	Front-end	Back-end	Network
Online	9.7 (10)	10.2 (3)	0.75 (2)
Content	2.5 (10)	14 (3)	1.2 (2)
RdMostly	N/A (0)	0.17 (1)	1.17 (16)

Table 6: Average TTR by part of service, in hours. The number in parentheses is the number of service failures used to compute that average. This number is provided to give a general notion of the accuracy of the corresponding average.

	Node op	Net op	Node hw	Net hw	Node sw	Net sw	Node unk	Net unk
Online	15 (7)	N/A (0)	1.7 (2)	0.50 (1)	3.7 (4)	N/A (0)	N/A (0)	N/A (0)
Content	1.2 (8)	N/A (0)	N/A (0)	N/A (0)	0.23 (4)	N/A (0)	N/A (0)	1.2 (2)
Read-Mostly	0.17 (1)	0.13 (3)	N/A (0)	6.0 (2)	N/A (0)	1.0 (4)	N/A (0)	0.11 (6)

Table 7: Average TTR for failures by component and type of cause, in hours. The component is described as node or network (net), and fault type is described as operator error (op), hardware (hw), software (sw), unknown (unk), or environment. The number in parentheses is the number of service failures used to compute that average. This number is provided to give a general notion of the accuracy of the corresponding average. We have excluded the "disk full/overload" category because of the very small number of failures caused.

We next analyze the average Time to Repair (TTR) for service failures: time from problem detection to restoration of the service to its pre-failure Quality of Service. We have categorized TTR by the problem root cause location and type[2]. Table 6 shows that although front-end problems vastly outnumber back-end problems in *Online* and *Content* in terms of causing service failures, the time needed to repair problems with the front-end is smaller than the time needed to repair problems with the back-end. There are different

2. The Time to Repair statistics for *Online* and the data for the tables in Section 4.3 were taken from a different set of problems than were used to generate the previous tables and graphs in this section. We have no reason to believe these problems are any different from the other problems we analyzed. For the other services, a subset of the reports used in the other tables and graphs in this section were used to compute average TTR. In particular, only problem reports from which an accurate estimate of service interruption time could be estimated, were used.

	Front-end	Back-end	Network
Online	76%	5%	19%
Content	34%	34%	30%

Table 8: Component failure cause by location in the service.

	Node op	Net op	Node hw	Net hw	Node sw	Net sw	Node unk	Net unk	Env
Online	12%	5%	38%	5%	12%	4%	4%	5%	0%
Content	18%	1%	4%	1%	41%	1%	1%	27%	1%

Table 9: Component failure cause by type. The component is described as node or network (net), and type is described as operator error (op), hardware (hw), software (sw), unknown (unk), or environment. We have excluded the "disk full/overload" category because of the very small number of failures caused.

reasons for the two services: for *Content* this was due to incidents involving serious back-end software bugs that were difficult to isolate and fix, while in *Online* this was due to an operator error that was extremely difficult to undo (a repartition of users that was performed incorrectly).

Table 6 and Table 7 indicate that networking problems generally have the smallest Time to Repair. This is because the locus of a network problem is usually easy to determine, and the fix is generally rebooting, rebooting and reloading (possibly updated) firmware, or switching out a failed component with another identical one.

Finally, Table 8 and Table 9 show the source location and type of component failures in *Online* and *Content*. As with service failures, component failures arise primary in the front-end. However, hardware and/or software problems dominate operator error. 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.

4.3 Techniques for mitigating failure

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

- prevent the original component fault
- 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 service failure is observed)
- decrease the Time to Detection (TTD): time from component failure to detection of the failure

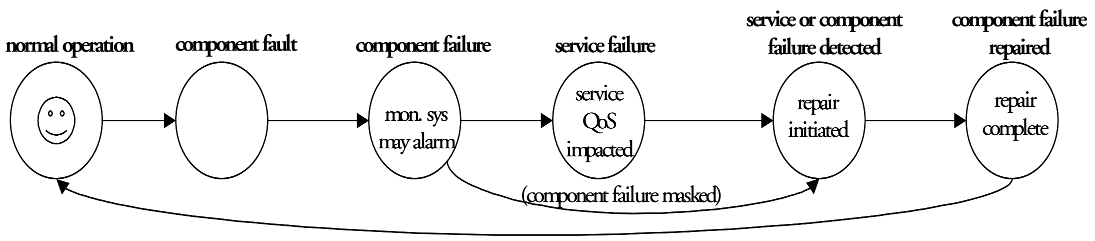

Figure 9: Timeline of a failure. The system starts out in normal operation. A component fault, such as an operator configuration error, a software bug, or an alpha particle flipping a memory bit, may or may not eventually lead the affected component to fail. If the component fails, recovery may be initiated automatically (as in the case of a CPU panic causing a node to automatically reboot), and/or the service's monitoring system may notify an operator. The component failure may cause a service failure if it is not sufficiently masked. Eventually the component failure will be noticed and repaired, returning the system to normal operation. "Time to Repair" is therefore the time between "service or component failure detected" and "component failure repaired."

- decrease the Time to Repair (TTR): time from component failure detection to component repair. (This corresponds to the time during which system QoS is degraded.)

Figure 9 shows how the above categories can be viewed as a state machine or timeline, with component fault leading to component failure, causing a user-visible service 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

- **testing (pre-test or online-test)**: testing the system for correct behavior given normal inputs, either in components before deployment or in the production system. Pre-deployment testing prevents component faults in the deployed system, and online testing detects faulty components before they fail under normal operation.
- **redundancy (red)**: replicating data, computational functionality, and/or networking functionality [10]. Using sufficient redundancy prevents component failures from turning into service failures.
- **fault injection and load testing (pre-flt/ld or online-flt/ld)**: 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 [26]. Pre-deployment fault injection and load testing prevents components from being deployed that are faulty in their error-handling or load-handling capabilities, and online fault injection and load testing detects components that are faulty in their error-handling or load-handling capabilities before they fail to handle anticipated faults and loads.
- **configuration checking (config)**: using tools to check that low-level configuration files meet sanity constraints. Such tools could prevent faulty configuration files in deployed systems.
- **isolation (isol)**: increasing isolation between software components, to reduce failure propagation [10]. Isolation can prevent a component failure from turning into a service failure.
- **restart (restart)**: periodic prophylactic rebooting of hardware and restarting of software [13]. Periodic component restarts can prevent faulty components with latent errors *(e.g.,* resource leaks such as memory leaks) from failing in the deployed sys-

tem. For example, periodically rebooting a node that is running software that has a memory leak can prevent the leak from growing large enough to cause the node to exhaust its available swap space and thereby crash.

- **exposing** (**exp/mon**): better exposing software and hardware component failure to other modules and/or to a monitoring system, or using better tools to diagnose problems. This technique can reduce time to detect and repair component failures.

Table 10 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 failure state described in the previous paragraph as corresponding to that failure mitigation technique. Table 11 does the same, but shows how each technique could reduce Time to Repair and Time to Detection.

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. Also, note that techniques that reduce time to detect or time to repair component failure reduce the overall service 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 10 and Table 11 we observe that just increasing the degree of redundancy and using online testing would have mitigated 8 and 11 service failures in *Online*, respectively. Furthermore, more thoroughly exposing and reacting to software and hardware failures would have decreased TTR and/or TTD in 8 instances. 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 redundancy, isolation, restart, and pre-deployment and online testing, so Table 10 and Table 11 underestimate the effectiveness of adding those techniques to a system that does not already use them.

All of the failure mitigation techniques described in this section have not only benefits (which we have just analyzed), 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 in the potential to actually reduce reliability (if the complexity of the technique means bugs are likely in the technique's implementation). We analyze the proposed failure mitigation techniques with respect to their costs in Table 12. With this cost tradeoff in mind, we observe that two techniques of adding additional redundancy and better exposing failures offer significant "bang for the buck" in the sense that they would have helped mitigate a relatively large number of failure scenarios (as indicated in Table 10 and Table 11) while incurring relatively low cost.

4.4 Case studies

In this section we examine in detail a few of the more interesting service failures from *Online*.

One interesting case study is "problem #3" from Section 4.5; only one technique we examined, redundancy, would have mitigated the service failure in that case. In that problem, an operator at *Online* accidentally brought down half of the front-end servers using the same administrative shutdown command issued individually to three of the six servers in one of the service groups. Because the service had neither a remote console nor remote power supply control to those servers, an operator had to physically travel to the coloca-

#	Prob type	*Online -test*	*Red*	Online -flt/ld	Config	*Isol*	Pre-flt/ld	*Restrt*	*Pre-test-*
1	FE-OP	X			X				X
2	FE-OP	X							
3	FE-OP		X						
4	FE-OP	X							
5	BE-OP	X			X				
6	BE-SW	X			X				
7	EXT-SW	X							
8	NET-HW		X			X			
9	EXT-OP	X							
10	FE-SW	X				X			
11	BE-HW		X						
12	FE-SW		X	X			X	X	
13	FE-OP	X							
14	FE-SW	X							
15	FE-HW		X						
16	FE-HW	X	X	X					
17	FE-SW		X						
18	NET-OL		X	X			X		
19	BE-HW		X						
	TOTAL	11	9	3	3	2	2	1	1

Table 10: 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 4.2) because here we have also included problems whose sources were external to the services (*i.e.,* due to failure at 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 refer to correctness testing before and after component deployment, respectively. Those techniques that *Online* is already using are indicated in italics; in those cases we evaluate the benefit from using the technique more extensively. We also use the following abbreviations: FE = front-end, BE = back-end, EXT = external, NET = network, OP = operator failure, SW = software, HW = hardware, OL = overload.

tion 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 to the machine, and hence a form of redundancy in the control path of the service. 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 mechanisms, such as an alternate path to controlling an existing set of machines.

#	problem type	*exp/mon*-TTR	*exp/mon*-TTD
1	FE-OP		
2	FE-OP		X
3	FE-OP		
4	FE-OP	X	
5	BE-OP		
6	BE-SW	X	X
7	EXT-SW	X	
8	NET-HW	X	X
9	EXT-OP	X	
10	FE-SW		X
11	BE-HW		
12	FE-SW		
13	FE-OP		X
14	FE-SW	X	X
15	FE-HW		
16	FE-HW	X	X
17	FE-SW	X	X
18	NET-OL		
19	BE-HW		
	TOTAL	8	8

Table 11: How better exposing component failures and better monitoring would have improved Time to Detection and Time to Repair for nineteen service failures from *Online*. These are the same problems as in Table 10.

Another interesting case study is "problem #10," which 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 actually stores usernames and aliases. The daemon continually retried all lookups, eventually overloading the back-end database, and thus bringing down all services that use 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, though pre-deployment testing did not, because the failure scenario was dependent on the interaction between the new software module and the unchanged back-end database that serves the module's requests.

Isolation in the form of throttling back username/alias lookups when they start failing repeatedly during a short period of time would have mitigated this failure, by preventing the database from becoming overloaded and hence unusable for providing services other than username/alias lookups.

A third interesting case study is "problem #13," which provides a good example of how lack of a high-level understanding of how software components interact can lead to service failure. In that situation, it was noticed that users' 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

Technique	Implementation cost	Potential reliability cost	Performance impact
Redundancy	low	low	very low
Isolation	moderate	low	moderate
Restart	low	low	low
Pre-deployment fault/load test	high	zero	zero
Pre-deployment correctness testing	medium to high	zero	zero
Online fault/load testing	high	high	moderate to high
Online correctness testing	medium to high	low to moderate	low to moderate
Tools for checking configurations	medium	zero	zero
Better exposing failures/monitoring	medium	low (false alarms)	low

Table 12: Costs of implementing the failure mitigation techniques described in this section.

valid *Online* user, the server silently drops the message. The service operators at some point had configured that email server not to run the daemon that looks up usernames and aliases (the same daemon as is described in problem #10 above), so the server was silently dropping all news-postings-converted-into-email-messages that it was receiving. The operators accidentally configured that special email server not to run the username/alias lookup daemon because they did not realize that proper operation of that mail server depended on its running the username/alias lookup daemon.

The lessons to be learned here are that operators need an understanding of the high-level dependencies among the software modules that comprise the service, and that software should never silently drop messages or other data in response to an error condition. Online testing would have detected this dependency, while better exposing failures, and improved techniques for diagnosing failures, would have decreased the amount of time needed to detect and localize this problem.

Finally, "problem #9"is representative of the occasional failures *Online* experienced because of problems with external services. In this case, *Online* used an external provider for one of its services[3]. 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 identify because of a lack of thorough error reporting in *Online*'s software and a lack of communication between *Online* and the external service when the external service made the change. Online testing of the security changes would have detected this problem.

Although problems with external providers were rare during the time period of problem tickets we examined from *Online*, this type of problem 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.

Our data argues for the importance of including techniques such as online testing, exposing and logging of operator actions, thorough reporting of errors, good distributed diagnosis tools, and communication among service providers. We believe it is important that APIs for these emerging services allow for easy online testing, automatic propagation of errors to code modules and/or operators that can handle them, and distributed control-flow and data-flow tracing to help detecting, diagnosing, and debugging performance, functionality, and security failures.

4.5 Comments on data collection and analysis

The process of collecting and analyzing failure data from *Online*, *Content*, and *ReadMostly*, and attempting to collect and/or analyze failure data from a number of other organizations, has led us to several observations about the data collection and analysis process itself.

First, our use of operations problem tracking databases may have skewed our results. At both *Online* and *Content*, entries are made in the database both automatically and manually. A new problem ticket is opened automatically when the monitoring software

3. We do not indicate the nature of the outsourced service because it might help to reveal the actual identity of *Online*.

detects a that a monitored component (*e.g.,* a node) is not responding properly. A new problem ticket is opened manually when an operator detects a problem or is notified of a problem by a customer. Therefore an operator can "cover up" an error he or she made if it does not cause a problem that anyone notices. This may happen if the operator fixes the error before it causes an observable component or service failure, or if the operator makes the error in the course of fixing another failure but does not cause a different component or service failure. This imprecision has two important impacts. First, by leaving out operator failures that are "covered up," we are underestimating the operator failure percentage of total component and service failure causes. Second, our observation in Section 4.2.2 that most operator errors were made during the course of normal maintenance (not while fixing a problem) may simply be because operators were reluctant to record errors they made, but were able to correct, while fixing a problem.

Second, the contents of most problem reports were useless except for the name of the component that failed and the human operator narrative log. Both *Online* and *Content* used customized failure tracking databases (based on software by Remedy, Inc.), but vital fields such as problem type, problem cause, whether the problem affected customers, and problem duration, were often filled in wrong. We extracted this information by reading the operator log; because it is timestamped, we could extract problem duration for most problems.

Third, even if operators were forced to fill in the existing problem type and problem cause fields correctly, the problem forms used by the problem tracking databases (1) use nonstandard, and often insufficiently sufficient, categories; and (2) do not allow for multiple causes of a problem to be expressed. The first problem could be addressed by instituting a standard taxonomy for problem causes. We used an *ad hoc* taxonomy in this section, based on dividing root cause location into front-end, back-end, or network, and root cause type into the cross-product of <node or network> and <hardware, software, operator, or unknown>. An additional improvement would be to allow multiple failure causes to be indicated, with the rule that the components contributing to the failure (starting with the root cause) be listed in the temporal order in which they contributed to the problem. This listing of all contributors to a failure would allow a variety of information to be computed automatically, such as the frequency and magnitude of cascading failures, the number of components typically involved in cascades, which types of failures are most likely to be part of a cascade (and hence suggestions where failure isolation boundaries needed to be strengthened), and the like.

Fourth, even if operators were forced to fill in the existing problem duration field accurately, the services do not measure true customer impact by computing a metric like customer-minutes [16]. Computing this metric would require knowing the number of customers who were using the service or the portion of the service that is affected during the service failure. Alternatively, customer-minutes could be estimated by using historical data about the number of customers who are generally using the service, or the portion of the service that is affected, at about the time and day of the week of the failure.

Finally, we note that we used a combination of text files and spreadsheets to record and tabulate the data presented in this section. This was a mistake--we should have kept the data in a database of our own. Using text files and spreadsheets, we had to make a new pass over the entire dataset each time we wanted to try a different categorization of the data or ask a new "question" about it. The flexible querying enabled by databases would have allowed us to avoid numerous passes over the source data. Certainly we now know what the appropriate database schema would look like for future failure analysis data collection.

4.6 Conclusion

From our examination of the failure data from *Online*, *Content*, and *ReadMostly*, we conclude that

- operator error is the leading contributor to user-visible failures;
- operator error is the most difficult type of failure to mask, and generally comes in the form of configuration mistakes;
- contrary to conventional wisdom, front-end software can be a significant cause of user-visible failure, primarily due to operator configuration mistakes;
- back-end failures, while infrequent, take longer to repair than do front-end failures; and

So, as we ask in the title of this report, what can be done to avoid or mitigate failures?

Clearly, better online testing would have mitigated a large number of system failures in *Online*. The kind of online testing that would have helped is fairly high-level 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 with the other services we examined as well, so it seems worth pursuing in real systems.

Online fault injection and load testing would also have been helpful in *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 those faults and error conditions, and then monitoring for a response, requires potentially even more work than does online testing. Moreover, online fault injection and load testing essentially require an isolated subset of the production service to be used, because of the threat to reliability and performance that they pose if used in the standard production cluster. But we found that, despite the best intentions, test clusters tend to be set up slightly differently than the production cluster, so any partitioning mechanism needs to be extremely transparent, perturbing the configuration of the system being tested as little as possible.

Despite the huge contribution of operator error to service failures, 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 redundancy. We found that tools to sanity check configuration files against actual system configuration and operator *intent* would have helped to avoid most operators errors. 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 how the service's architecture should appear and how the service it should work functionally, such that the specification could be checked against the existing configuration.

An overarching difficulty related to diagnosing and fixing problems relates to problem solving with multiple entities. Even single-function Internet services like *Content* and *ReadMostly* require coordinated activity by multiple administrative entities for diagnosing and solving some problems. These entities include the network 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 (in cases where customer problems are escalated to the service's operations staff). Today, this coordination is almost entirely handled manually, via telephone calls to contacts at the various services. Tools for determining the root cause of problems across administrative domains are rudimentary, e.g., traceroute, and the tools generally cannot distinguish between certain types of problems, such as end-host failures from 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, and that may be the most affected by it. These problems lead to increased Mean Time to Repair for problems.

Clearly, improvements could be made in the area of tools and techniques for problem diagnosis and repair across administrative boundaries. This issue is likely to become even more important in the age of composed network services built on top of emerging platforms such as Microsoft's .NET and Sun's SunOne.

Qualitatively we can say that redundancy, isolation through partitioning into stateless front-ends and stateful back-ends, incremental rollout of code, integrating operations and development roles, periodic component restart, and offline component testing, are all important, and already widely-used, techniques for achieving high availability in large-scale Internet services. We note that the observed need for more redundancy in *Online* was more an artifact of *Online*'s architecture than a valid generalization across multiple services. In particular, their use of a single back-end database to store information about customer billing, passwords, and so on, represented a single point of failure that caused many service failures. Similarly, the use of only six "storefull" machines per service group frequently led to service failure in the face of correlated failures on a few of those machines. The other services we examined generally used sufficient redundancy.

Section 5

Related work

Because large-scale commercial Internet services are a relatively new phenomenon, their architecture, operational practices, and failure causes are relatively unstudied. Some have studied or made suggestions for appropriate architectures of such sites [2] [21] but we are not aware of any studies of operational practices or failure causes.

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 [10] [11] is responsible for the most widely cited studies of reliable computer system failure data. In 1986 he found that human error was the largest single cause of failure in deployed Tandem systems, accounting for 42% of failures, with software the runner-up at 25%. Somewhat amusingly, we found almost identical percentages at *Online* and *Content* (an average of 39% for node operator error and 24% for node software). He found hardware to be responsible for 18% of failures and environmental problems to be responsible for 14%; we found that node hardware and environmental problems contributed negligibly to failures in *Online* and *Content*, but that networking problems largely replaced those two categories in causing problems (responsible for about 20% of failures). In 1989, however, he found that software had become the major source of outages (55%), swamping the second largest contributor, system operations (15%). He found that hardware and software outages have an MTTR of 4 hours and operations outages an MTTR of 10 hours, but that these distributions have very high variance. We are wary to compare the results in Table 7 to our results because we found hardware TTR was so much smaller than software TTR, and operations TTR was so widely varying. Finally, Gray also looked at the length of fault chains; though we did not analyze this quantitatively, he found longer fault chains than we did (Gray found 20% of fault chains were of length three or more, with only 20% of length one, whereas we found that almost all chains were of length two (the system was designed to mask a component failure but some other problem caused it not to) or one (the system was not designed to mask a particular failure mode).

Kuhn [16] examined two years of data on failures in the Public Switched Telephone Network as reported to the FCC by telephone companies. He concluded that human error was responsible for more than 50% of failure incidents, and about 30% (still the largest single cause of failure) in terms of customer impact (measuring in customer minutes, *i.e..,* number of customers impacted multiplied by number of minutes the failure lasted). Enriquez [9] extended this study by examining a year's worth of more recent data and also considering the blocked calls metric collected on the outage reports. 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 *et al.* [32] examined failures in a network of 69 SunOS workstations but only divided problem root cause into network, non-disk machine problems, and disk-related machine problems. Kalyanakrishnam *et al.* [15] studied six months of event logs from a LAN of Windows NT workstations used for mail delivery, to determine the causes of machines rebooting. They found that most problems are software-related, and that average downtime is two hours. They used a state machine model of system failure states similar in spirit to, though more detailed than, our Figure 9 to describe failure timelines on a single node. In a closely related study, Xu *et al.* [34] studied 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. 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 (18% and 8%, respectively), and that system software and planned maintenance caused the largest amount of total downtime (22% and 13%, respectively). 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 [28] [29] examined software defects in MVS, DB2, and IMS. Tang and Iyer [31] did the same for the VAX/VMS operating system in two VAXclusters. Lee and Iyer [18] categorized software faults in the Tandem GUARDIAN operating system. Murphy and Gent [23] examined causes of system crashes in VAX systems between 1985 and 1993; in 1993 they found that hardware was responsible for about 10% of failures, software for about 20% of failures, and system management for a bit over 50% of failures. Lancaster and Rowe [17] recently examined NFS data availability in Network Appliance's NetApp filers. They found that power failures and software failures were the largest contributors to downtime, with operator failures contributing negligibly.

Finally, there is a large body of existing work in the areas of system monitoring, diagnosis, and configuration. SNMP is the most commonly used protocol for remote monitoring of networked devices [6]. A number of free and commercial products use SNMP alerts as the basis of their monitoring systems; examples include Netcool [20], OpenView [12], and various offerings from Tivoli [14].

True root cause analysis and problem diagnosis are significantly more complicated than simple monitoring. Among the most sophisticated systems recently developed is the auto-diagnosis subsystem that comprises part of Network Appliance's Data ONTAP operating system [1]. It combines continuous monitoring, protocol augmentation, cross-layer analysis, active probing, and automatic configuration change tracking to detect and localize hardware and software problems in NetApp fileservers.

Tools for automating the configuration of computer systems date back at least to the early 1980's, with McDermott's R1, which was used to suggest how to physically configure VAX 11/780 systems given a customer's order [19]. Recent tools for deploying and configuring software include Sun's JumpStart [30] and Burgess's Cfengine [5].

Section 6

Conclusion and future work

We have studied the architecture, operational practices, and failure data of three large-scale Internet services. Despite differences in number of users, number of machines, and workload, these service, and others we surveyed but did not describe in this report, share a number of similarities with respect to their node, site, and geographic structure and building blocks, as well as many of their operational practices. From a study of approximately 200 component failures and approximately 60 user-visible service failures, we draw conclusions about failure distribution across parts of the service, the primary contributors to failure and their associated Times to Repair, and the effectiveness of redundancy techniques at preventing component failure from turning into service-level failure. Finally, we describe a "thought experiment" in which we explore how various failure mitigation techniques might have prevented component faults from occurring, component faults from turning into component failures, component failures from turning into service failures, and might have reduced time to detect and time to repair failures.

This work suggests a number of future directions, in the areas of both failure data collection and analysis, and benchmarking.

6.1 Future directions for failure data collection and analysis

There are a number of avenues for expanding the data collection and analysis work described in this paper. First, it would be interesting to examine data from a broader spectrum of services, particularly over a range of service type (*e.g.,* including e-commerce, an important category of service that we did not examine in this study), maturity, rate of change, architecture, and operational practices. In doing so, one could attempt to correlate some of these service characteristics to overall service dependability. Also, more services means more data, which would increase the statistical validity of our results.

Second, it would be interesting to collect more data from the three services we have already examined, both to increase the statistical significance of the data, and as part of a longitudinal study of one or more services. A longitudinal study might address questions such as whether a service becomes more reliable over time (because software quality improves, operators learn from their mistakes, *etc.*). While *Online* and *ReadMostly* are relatively mature services, *Content* would be an excellent candidate for such a study, because it did mature in both software and operational practices over the time period of the dataset to which we have access (although not the single month we examined in this study). Also, with more data (from the services we have already examined or from new ones) we would be able to examine the role of failure propagation in causing service failures, something we were not able to do because there simply were not enough problem reports that actually involved multiple component failures leading to a larger failure. It would be interesting to see what components are most often involved in such cascading failure scenarios, to inform the implementation of better isolation boundaries or to otherwise decorrelate correlated failures.

A third avenue for expanding the work described in this paper is to broaden the range of quantitative metrics we are collecting. Among the possibilities are

- the time between the events of fault occurrence, component failure, service failure (or masking of the component failure), problem detection, service recovery (*i.e.,* original pre-failure service QoS restored), service restored to being fully redundant, and fixing the original flaw (by repairing a software bug, fixing a maintenance process, and the like)
- metrics that take into account the number of customers impacted, *e.g.,* "customer-minutes" or an Internet service analog of "blocked calls," both of which are used to characterize failures in the Public Switched Telephone Network [16]
- instead of treating service failure impact as binary (either a failure was severe enough to be considered a service failure or it was not), metrics that take into account the type and degree of Quality of Service degradation, *e.g.,* the fraction of the service that is unavailable or the percent reduction in service response time or throughput
- metrics that take into account revenue lost to the service itself and/or its customers, as well as the cost of repairing failures
- metrics internal to the service, as opposed to purely user-visible ones (*e.g.,* number of false alarms relative to true alarms)
- number of operators needed to repair different types of problems
- number of components involved in cascading failure scenarios
- number of problems that are automatically repaired (*e.g.,* a RAID system reconstructing onto a spare disk) as opposed to simple masked through redundancy

Fourth, it would be useful to investigate additional qualitative information related to problems, possibly leading to additional taxonomies for classifying them. We have implicitly used two taxonomies in this study to describe root cause: "location" of the failure (front-end, back-end, network) and "type" of the failure (the two-dimensional cross product of <node, network> x <hardware, software, operator, overload/disk full, unknown>). But there are other ways to classify failures. For example, one could look at the scope of the impact--a single component, a node, a rack of servers, a datacenter, or the entire service. Or, one could try to trace problems back to the fault rather than just the type of component failure, and then describe the fault using a taxonomy like that in [25] (*e.g.,* physical *vs.* logical, manufacturing *vs.* design *vs.* operation, and so on). Additional interesting attributes for partitioning problems include whether the fault originated in the service or with a vendor, whether a particular failure mode was anticipated or not, and what the exact causes of operator errors were (configuration *vs.* procedural, initial deployment *vs.* routine maintenance *vs.* responding to an existing problem, and so on). Finally, it would be interesting to classify problems according to what actions the services took in response to the problems (*e.g.,* using a short-term fix versus fixing the root cause); what type of fix, if any, was used (software *vs.* process *vs.* architectural), and whether the fix prevented the problem from recurring.

Fifth, it would be interesting to correlate the failures we observed to actual changes in Quality of Service as observed by end-users (or simulated end-users, *e.g.*, using a globally distributed testing service such as Keynote). As noted in Section 4.1, the study we conducted looked at failures only from a vantage point within the service; we did not actually verify our intuition about which component failures had "significant" end-user visible impacts (in degradation of performance or service functionality or fidelity) and which were truly masked by redundancy.

Sixth, it would be interesting to examine the potential benefit of other failure mitigation techniques that have been proposed, including improvements in operations processes, and to investigate ways to quantify the benefits of potential mitigation techniques.

We have only begun to explore the potential value of mining failure data. Only by allowing many researchers to access failure datasets can the full potential of this data be explored. Our experience was that companies are extremely reluctant to release this data, especially for examination off-site, out of concern for their own privacy and that of vendors and customers who may be mentioned in problem reports. The solution to this problem is to develop a standard for anonymizing failure data. Such a system requires a standard schema for problem reports, including a taxonomy of underlying fault causes and failed component locations and types. Of course, the accuracy of such a dataset hinges on operators' filling out the multiple-choice part of problem report forms much more accurately than they did at the two services we examined that use problem tracking databases. Alternatively, a human researcher could read the operator narratives from the problem reports, ignoring the fields that we found were often filled out incorrectly, such as root cause, whether customers were affected, problem duration, and so on, and coerce this data into a standard format. Either way, such data then could be put into a database accessible by researchers either for downloading or through interactive querying via the web. We believe that this dataset would be of similar value as existing datasets of CPU instruction traces in the architecture community, network intrusion traces in the security community, and SQL workloads in the database community. Of course, one common use of such datasets is benchmarking (new CPU and system architecture designs, new intrusion detection techniques, and new database architectures and implementations, in the previous examples), so we next discuss how the data we have collected could be used to drive dependability and recoverability benchmarks.

6.2 Making use of failure data: benchmarks

In this report we have examined failure data to learn why systems fail. In addition to studying this data to point to how individual services and services as a whole can improve their dependability, we can use the data to drive service-level dependability or recovery benchmarks akin to the availability benchmarks described in [4]. Such benchmarks require a faultload, and the data we have collected, including that related to relatively under-studied failure causes such as those induced by humans and those occurring in networks among geographically distributed sites, could be turned into a stochastic fault model to drive dependability or recoverability benchmarks. Non-binary Quality of Service metrics such as throughput, fraction of service functionality available, response time, financial cost of unavailability, and the like could be used to measure the impact of failures and subsequent recovery (be it automatic or human-controlled) while a workload is presented the system.

To perform a recoverability benchmark, the data presented in this report, perhaps augmented with data from other services, could be used to establish the set of component failures that services experience. The simplest way to evaluate a service is to inject each of the component failure modes we have observed (including operator failures) one at a time and to measure the service's Quality of Service response. The Quality of Service decrease while the service is affected is an inverse measure of its "recoverability." These responses to component failures can be reported individually, to indicate how a service could improve its recoverability. A somewhat more sophisticated evaluation would weight the responses to these individual recovery events by the relative frequency with which the different classes of failures occur in the service being benchmarked, to indicate an aggregate measure of the service's recoverability. Finally, one could test multi-failure scenarios, as described in [4], to test system response to cascading failures and human repair activity following an initial failure.

Benchmarking dependability is similar to benchmarking recoverability, but it requires information about the absolute frequency with which the failures occur. In other words, for dependability benchmarks, we need to know the MTTF for each component that will fail, whereas a service's "recoverability" depends only on the mix of component failures it is likely to experience. The recoverability benchmark described above can be turned into a dependability benchmark by weighting the QoS responses to the recovery events by the frequencies of the component failures computed from their MTTF.

We must also address the issue of what workload should be presented to the system while it is being benchmarked. In this respect, Internet service benchmarks are more difficult to design than are benchmarks for traditional applications, *e.g.,* SPECweb and SPECmail [27], because for Internet services the workload is very site-specific. Among existing benchmarks, TPC-W [33] comes closest to approximating what the workload for an Internet service benchmark might look like, in the sense that it specifies a mixture of high-level user tasks rather than a protocol-level workload (as in SPECweb and SPECmail). It may be possible to formulate a taxonomy of Internet service functionality (email, news, instant messenger, web search, retrieving static web content, auctions, traditional web-based product ordering, and so on), and then to develop an application-level workload similar to TPC-W's but targeted to each of these services. Then, an Internet service could be benchmarked for the particular mixture of services it provides.

Note that not only user activities, but also operator activities, are a vital component of a service-level benchmark workload. Because Internet services operate 24x7, administrative tasks such as preventative hardware maintenance, software upgrades, scaling, rebalancing data across back-end machines, and the like should be considered a standard part of the workload. The data we collected for this report did not include routine administrative tasks (except when they resulted in a component or service failure), but one could ask the administrators of these or similar services to record their daily activities over a period of several months. This data could then be used to build a stochastic model of operator activities (similar to the stochastic failure model described above and the stochastic workload mixture utilized in TPC-W) to be used as part of the service workload.

Finally, one can imagine developing languages to specify user dependability requirements and failure models of system components (hardware, software, networking, and human operators) so that tools can be developed (1) to match user performance and dependability requirements to system capabilities, and (2) to automate benchmark generation and the fault injection process.

References

[1] G. Banga. Auto-diagnosis of field problems in an appliance operating system In *Proceedings of the 2000 USENIX Annual Technical Conference*, 2000, 2000.

[2] E. Brewer. Lessons from giant-scale services. *IEEE Internet Computing*, July 2001.

[3] 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.

[4] A. Brown and D. A. Patterson. Towards Availability Benchmarks: A Case Study of Software RAID Systems. *Proceedings of the 2000 USENIX Annual Technical Conference*, San Diego, CA, 2000.

[5] M. Burgess. Cfengine: a site configuration engine. USENIX Computing systems, vol. 8, no. 3, 1995.

[6] J. Case, M. Fedor, M. Schoffstall, and J. Davin. A simple network management protocol (SNMP). Internet RFC 1157.

[7] R. Chillarege, S. Biytani, and J. Rosenthal. Measurement of failure rate in widely distributed software. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing*, 1995.

[8] J. Christmansson and R. Chillarege. Generation of an error set that emulates software faults based on field data. In *Proceedings of FTCS-26*, 1996.

[9] P. Enriquez, A. Brown, and D. A. Patterson. Lessons from the PSTN for dependable computing. Submission to *Workshop on Self-Healing, Adaptive and self-MANaged Systems*, 2002.

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

[11] J. Gray. Why do computers stop and what can be done about it? *Symposium on reliability in distributed software and database systems*, 3-12, 1986.

[12] Hewlett Packard. HP OpenView. http://www.openview.hp.com/

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

[14] IBM. Tivoli software. http://www.tivoli.com/

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

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

[17] L. Lancaster and A. Rowe. Measuring real-world data availability. *Proceedings of LISA 2001*, 2001.

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

[19] J. McDermott. R1: A Rule-Based Configurer of Computer Systems. *Artificial Intelligence* vol. 19, no. 1, 1982.

[20] Micromuse. Netcool. http://www.micromuse.com/

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

[22] B. Murphy. Windows 2000 Dependability. *Proceedings of the IEEE International Conference on Dependable Systems and Networks*, 2000.

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

[24] D. A. Patterson. A simple way to estimate the cost of downtime. Submission to 16th Systems Administration Conference (LISA '02), 2002

[25] D. A. Patterson. Five nines of availability, million hour MTBF, and other myths. Submission to *;login*, 2002.

[26] 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. UC Berkeley Computer Science Technical Report UCB//CSD-02-1175, 2002.

[27] Standard Performance Evaluation Corporation. http://www.spec.org/

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

[29] M. Sullivan and R. Chillarege. Software defects and their impact on system availability--a study of field failures in operating systems. In *Proceedings of the 21st International Symposium on Fault-Tolerant Computing*, 1991.

[30] Sun Microsystems. Solaris JumpStart.
http://wwws.sun.com/software/solaris/archive/8/ds/ds-webstart/

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

[32] 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.

[33] Transaction Processing Council. http://www.tpc.org/

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