Architecture and Dependability of Large-Scale Internet Services

An analysis of the architectures and causes of failure at three large-scale Internet services can help developers plan reliable systems offering maximum availability.

David Oppenheimer and David A. Patterson
University of California at Berkeley

In the past few years, the popularity of large-scale Internet infrastructure services such as AOL, Google, and Hotmail has grown enormously. The scalability and availability requirements of these services have led to system architectures that diverge significantly from those of traditional systems like desktops, enterprise servers, or databases. Given the need for thousands of nodes, cost necessitates the use of inexpensive personal computers wherever possible, and efficiency often requires customized service software. Likewise, addressing the goal of zero downtime requires human operator involvement and pervasive redundancy within clusters and between globally distributed data centers.

Despite these services' success, their architectures — hardware, software, and operational — have developed in an ad hoc manner that few have surveyed or analyzed.[1,2] Moreover, the public knows little about why these services fail or about the operational practices used in an attempt to keep them running 24/7. Existing failure studies have examined hardware and software platforms not commonly used for running Internet services, or in operational environments unlike those of Internet services. J. Gray, for example, studied fault-tolerant Tandem systems;[3] D. Kuhn studied the public telephone network;[4] B. Murphy and T. Gant examined VAX systems;[5] and J. Xu and his colleagues studied failures in enterprise-scale Windows NT networks.[6]

As a first step toward formalizing the principles for building highly available and maintainable large-scale Internet services, we are surveying existing services'

 1089-7801/02/$17.00©2002 IEEE

Table 1. Comparison of Internet service application characteristics with those of traditional applications.

Characteristic	Traditional desktop applications	Traditional high-availability applications	Large-scale Internet service applications
Dominant application	Productivity applications, games	Database, enterprise messaging	E-mail, search, news, e-commerce, data storage
Typical hardware platform	Desktop PC	Fault-tolerant server or failover cluster	Clusters of hundreds to thousands of cheap PCs, often geographically distributed
Software model	Off-the-shelf, standalone	Off-the-shelf, multitier	Customized, multitier
Software release frequency	Months	Months	Days or weeks
Networking environment	None (standalone)	Enterprise-scale	Within cluster, on the Internet between data centers, and to and from user clients
Operator	end user	Corporate IT staff	Service operations staff, data center/collocation site staff
Typical physical environment	Home or office	Corporate machine room	Collocation facility
Key metrics	Functionality, interactive latency	Availability, throughput	Availability, functionality, scalability, manageability, throughput

architectures and dependability. This article describes our observations to date.

A New Application Environment

The dominant software platform is evolving from one of shrink-wrapped applications installed on end-user PCs to one of Internet-based application services deployed in large-scale, globally distributed clusters. Although they share some characteristics with traditional desktop and high-availability software, services comprise a unique combination of building blocks, with different requirements and operating environments. Table 1 summarizes the differences between the three types of applications.

Users increasingly see large-scale Internet services as essential to the world's communications infrastructure and demand 24/7 availability. As Table 1 suggests, however, these services present an availability challenge because they

- typically comprise many inexpensive PCs that lack expensive reliability features;
- undergo frequent scaling, reconfiguration, and functionality changes;
- often run custom software that has undergone limited testing;
- rely on networks within service clusters, between geographically distributed collocation facilities, and between collocation facilities and end-user clients; and
- aim to be available 24/7 for users worldwide, making planned downtime undesirable or impossible.

On the other hand, service architects can exploit some features of large-scale services to enhance availability:

- Plentiful hardware allows for redundancy.
- Geographic distribution of collocation facilities allows control of environmental conditions and resilience to large-scale disasters.
- In-house development means that operators can learn the software's inner workings from its developers, and can thus identify and correct problems more quickly than IT staffs running shrink-wrapped software that they see only as a black box.

We examine some techniques that exploit these characteristics.

Maintainability is closely related to availability. Not only do services rely on human operators to keep them running, but the same issues of scale, complexity, and rate of change that lead to failures also make the services difficult to manage. Yet, we found that 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.

Architectural Case Studies

To demonstrate the general principles commonly used to construct large-scale Internet services, we examine the architectures of three representative

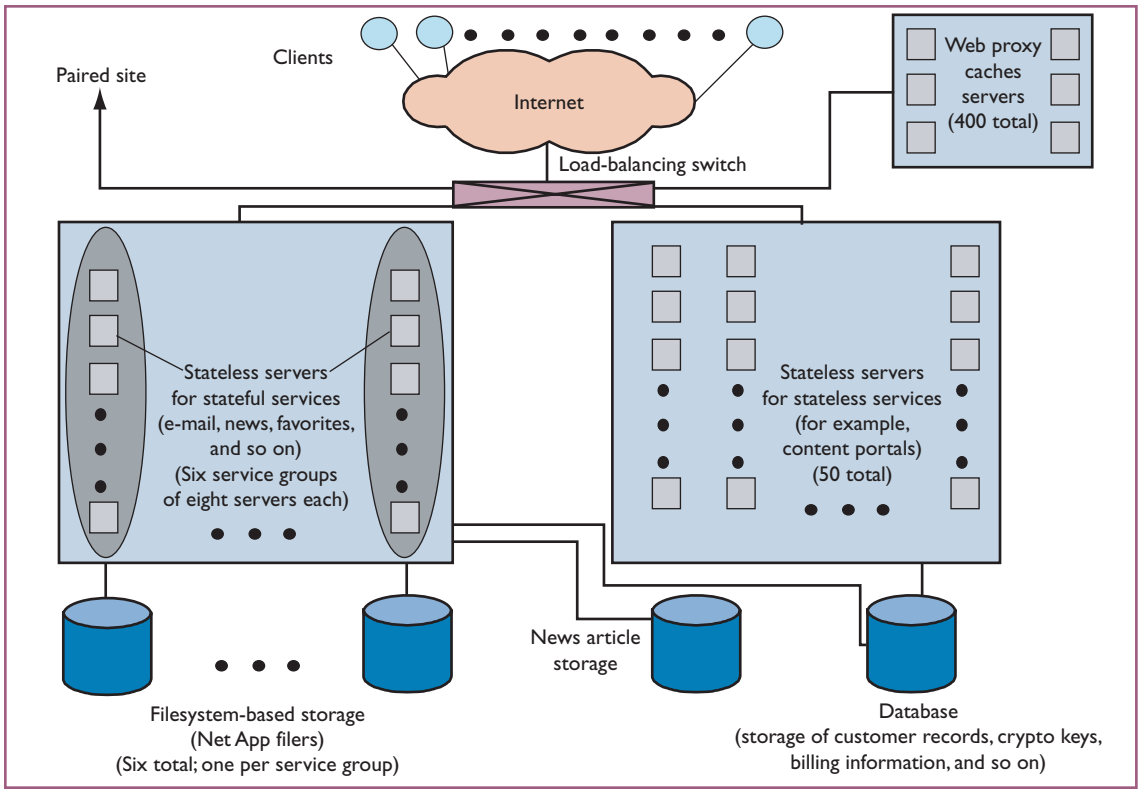

Figure 1. Architecture of an Online site. Depending on the feature selected, the client software chooses to route the user request to a Web proxy cache server, one of 50 stateless servers, or one of the eight servers from the user's service group. Network appliance servers store persistent state, which cluster nodes access via Network File System (NFS) over the user datagram protocol (UDP). A leased network connection links the cluster to a second site at a collocation facility.

services, which we will call:

- Online—an online service/Internet portal
- Content—a global content-hosting service
- ReadMostly—a high-traffic Internet service with a very high read-to-write ratio

Table 2 highlights some of the services' key characteristics. To keep the services' identities confidential, we have abstracted some of the information to make it more difficult to identify them.

Architecturally, these services

- reside in geographically distributed collocation facilities,
- consist largely of commodity hardware but custom software,
- achieve improved performance and availability through multiple levels of redundancy and load balancing, and
- contain a load-balancing tier, a stateless (stores no persistent state except operating system code) front-end tier, and a stateful (stores persistent data) back-end tier.

As Table 2 (next page) shows, the primary differences between these services are load and read/write ratio.

Geographic Server Distribution

At the highest level, many services distribute their servers geographically. Online distributes its servers between its headquarters and a nearby collocation facility; ReadMostly uses a pair of facilities on the United States East Coast and another pair on the West Coast; and Content uses four facilities: one each in Asia, Europe, and the East and West Coasts of the U.S. All three services use this geographic distribution for availability, and in all but Content the redundant data centers share in handling user requests to improve performance.

When using distributed data centers to share load, services can employ several mechanisms to direct user queries to the most appropriate site. The choice of site generally takes into account each site's load and availability.

Content pushes this functionality to the client, which runs custom software pointed to one primary and one backup site. To reduce administra-

Table 2. Characteristics of the large-scale Internet services examined.

Characteristic	Online	Content	ReadMostly
Hits per day	~100 million	~7 million	~100 million
Number of machines	~500, at two data centers	~500, at four data centers plus client sites	>2,000, at four data centers
Hardware	Sparc and x86	x86	x86
Operating system	Solaris	Open-source x86	Open-source x86
Relative read/write ratio	High	Medium	Very high (and users update very little data)

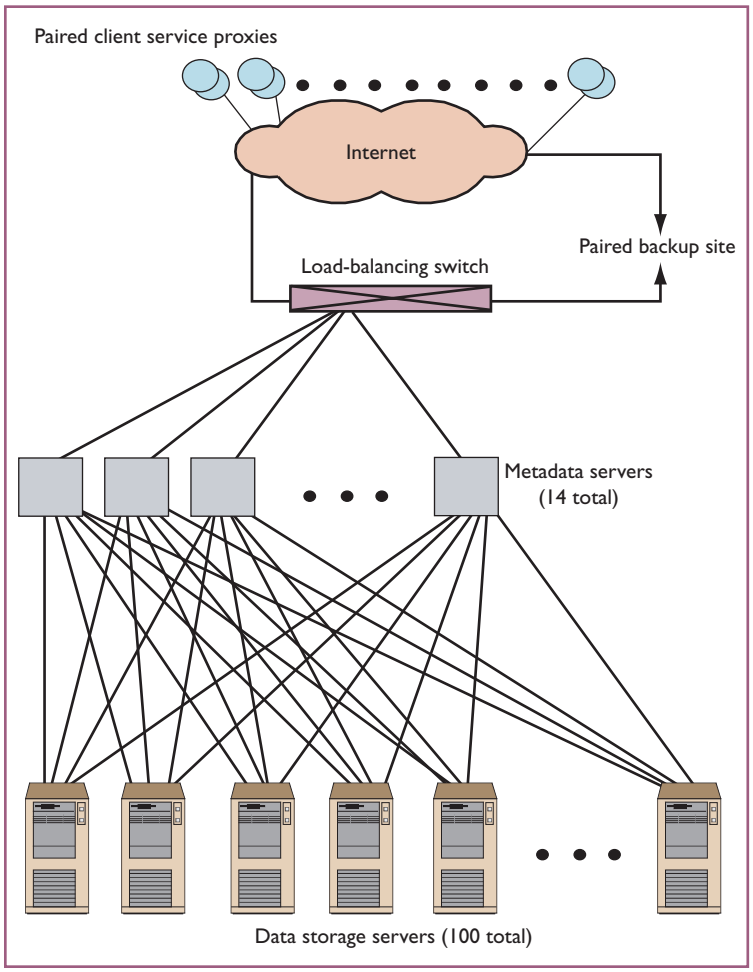

Figure 2. Architecture of a Content site. Stateless metadata servers provide file metadata and route requests to the appropriate data storage server. These servers, which use commodity hardware and run custom software, are accessed via a custom protocol over UDP. The Internet connects each cluster to its twin backup site.

tive complexity, the two sites work in redundant pairs — the service points some clients to one pair of sites, and other clients to another pair. A client's primary server site propagates updates to its secondary server site nightly.

Whenever an Online client connects, a server at the company's headquarters provides the client with an up-to-date list of the best servers to contact. The servers might be at the company's headquarters or at its collocation facility.

ReadMostly uses its switch vendor's proprietary global load-balancing mechanism to direct users. This mechanism rewrites DNS responses based on sites' load and health information, which it collects from cooperating switches at those sites.

Single-Site Architecture

A single site's architecture consists of three tiers: load balancing, front-end servers, and back-end servers. Figures 1, 2, and 3 depict the single-site architectures of Online, Content, and ReadMostly, respectively.

Load balancing. To balance load, one or more network switches distributes incoming requests to front-end servers based on the servers' loads. Although 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 services we surveyed use this feature. Instead, they generally use simple round-robin DNS or Layer-4 load distribution to direct clients to the least loaded front-end server. In round-robin DNS, a service advertises multiple IP addresses and continuously reprioritizes the addresses to spread load among the corresponding machines. In Level-4 load distribution, clients connect to a single IP address and the cluster's switch routes the connection to a front-end server.

Content and ReadMostly use Layer-4 request distribution at each site, as does Online for stateless parts of its service (the Web proxy cache and content portals, for example). For the stateful parts of its service (such as e-mail), Online also uses Layer-4 request distribution, but adds a level of stateful front-end load balancing on top. In particular, Online maps a user to one of several clusters (called service groups) based on the user's identity (determined when the user logs in to the

service). The service further load balances each cluster internally using Level-4 load balancing.

A general principle in service building is to use multiple levels of load balancing: among geographically distributed sites, among front ends in a cluster, and possibly among subsets of a cluster.

Front end. Once the service makes the load-balancing decision for a request at the cluster level, it sends the request 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 suitable presentation format). We say "in some cases" because some services, such as Content, return data to clients directly from the back-end server. The exact ratio of back end to front-end servers depends on the type of service and the performance characteristics of the back end and front-end systems. The ratio ranged from 1:10 for Online to 50:1 for ReadMostly.

Depending on its needs, the service might partition the front-end machines by functionality and/or data. Content and ReadMostly use neither characteristic; all front-end machines access data from all back-end machines. Online, however, does both. It partitions for functionality by using three kinds 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 because they act as Internet content caches. For data, Online further partitions the front ends to stateful services into service groups.

In all cases, the front-end machines were inexpensive PCs or workstations. Table 3 summarizes the three services' internal architectures. All three use customized front-end software rather than off-the-shelf Web servers for functionality or performance reasons.

Back end. Back-end servers provide services with the data they need to satisfy a request. These servers store persistent data, such as files (Content and ReadMostly) and e-mail, news articles, and user preferences (Online). The back end exhibits the greatest variability in node architecture: Content and ReadMostly use inexpensive PCs, while Online uses file servers designed especially for high availability. Online uses Network Appliance's Write Anywhere File Layout (WAFL) filesystem, Content uses a custom filesystem, and ReadMostly uses a Unix filesystem.

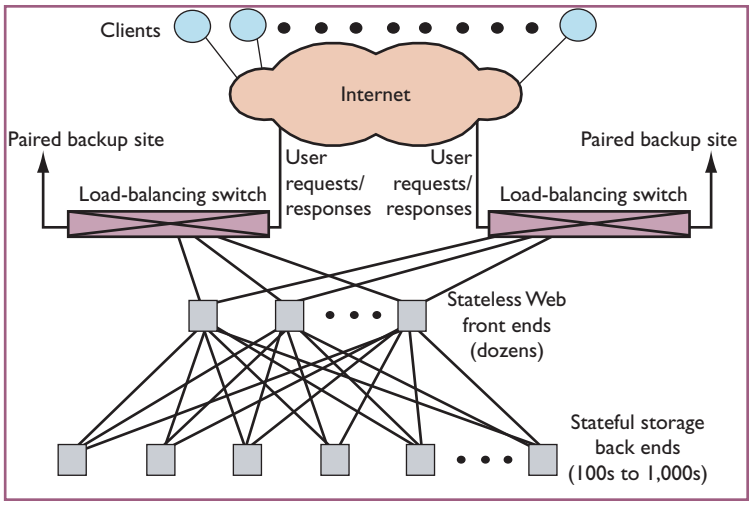

Figure 3. Architecture of a ReadMostly site. Web front ends direct requests to the appropriate back-end storage server(s). Commodity PC-based storage servers store persistent state, which is accessed via a custom protocol over TCP. A redundant pair of network switches connects the cluster to the Internet and to a twin backup site via a leased network connection.

In the back end, data partitioning is key to overall service scalability, availability, and maintainability. Scalability takes the form of scale-out — that is, services add back-end servers to absorb growing demands for storage and aggregate throughput. Online, Content, and ReadMostly represent a spectrum of back-end redundancy schemes for improving availability and performance.

Each Online service group consists of approximately 65,000 users and is assigned to a single back-end Network Appliance filer. Each filer uses a redundant array of inexpensive disks, level 4 (RAID-4), which allows the service group to tolerate a single back-end disk failure. If a single filer fails, however, all users in that group lose access to their data. Moreover, the scheme does not improve performance, because the filer stores only one copy of the data.

Content partitions data among back-end storage nodes. Instead of using RAID, the service replicates each piece of data on a storage server at a twin data center. This scheme offers higher theoretical availability than the Online scheme because it masks single disk failure, single storage server failure, and single site failure (due to networking problems, environmental issues, and the like). Content's back-end data redundancy scheme does not improve performance either, because it assigns only one default site to retrieve data to service user requests, and uses the backup twin site only when

Table 3. Node architectures of the three services studied.

Architecture	Disks	CPUs	Network interface	Memory
Online				
Worker node	Two 40-Gbyte disks	Two 300-MHz Sparc	100-Mbps Ethernet	1 Gbyte
Proxy cache node	Two 18-Gbyte disks	Two 550-MHz x86	100-Mbps Ethernet	1 Gbyte
Content				
Metadata server node	One 30-Gbyte disk	Two 844-MHz x86	100-Mbps Ethernet	512 Mbytes
Storage server node	Six 40-Gbyte disks	Two 550-MHz x86	100-Mbps Ethernet	256 Mbytes
ReadMostly				
Service node	Two 100-Gbyte disks	Two 750-MHz x86	100-Mbps Ethernet	512 Mbytes

the first storage server or its site fails.

Finally, ReadMostly uses full replication to achieve high availability and improve performance. The service replicates each piece of data among nodes in the cluster and among geographically distributed clusters. It load balances incoming requests from the front end among redundant copies of data stored in the back end. This improves both back-end availability and performance by a factor proportional to the number of data copies stored. However, such a redundancy scheme would not be appropriate for a more write-intensive service: The service must propagate updates to all nodes storing a particular piece of data, which slows updates.

Networking. The wide-area networking structure among server sites varies greatly for the three services, whereas single-site networking follows 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-Mbps Ethernet to individual nodes. Although sites commonly use redundant connections between first- and second-level switches, none of the sites use redundant Ethernet connections between nodes and second-level switches, and only one (ReadMostly) uses redundant connections between the collocation facility's network and the first-level switches.

Despite the recent emergence of industry-standard, low-latency, high-bandwidth system-area networking technologies such as Virtual Interconnect Architecture (VIA) and Infiniband, all three services use UDP over IP or TCP over IP within clusters. This is most likely due to cost: a 100BaseT NIC card costs less than US$50 (and is often integrated into modern motherboards), although a PCI card for VIA costs about US$1,000.

Service Operational Issues

The need to develop, deploy, and upgrade services on "Internet time," combined with the size and complexity of service infrastructures, places unique pressures on the traditional software life cycle. This is most apparent in testing, deployment, and operations. The amount of time available for testing has shrunk, the frequency and scale of deployment 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. Online, Content, and ReadMostly address these challenges in various ways.

Testing

All three services test their software before releasing it to their production cluster. Development quality assurance (QA) teams test the software before deployment. These groups use traditional testing techniques such as unit and regression tests, and use both single nodes and small-scale test clusters that reflect the production cluster architecture.

Online has a three-step testing and deployment process for new software and for features added after the software passes unit development QA tests.

- The development group deploys the candidate software on its test cluster. Developers run the software, but the operations team configures it.
- The operations group takes the stable version of the software, deploys it, and operates it on its test cluster.
- The operations group releases alpha and beta versions of the software to the production service. For major releases, the group releases the new version to a subset of users for up to two weeks before rolling it out to the rest of the users.

This testing and deployment methodology integrates the operations team from the very beginning, considering operators as first-class users of the software as much as end users.

Deployment

Deploying software to large clusters requires automation for both efficiency and correctness. The three services use tools developed in-house to deploy and upgrade production software and configurations for applications and operating systems. They all use rolling upgrades to deploy software to their clusters.[1] All three use version control during software development, and Online and ReadMostly use it to manage configuration as well. Online always keeps two versions of the service software installed on every machine, allowing administrators to revert to an older version in less than five minutes if a new installation goes awry.

Daily Operation

Service monitoring, problem diagnosis, and repair are significant challenges for large-scale Internet services. Reasons for this include

- frequency of software and configuration changes,
- scale of the services,
- unacceptability of taking the service down to localize and repair problems,
- complex interactions between application software and configuration, operating system software and configuration,
- collocation site networks,
- 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 presents a key challenge to maintainability. Data must be distributed among back-end storage nodes to balance load and to avoid exceeding server storage capacity. As the number of back-end machines grows, operators generally repartition back-end data to incorporate the new machines. Currently, humans make partitioning and repartitioning decisions, and the repartitioning process is automated by tools that are at best ad hoc.

The scale, complexity, and speed of Internet service evolution leads to frequent installations, upgrades, problem diagnosis, and system component repair. 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. The Internet services we studied encourage frequent dialogue between operators and developers during upgrades, problem diagnosis, and repair — for example, when 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). As a result, these services intentionally blur the line between "operations" and "software development" personnel.

Despite automated tasks such as monitoring and deploying software to a cluster, human operators are key to service evolution and availability. Moreover, because an operator is as much a user of Internet service software as is an end user, developers should integrate operator usability throughout the software development process. This is especially true because many operational tasks are not fully automated or not automated at all — such as configuring and reconfiguring software and network equipment, identifying a problem's root cause, and fixing hardware, software, and networking issues. Conversely, much opportunity exists for more sophisticated and standardized problem diagnosis and configuration management tools, as all of the services we surveyed used some combination of their own customized tools and manual operator action for these tasks.

Characterizing Failures

Our architectural and operational study reveals pervasive use of hardware and software redundancy at the geographic and node levels, and 24/7 operations centers staffed by personnel who monitor and respond to problems. Nonetheless, outages at large-scale Internet services are relatively frequent.

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 uses problem-tracking databases to record information about component and service failures. Online and Content gave us access to these databases; ReadMostly gave us access to their problem post mortem reports.

Component and Service Failures

We define a component failure as a failure of some service component to behave as expected — a hard drive failing to spin up when it is power-cycled, a

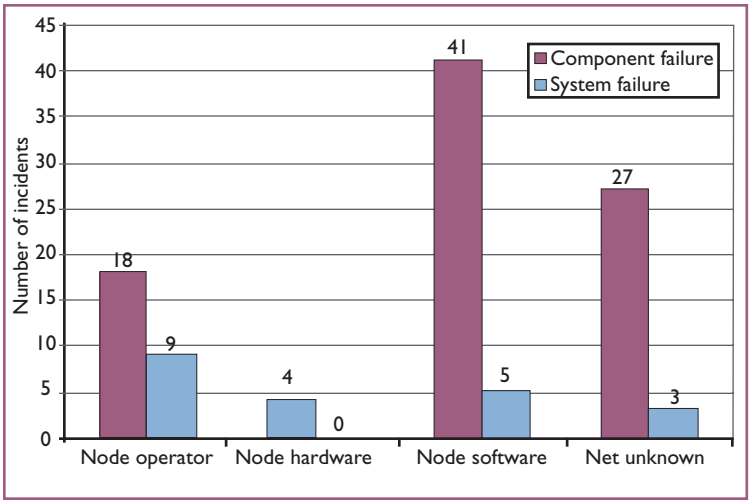

Figure 4. Component failures and resulting service failures for Content site. For all categories but node operator, 11 percent or fewer component failures led to a service failure.

Table 4. Failure cause by type for Content and ReadMostly (in percentages).

Error type	Content	ReadMostly
Node operator	45	5
Network operator	5	14
Node hardware	0	0
Network hardware	0	10
Node software	25	5
Network software	0	19
Unknown node error	0	0
Unknown network error	15	33
Environment	0	0

software crash, an operator misconfiguring a network switch, and so on. A component failure causes a service failure if it theoretically prevents an end user from accessing the service or a part of the service (even if the user receives a reasonable error message) or if it significantly degrades a user-visible aspect of system performance. We say "theoretically" because we infer a component failure's impact from various aspects of the problem report, such as the components that failed and any customer complaint reports, combined with knowledge of the service architecture.

We studied 18 service failures from Online (based on 85 component failures), 20 from Content (based on 99 component failures), and 21 from ReadMostly. The problems corresponded to the service failures during four months at Online, one month at Content, and six months at ReadMostly. A detailed analysis of this data appears elsewhere; [7] here we highlight a few data attributes from Content and ReadMostly.

We attributed a service failure to the first component that failed in the chain of events leading to the failure. We categorized this root-cause component by location (front end, back end, network, unknown) and type (node hardware, node software, network hardware, network software, operator, environment, overload, and unknown). Front-end nodes are initially contacted by end user clients and, at Content, by client proxy nodes. front-end nodes do not store persistent data, but they may cache data. back-end nodes store persistent data. We include the "business logic" of traditional three-tier system terminology in our front-end definition because these services integrate their service logic with the code that receives and replies to user client requests.

Using Content data, we observe that the rate at which component failures turn into service failures depends on the reason for the original problem, as Figure 4 shows. We list 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).

Most network failures have no known cause because they involve Internet connections, problems between collocation facilities, or between customer proxy sites and collocation facilities. For all but the node operator case, 11 percent 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 existing high-availability techniques such as redundancy. Table 4 lists service failures by type.

Observations

Our analysis of the failure data so far leads us to several observations. First, operator error is the largest single root cause of failures for Content, and the second largest for ReadMostly. In general, failures arose when operators made changes to the system — for example, scaling or replacing hardware or reconfiguring, deploying, or upgrading software. Most operator errors occurred during normal maintenance, not while an operator was in the process of fixing a problem. Despite the huge contribution of operator error to service failures, developers almost completely overlook these errors when designing high-dependability systems and the tools to monitor and control them.

Second, existing high-availability techniques such as redundancy almost completely masked hardware and software failures in Content and ReadMostly. Networks, however, are a surprisingly significant cause of problems, largely because they often provide a single point of failure. ReadMostly, for example, was the only service 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" networks will actually share a physical link or switch fairly close (in terms of Internet routing topology) to the collocation site. And because most services use a single network operations center, they sometimes cannot see the network problems between collocation sites or between collocation sites and customers; these problems can be the root of mysterious performance problems or outages.

Finally, an overarching difficulty in diagnosing and fixing problems in Internet services relates to the fact that multiple administrative entities must coordinate their problem-solving efforts. These entities include the service's network operations staff, its software developers, the operators of the collocation facilities that the service uses, the network providers between the service and its collocation facilities, and sometimes the customers. Today this coordination is handled almost entirely manually via telephone calls or e-mails between operators.

Tools for determining the root cause of problems across administrative domains are rudimentary — traceroute, for example — and they generally cannot distinguish between types of problems, such as end-host failures and problems on the network segment where a node is located. Moreover, the administrative entity that owns the broken hardware or software controls the tools for repairing a problem, not the service that determines the problem exists or is most affected by it. This results in increased mean time to repair.

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

Conclusions

The research we describe here opens a number of avenues for future research for members of the dependability community. Studying additional services and problem reports would increase the statistical significance of our findings with respect to failure causes and service architectures. Studying additional metrics such as Mean Time to Repair and/or the number of users affected by a failure would provide a view of service unavailability that more accurately gauges user impact than does the number-of-outages-caused metric we use here. Finally, one might analyze failure data with an eye toward identifying the causes of correlated failure, to indicate where additional safeguards might be added to existing systems.

References

1. E. Brewer, "Lessons from Giant-Scale Services, IEEE Internet Computing, vol. 4, no. 4, July/Aug. 2001, pp. 46-55.
2. Microsoft TechNet, "Building Scalable Services," tech. report, www.microsoft.com/technet/treeview/default.asp?url=/TechNet/itsolutions/ecommerce/deploy/projplan/bss1.asp, 2001.
3. J. Gray, "Why Do Computers Stop and What Can Be Done About It?" *Proc. Symp. Reliability in Distributed Software and Database Systems*, IEEE CS Press, Los Alamitos, Calif., 1986, pp. 3-12.
4. D. Kuhn, "Sources of Failure in the Public Switched Telephone Network," *Computer*, vol. 30, no. 4, Apr. 1997, pp. 31-36.
5. B. Murphy and T. Gant, "Measuring System and Software Reliability Using an Automated Data Collection Process," *Quality and Reliability Eng. Int'l*, vol. 11, 1995, pp. 341-353.
6. J. Xu, Z. Kalbarczyk, and R. Iyer, "Networked Windows NT System Field Failure Data Analysis," *Proc. 1999 Pacific Rim Int'l Symp. Dependable Computing*, IEEE CS Press, Los Alamitos, Calif., 1999.
7. D. Oppenheimer, "Why Do Internet Services Fail, and What Can Be Done about It?" tech. report UCB-CSD-02-1185, Univ. of Calif. at Berkeley, May 2002.

David Oppenheimer is a PhD candidate in computer science at the University of California at Berkeley. He received a BSE degree in electrical engineering from Princeton University in 1996 and an MS degree in computer science from the University of California at Berkeley in 2002. His research interests include techniques to improve and evaluate the dependability of cluster and geographically distributed services.

David A. Patterson joined the faculty at the University of California at Berkeley in 1977, where he now holds the Pardee Chair of Computer Science. He is a member of the National Academy of Engineering and is a fellow of both the ACM and the IEEE.

Readers can contact the authors at {davidopp,patterson}@cs.berkeley.edu.