CHAPTER 7

Dealing with Failures and Repairs

The promise of web-based, service-oriented computing will be fully realized only if users can trust that the services they increasingly rely on will be always available. This expectation translates into a high-reliability requirement for building-sized computers. Determining the appropriate level of reliability is fundamentally a tradeoff between the cost of failures (including repairs) and the cost of preventing them. For traditional enterprise-class servers, the cost of failures is thought to be high, and thus designers go to great lengths to provide more reliable hardware by adding redundant power supplies, fans, error correction coding (ECC), RAID disks, and so on. Many legacy enterprise applications were not designed to survive frequent hardware faults, and it is hard to make them fault-tolerant after the fact. Under these circumstances, making the hardware very reliable becomes a justifiable alternative.

In WSCs, however, hardware reliability alone cannot deliver sufficient availability primarily due to its scale. Suppose that a cluster has ultra-reliable server nodes with a stellar mean time between failures (MTBF) of 30 years (10,000 days)—well beyond what is typically possible to achieve at a realistic cost. Even with these ideally reliable servers, a cluster of 10,000 servers will see an average of one server failure per day. Thus, any application that depends on the availability of the entire cluster will see an MTBF no better than one day. In reality, typical servers see an MTBF substantially less than 30 years, and thus the real-life cluster MTBF would be in the range of a few hours between failures. Moreover, large and complex internet services are often composed of several software modules or layers that are not bug-free and can themselves fail at even higher rates than hardware components. Consequently, WSC applications must work around failed servers in software, either with code in the application itself or via functionality provided by middleware, such as a provisioning system for virtual machines that restarts a failed VM on a spare node. Some of the implications of writing software for this environment are discussed by Hamilton [Ham07], based on experience on designing and operating some of the largest services at MSN and Windows Live.

Before we continue, it's important to understand the difference between availability, unavailability, and failure. A system's availability is the fraction of time during which it is available for use; conversely, its unavailability is the fraction of time during which the system isn't available for some reason. Failures are one cause of unavailability, but are often much less common than other causes such as planned maintenance for hardware or software upgrades. Thus, a system with zero failures may still have availability of less than 100%, and a system with a high failure rate may have better availability than one with low failures, if other sources of unavailability dominate.

7.1 IMPLICATIONS OF SOFTWARE FAULT TOLERANCE

Fault-tolerant software is inherently more complex than software that can assume fault-free operation. As much as possible, we should try to implement a fault-tolerant software infrastructure layer that can hide much of the complexity of dealing with failures from application-level software.

There are some positive consequences of adopting a fault-tolerant model, though. Once hardware faults can be tolerated without undue disruption to a service, computer architects have some leeway to choose the level of hardware reliability that maximizes overall system cost efficiency. This leeway enables consideration, for instance, of using inexpensive PC-class hardware for a server platform instead of mainframe-class computers, as discussed in Chapter 3. In addition, this model can lead to simplifications in common operational procedures. For example, to upgrade system software in a cluster, you can load a newer version in the background (that is, during normal operation), kill the older version, and immediately start the newer one. Hardware upgrades can follow a similar procedure. Basically, the same fault-tolerant software infrastructure mechanisms built to handle server failures could have all the required mechanisms to support a broad class of operational procedures. By choosing opportune time windows and rate-limiting the pace of kill–restart actions, operators can still manage the desired number of planned service-level disruptions.

The basic property being exploited here is that, unlike in traditional server setups, it is no longer necessary to keep a server running at all costs. This simple requirement shift affects almost every aspect of the deployment, from machine and data center design to operations, often enabling optimization opportunities that would not be on the table otherwise. For instance, let us examine how this affects the recovery model. A system that needs to be highly reliable in the presence of unavoidable transient hardware faults, such as uncorrectable errors caused by cosmic particle strikes, may require hardware support for checkpoint recovery so that upon detection the execution can be restarted from an earlier correct state. A system that is allowed to go down upon occurrence of such faults may choose not to incur the extra overhead in cost or energy of checkpointing.

Another useful example involves the design tradeoffs for a reliable storage system. One alternative is to build highly reliable storage nodes through the use of multiple disk drives in a mirrored or RAIDed configuration so that a number of disk errors can be corrected on the fly. Drive redundancy increases reliability but by itself does not guarantee that the storage server will be always up. Many other single points of failure also need to be attacked (such as power supplies and operating system software), and dealing with all of them incurs extra cost while never assuring fault-free operation. Alternatively, data can be mirrored or RAIDed across disk drives that reside in multiple machines—the approach chosen by Google's GFS or Colossus file systems [GGL03]. This option tolerates not only drive failures but also entire storage server crashes because other replicas of each piece of data are accessible through other servers. It also has different performance characteristics from the centralized storage server scenario. Data updates may incur higher networking overheads because they require communicating with multiple systems to update all replicas, but aggregate read bandwidth can be greatly increased because clients can source data from multiple endpoints (in the case of full replication).

In a system that can tolerate a number of failures at the software level, the minimum requirement made to the hardware layer is that its faults are always detected and reported to software in a timely enough manner as to allow the software infrastructure to contain it and take appropriate recovery actions. It is not necessarily required that hardware transparently correct all faults. This does not mean that hardware for such systems should be designed without error correction capabilities. Whenever error correction functionality can be offered *within a reasonable cost or complexity*, it often pays to support it. It means that if hardware error correction would be exceedingly expensive, the system would have the option of using a less expensive version that provided *detection* capabilities only. Modern DRAM systems are a good example of a case in which powerful error *correction* can be provided at a low additional cost.

Relaxing the requirement that hardware errors be detected, however, would be much more difficult because every software component would be burdened with the need to check its own correct execution. At one early point in its history, Google had to deal with servers whose DRAM lacked even parity checking. Producing a web search index consists essentially of a very large shuffle/merge sort operation, using several machines over a long period. In 2000, one of the then monthly updates to Google's web index failed pre-release checks when a subset of tested queries was found to return seemingly random documents. After some investigation a pattern was found in the new index files that corresponded to a bit being stuck at zero at a consistent place in the data structures; a bad side effect of streaming a lot of data through a faulty DRAM chip. Consistency checks were added to the index data structures to minimize the likelihood of this problem recurring, and no further problems of this nature were reported. Note, however, that this workaround did not guarantee 100% error detection in the indexing pass because not all memory positions were being checked—instructions, for example, were not. It worked because index data structures were so much larger than all other data involved in the computation that having those self-checking data structures made it very likely that machines with defective DRAM would be identified and excluded from the cluster. The next machine generation at Google did include memory parity detection, and once the price of memory with ECC dropped to competitive levels, all subsequent generations have used ECC DRAM.

7.2 CATEGORIZING FAULTS

An efficient fault-tolerant software layer must be based on a set of expectations regarding fault sources, their statistical characteristics, and the corresponding recovery behavior. Software developed in the absence of such expectations can suffer from two risks: being prone to outages if the underlying faults are underestimated, or requiring excessive overprovisioning if faults are assumed to be much more frequent than they actually are.

Providing an accurate quantitative assessment of faults in WSC systems is challenging given the diversity of equipment and software infrastructure across different deployments. Instead, we will attempt to summarize the high-level trends from publicly available sources and from our own experience.

7.2.1 FAULT SEVERITY

Hardware or software faults can affect internet services in varying degrees, resulting in different service-level failure modes. The most severe modes may demand high reliability levels, whereas the least damaging modes might have more relaxed requirements that can be achieved with less expensive solutions. We broadly classify service-level failures into the following categories, listed in decreasing degree of severity.

- *Corrupted*: Committed data are impossible to regenerate, lost, or corrupted.
- *Unreachable*: Service is down or otherwise unreachable by users.
- *Degraded*: Service is available but in some degraded mode.
- *Masked*: Faults occur but are completely hidden from users by fault-tolerant software and hardware mechanisms.

Acceptable levels of robustness will differ across those categories. We expect most faults to be masked by a well-designed fault-tolerant infrastructure so that they are effectively invisible outside of the service provider. It is possible that masked faults will impact the service's maximum sustainable throughput capacity, but a careful degree of overprovisioning can ensure that the service remains healthy.

If faults cannot be completely masked, their least severe manifestation is one in which there is some degradation in the quality of service. Here, different services can introduce degraded availability in different ways. One example of such degraded service is when a web search system uses data partitioning techniques to improve throughput but loses some systems that serve parts of the database [Bre01]. Search query results will be imperfect but probably still acceptable in many cases. Graceful degradation as a result of faults can also manifest itself as decreased freshness. For example, a user may access his or her email account, but new email delivery is delayed by a few minutes, or some fragments of the mailbox could be temporarily missing. Although these kinds of faults also need to be minimized, they are less severe than complete unavailability. Internet services need to be deliberately designed to take advantage of such opportunities for gracefully degraded service. In other words, this support is often application-specific and not easily hidden within layers of cluster infrastructure software.

Service availability and reachability are very important, especially because internet service revenue is often related in some way to traffic volume [Cha+01b]. However, perfect availability is not a realistic goal for internet-connected services because the internet itself has limited availability characteristics. Chandra et al. [Cha+01b] report that internet endpoints may be unable to reach each other between 1% and 2% of the time due to a variety of connectivity problems, including routing issues. That translates to an availability of less than "two nines." In other words, even if an internet service is perfectly reliable, users will, on average, perceive it as being no greater than 99.0% available. As a result, an internet-connected service that avoids long-lasting outages for any large group of users and has an average unavailability of less than 1% will be difficult to distinguish from a perfectly reliable system. Google measurements of internet availability as of 2014 indicated that it was likely on average in the range of 99.6–99.9% when Google servers are one of the endpoints, but the spectrum is fairly wide. Some areas of the world experience significantly lower availability.

Measuring service availability in absolute time is less useful for internet services that typically see large daily, weekly, and seasonal traffic variations. A more appropriate availability metric is the fraction of requests satisfied by the service divided by the total number of requests made by users; a metric called *yield* by Brewer [Bre01].

Finally, one particularly damaging class of failures is the loss or corruption of committed updates to critical data, particularly user data, critical operational logs, or relevant data that are hard or impossible to regenerate. Arguably, it is much more critical for services not to lose data than to be perfectly available to all users. It can also be argued that such critical data may correspond to a relatively small fraction of all the data involved in a given service operation. For example, copies of the web and their corresponding index files are voluminous and important data for a search engine, but can ultimately be regenerated by recrawling the lost partition and recomputing the index files.

In summary, near perfect reliability is not universally required in internet services. Although it is desirable to achieve it for faults such as critical data corruption, most other failure modes can tolerate lower reliability characteristics. Because the internet itself has imperfect availability, a user may be unable to perceive the differences in quality of service between a perfectly available service and one with, say, four 9s (99.99%) of availability.

7.2.2 CAUSES OF SERVICE-LEVEL FAULTS

In WSCs, it is useful to understand faults in terms of their likelihood of affecting the health of the whole system, such as causing outages or other serious service-level disruption. Oppenheimer et al. [OGP03] studied three internet services, each consisting of more than 500 servers, and tried to identify the most common sources of service-level failures. They conclude that operator-caused or misconfiguration errors are the largest contributors to service-level failures, with hardware-related faults (server or networking) contributing to 10–25% of the total failure events.

Oppenheimer's results are somewhat consistent with the seminal work by Gray [Gra90], which doesn't look at internet services but instead examines field data from the highly fault-tolerant Tandem servers between 1985 and 1990. He also finds that hardware faults are responsible for a small fraction of total outages (less than 10%). Software faults (~60%) and maintenance/operations faults (~20%) dominate the outage statistics.

It is surprising at first to see hardware faults contributing to so few outage events in these two widely different systems. Rather than making a statement about the underlying reliability of the hardware components in these systems, such numbers indicate how successful the fault-tolerant techniques have been in preventing component failures from affecting high-level system behavior. In Tandem's case, such techniques were largely implemented in hardware, whereas in the systems Oppenheimer studied, we can attribute it to the quality of the fault-tolerant software infrastructure. Whether software- or hardware-based, fault-tolerant techniques do particularly well when faults are largely statistically independent, which is often (even if not always) the case in hardware faults. Arguably, one important reason why software-, operator-, and maintenance-induced faults have a high impact on outages is because they are more likely to affect multiple systems at once, thus creating a correlated failure scenario that is much more difficult to overcome.

Our experience at Google is generally in line with Oppenheimer's classification, even if the category definitions are not fully consistent. Figure 7.1 represents a rough classification of all events that corresponded to noticeable disruptions at the service level in one of Google's large-scale online services. These are not necessarily outages (in fact, most of them are not even user-visible events), but correspond to situations where some kind of service degradation is noticed by the monitoring infrastructure and must be scrutinized by the operations team. As expected, the service is less likely to be disrupted by machines or networking faults than by software errors, faulty configuration data, and human mistakes.

Factors other than hardware equipment failure dominate service-level disruption because it is easier to architect services to tolerate known hardware failure patterns than to be resilient to general software bugs or operator mistakes. A study by Jiang et al. [Jia+08], based on data from over 39,000 storage systems, concludes that disk failures are in fact not a dominant contributor to storage system failures. That result is consistent with analysis by Ford et al. [For+10] of distributed storage systems availability. In that study, conducted using data from Google's Colossus distributed file system, planned storage node reboot events are the dominant source of node-level unavailability. That study also highlights the importance of understanding correlated failures (failures in multiple storage nodes within short time windows), as models that don't account for correlation can underestimate the impact of node failures by many orders of magnitude.

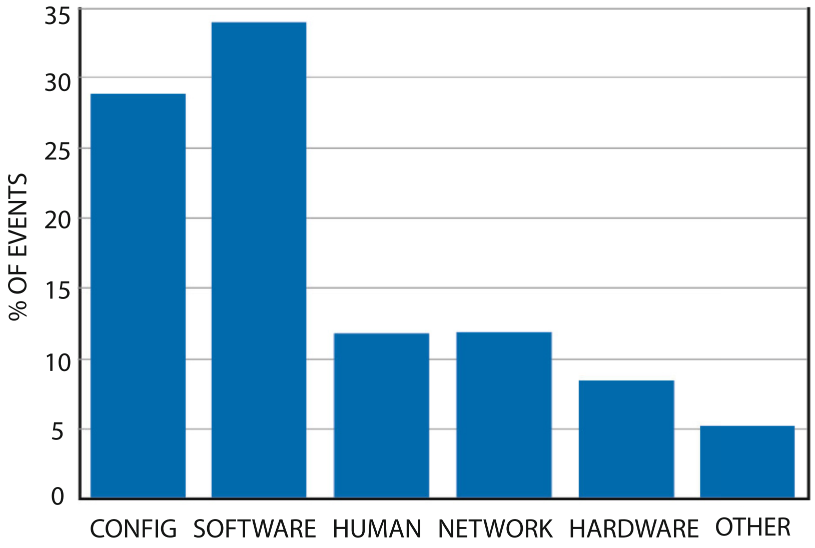

Figure 7.1: Distribution of service disruption events by most likely cause at one of Google's main services, collected over a period of six weeks by Google's Robert Stroud.

7.3 MACHINE-LEVEL FAILURES

An important factor in designing fault-tolerant distributed systems is understanding availability at the server level. Here we consider machine-level failures to be any situation that leads to a server being down, whatever the cause (such as operating system bugs).

As with cluster-service failures, relatively little published field data exists on server availability. A 1999 study by Kalyanakrishnam et al. [KKI99] finds that Windows NT machines involved in a mail routing service for a commercial organization were on average 99% available. The authors observed 1,100 reboot events across 66 servers and saw an average uptime of 11.82 days (median of 5.54 days) and an average downtime of just less than 2 hr (median of 11.43 min). About half of the reboots were classified as abnormal; that is, were due to a system problem instead of a normal shutdown. Only 10% of the reboots could be blamed on faulty hardware or firmware. The data suggest that application faults, connectivity problems, or other system software failures are the largest known crash culprits. If we are interested only in the reboot events classified as abnormal, we arrive at an MTTF of approximately 22 days, or an annualized machine failure rate of more than 1,600%.

Schroeder and Gibson [SG07a] studied failure statistics from high-performance computing systems at Los Alamos National Laboratory. Although these are not a class of computers that we are interested in here, they are made up of nodes that resemble individual servers in WSCs, so their data is relevant in understanding machine-level failures in our context. Their analysis spans nearly 24,000 processors, with more than 60% of them deployed in clusters of small-scale SMPs (2–4 processors per node). Although the node failure rates vary by more than a factor of 10x across different systems, the failure rate normalized by number of processors is much more stable—approximately 0.3 faults per year per CPU—suggesting a linear relationship between the number of sockets and unreliability. If we assume servers with four CPUs, we could expect machine-level failures to be at a rate of approximately 1.2 faults per year or an MTTF of approximately 10 months. This rate of server failures is more than 14 times lower than the one observed in Kalyanakrishnan's study [KKI99].

Google's machine-level failure and downtime statistics are summarized in Figures 7.2 and 7.3. The data is based on a six-month observation of all machine restart events and their corresponding downtime, where downtime corresponds to the entire time interval where a machine is not available for service, regardless of cause. These statistics cover all of Google's machines. For example, they include machines that are in the repairs pipeline, planned downtime for upgrades, as well as all kinds of machine crashes.

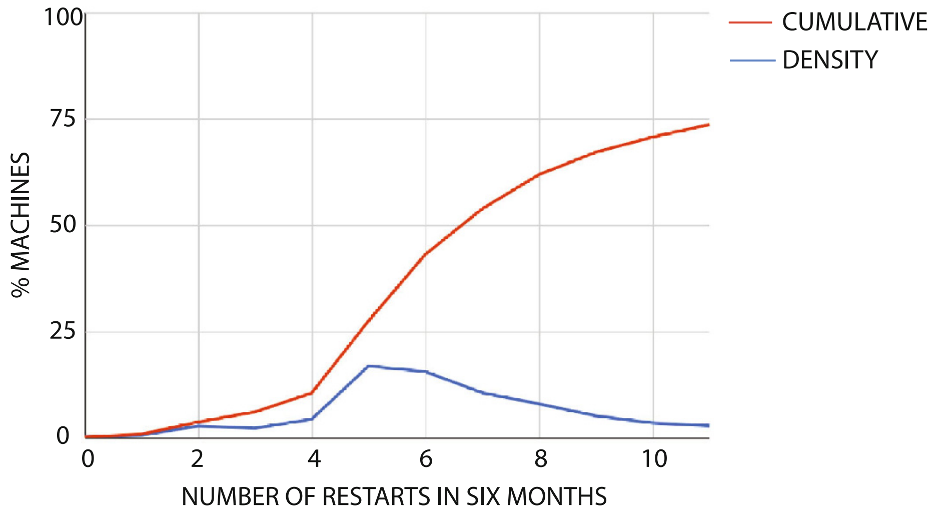

Figure 7.2: Distributions of machine restarts over six months at Google. (Updated in 2018.)

Figure 7.2 shows the distribution of machine restart events. The graph shows that 50% or more machines restart at least once a month, on average. The tail is relatively long (the figure truncates the data at 11 or more restarts) due to the large population of machines in Google's fleet. Approximately 5% of all machines restart more than once a week. Several effects, however, are smudged away by such large-scale averaging. For example, we typically see higher than normal failure rates during the first few months of new server product introduction. The causes include manufacturing bootstrapping effects, firmware and kernel bugs, and occasional hardware problems that become noticeable only after a large number of systems are in use. If we exclude from the sample all machines that are still suffering from such effects, the annualized restart rate corresponds to approximately one month between restarts, on average, largely driven by Google's bug resolution, feature enhancements, and security-processes-related upgrades. We also note that machines with frequent restarts are less likely to be in active service for long.

In another internal study conducted recently, planned server unavailability events were factored out from the total machine unavailability. The remaining unavailability, mainly from server crashes or lack of networking reachability, indicates that a server can stay up for an average of nearly two years (0.5 unplanned restart rate). This is consistent with our intuition that most restarts are due to planned events, such as software and hardware upgrades.

These upgrades are necessary to keep up with the velocity of kernel changes and also allows Google to prudently react to emergent & urgent security issues. As discussed earlier, it is also important to note that Google Cloud's Compute Engine offers live migration to keep the VM instances running by migrating the running instances to another host in the same zone rather than requiring your VMs to be rebooted. Note that live migration does not change any attributes or properties of the VM itself.

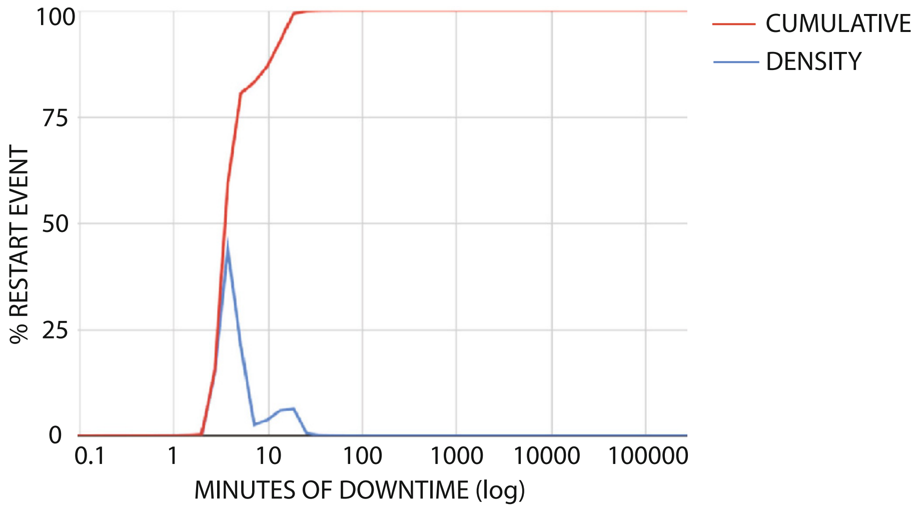

Figure 7.3: Distribution of machine downtime, observed at Google over six months. The average annualized restart rate across all machines is 12.4, corresponding to a mean time between restarts of just less than one month.

Restart statistics are key parameters in the design of fault-tolerant software systems, but the availability picture is complete only once we combine it with downtime data—a point articulated earlier by the Berkeley ROC project [Pat+02]. Figure 7.3 shows the distribution of downtime from the same population of Google servers. The x-axis displays downtime, against both density and cumulative machine distributions. Note that the data include both planned reboots and those caused by miscellaneous hardware and software failures. Downtime includes all the time a machine stopped operating until it has rebooted and basic node services were restarted. In other words, the downtime interval ends not when the machine finishes rebooting, but when key basic daemons are up.

Approximately 55% of all restart events last less than 3 min, 25% of them last between 3 and 30 min, with most of the remaining restarts finishing in about a day. These usually correspond to some combination of physical repairs (SW directed swaps) and automated file system recovery from crashes. Approximately 1% of all restart events last more than a day, usually corresponding to systems going into repairs. The average downtime is just more than 10 min. The resulting average machine availability is 99.93%.

When provisioning fault-tolerant software systems, it is also important to focus on real (unexpected) machine crashes, as opposed to the previous analysis that considers all restarts. In our experience, the crash rate of mature servers (those that survived infant mortality) ranges between 1.2 and 2 crashes per year. In practice, this means that a service that uses 2,000 servers should plan to operate normally while tolerating a machine crash approximately every 2.5 hr, or approximately 10 machines per day. Given the expected machine downtime for 99% of all restart cases is less than 2 days, one would need as few as 20 spare machines to safely keep the service fully provisioned. A larger margin might be desirable if there is a large amount of state that must be loaded before a machine is ready for service.

7.3.1 WHAT CAUSES MACHINE CRASHES?

Reliably identifying culprits for machine crashes is generally difficult because, in many situations, transient hardware errors can be hard to distinguish from operating system or firmware bugs. However, there is significant indirect and anecdotal evidence suggesting that software-induced crashes are much more common than those triggered by hardware faults. Some of this evidence comes from component-level diagnostics. Because memory and disk subsystem faults were the two most common diagnostics for servers sent to hardware repairs within Google in 2018, we will focus on those.

DRAM Soft Errors

Although there are little available field data on this topic, it is generally believed that DRAM soft error rates are extremely low once modern ECCs are used. In a 1997 IBM white paper, Dell [Del97] sees error rates from chipkill ECC being as low as six errors for 10,000 one-GB systems over three years (0.0002 errors per GB per year—an extremely low rate). A survey article by Tezzaron Semiconductor in 2004 [Terra] concludes that single-error rates per Mbit in modern memory devices range between 1,000 and 5,000 FITs (failures in time, defined as the rate of faults per billion operating hours), but that the use of ECC can drop soft-error rates to a level comparable to that of hard errors.

A study by Schroeder et al. [SPW09] evaluated DRAM errors for the population of servers at Google and found FIT rates substantially higher than previously reported (between 25,000 and 75,000) across multiple DIMM technologies. That translates into correctable memory errors affecting about a third of Google machines per year and an average of one correctable error per server every 2.5 hr. Because of ECC technology, however, only about 1.3% of all machines ever experience uncorrectable memory errors per year. A more recent study [HSS12] found that a large fraction of DRAM errors could be attributed to hard (non-transient) errors and suggested that simple page retirement policies could mask a large fraction of DRAM errors in production systems while sacrificing only a negligible fraction of the total DRAM in the system.

Disk Errors

Studies based on data from NetApp and the University of Wisconsin [Bai+07], Carnegie Mellon [SG07b], and Google [PWB07] have recently examined the failure characteristics of modern disk drives. Hard failure rates for disk drives (measured as the annualized rate of replaced components) have typically ranged between 2% and 4% in large field studies, a much larger number than the usual manufacturer specification of 1% or less. Bairavasundaram et al. [Bai+07] looked specifically at the rate of latent sector errors—a measure of data corruption frequency. In a population of more than 1.5 million drives, they observed that less than 3.5% of all drives develop any errors over a period of 32 months.

These numbers suggest that the average fraction of machines crashing annually due to disk or memory subsystem faults should be less than 10% of all machines. Instead, we observe crashes to be more frequent and more widely distributed across the machine population. We also see noticeable variations on crash rates within homogeneous machine populations that are more likely explained by firmware and kernel differences.

The effect of ambient temperature on the reliability of disk drives has been well studied by Pinheiro et al. [PWB07] and El Sayed et al. [ES+]. While common wisdom previously held that temperature had an exponentially negative effect on the failure rates of disk drives, both of these field studies found little or no evidence of that in practice. In fact, both studies suggest that most disk errors appear to be uncorrelated with temperature.

Another indirect evidence of the prevalence of software-induced crashes is the relatively high mean time to hardware repair observed in Google's fleet (more than six years) when compared to the mean time to machine crash (six months or less).

It is important to mention that a key feature of well-designed fault-tolerant software is its ability to survive individual faults, whether they are caused by hardware or software errors. One architectural option that can improve system reliability in the face of disk drive errors is the trend toward diskless servers; once disks are a networked resource it is easier for a server to continue operating by failing over to other disk devices in a WSC.

7.3.2 PREDICTING FAULTS

The ability to predict future machine or component failures is highly valued because it could avoid the potential disruptions of unplanned outages. Clearly, models that can predict most instances of a given class of faults with very low false-positive rates can be very useful, especially when those predictions involve short time-horizons—predicting that a memory module will fail within the next 10 years with 100% accuracy is not particularly useful from an operational standpoint.

When prediction accuracies are less than perfect, which unfortunately tends to be true in most cases, the model's success will depend on the tradeoff between accuracy (both in false-positive rates and time horizon) and the penalties involved in allowing faults to happen and recovering from them. Note that a false component failure prediction incurs all of the overhead of the regular hardware repair process (parts, technician time, machine downtime, etc.). Because software in WSCs is designed to gracefully handle all the most common failure scenarios, the penalties of letting faults happen are relatively low; therefore, prediction models must have much greater accuracy to be economically competitive. By contrast, traditional computer systems in which a machine crash can be very disruptive to the operation may benefit from less accurate prediction models.

Pinheiro et al. [PWB07] describe one of Google's attempts to create predictive models for disk drive failures based on disk health parameters available through the Self-Monitoring Analysis and Reporting Technology (SMART) standard. They conclude that such models are unlikely to predict individual drive failures with sufficient accuracy, but they can be useful in reasoning about the expected lifetime of groups of devices which can be useful in optimizing the provisioning of replacement units.

7.4 REPAIRS

An efficient repair process is critical to the overall cost efficiency of WSCs. A machine in repair is effectively out of operation, so the longer a machine is in repair, the lower the overall availability of the fleet. Also, repair actions are costly in terms of both replacement parts and the skilled labor involved. Last, repair quality—how likely a repair action will actually fix a problem while accurately determining which (if any) component is at fault—affects both component expenses and average machine reliability.

Two characteristics of WSCs directly affect repair efficiency. First, because of the high number of relatively low-end servers involved and the presence of a software fault-tolerance layer, quickly responding to individual repair cases is not as critical because the repairs are unlikely to affect overall service health. Instead, a data center can implement a schedule that makes the most efficient use of a technician's time by making a daily sweep of all machines that need repairs attention. The philosophy is to increase the rate of repairs while keeping the repair latency within acceptable levels.

In addition, when many thousands of machines are in operation, massive volumes of data about machine health can be collected and analyzed to create automated systems for health determination and diagnosis. Google's system health infrastructure, illustrated in Figure 7.4, is an example of a monitoring system that takes advantage of this massive data source. It constantly monitors every server for configuration, activity, environmental, and error data. This information is stored as a time series in a scalable repository where it can be used for various kinds of analysis, including an automated machine failure diagnostics tool that uses machine learning methods to suggest the most appropriate repairs action.

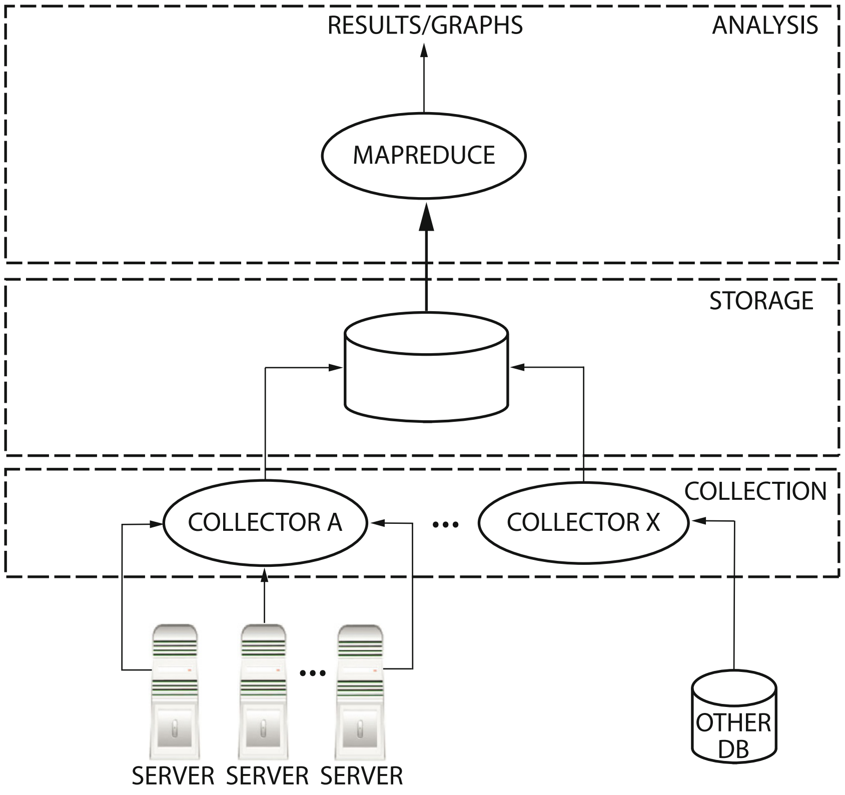

Figure 7.4: Google's system health monitoring and analysis infrastructure.

In addition to performing individual machine diagnostics, the system health infrastructure is useful in other ways. For example, it monitors the stability of new system software versions and has helped pinpoint a specific batch of defective components across a large fleet of machines. It has also been valuable in longer-term analytical studies, such as the disk failure study by Pinheiro et al., mentioned in the previous section, and the data center-scale power provisioning study by Fan et al. [FWB07].

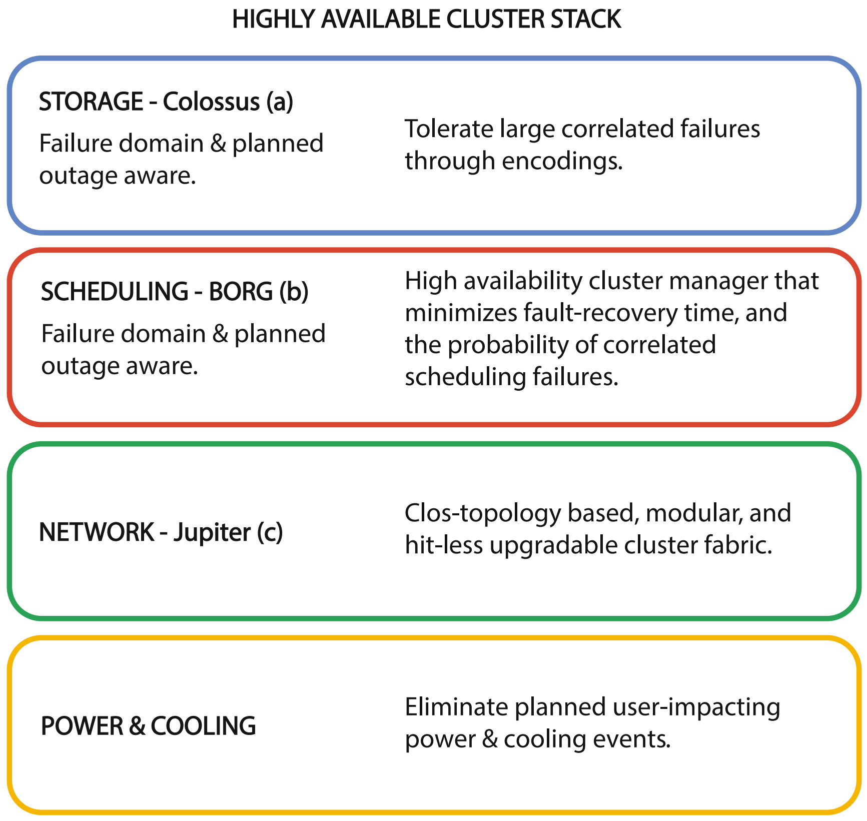
HIGHLY AVAILABLE CLUSTER STACK
STORAGE - Colossus (a)
Failure domain & planned outage aware.
Tolerate large correlated failures through encodings.
SCHEDULING - BORG (b)
Failure domain & planned outage aware.
High availability cluster manager that minimizes fault-recovery time, and the probability of correlated scheduling failures.
NETWORK - Jupiter (c)
Clos-topology based, modular, and hit-less upgradable cluster fabric.
POWER & COOLING
Eliminate planned user-impacting power & cooling events.

Figure 7.5: Highly available cluster architecture [Ser17, Ver+15, Sin+15].

7.5 TOLERATING FAULTS, NOT HIDING THEM

The capacity of well-designed fault-tolerant software to mask large numbers of failures with relatively little impact to service-level metrics could have unexpectedly dangerous side effects. Consider a three-tier application representing a web service with the backend tier replicated three times. Such replicated setups have the dual purpose of increasing peak throughput as well as tolerating server faults when operating below peak capacity. Assume that the incoming request rate is at 50% of total capacity. At this level, this setup could survive one backend failure with little disruption in service levels. However, a second backend failure would have a dramatic service-level impact that could theoretically result in a complete outage.

In systems descriptions, we often use N to denote the number of servers required to provide a service at full load, so N + 1 describes a system with one additional replica for redundancy. Common arrangements include N (no redundancy), N + 1 (tolerating a single failure), N + 2 ("concurrently maintainable," tolerating a single failure even when one unit is offline for planned maintenance), and 2N (mirroring of every unit).

Systems with large amounts of internal replication to increase capacity (horizontal scaling) provide redundancy at a very low cost. For example, if we need 100 replicas to handle the daily peak load, an N + 2 setup incurs just 2% overhead for high availability. Such systems can tolerate failures so well that an outside observer might be unaware of how much internal slack remains, or in other words, how close to the edge one might be. In those cases, the transition from healthy behavior to meltdown can be abrupt, which is not a desirable property. This example emphasizes the importance of comprehensive monitoring, both at the application (or service) level as well as the machine infrastructure level, so that faults can be well tolerated and yet visible to operators. This enables prompt corrective action when the amount of internal redundancy approaches the limits of what the fault-tolerant software layer can handle.

Still, broken machines eventually must be repaired. Traditional scenarios, where a repair must happen immediately, require more costly staging of replacement parts as well as additional costs to bringing a service technician on site. When we can batch repairs, we can lower these costs per repair. For reference, service contracts for IT equipment that provide on-site repair within 24 hr typically come at an annual cost of 5–15% of the equipment's value; a 4-hr response time usually doubles that cost. In comparison, repairs in large server farms are cheaper. To illustrate the point, assume that a WSC has enough scale to keep a full-time repairs technician busy. Assuming 1 hr per repair and an annual failure rate of 5%, a system with 40,000 servers would suffice; in reality, that number will be considerably smaller because the same technician can also handle installations and upgrades. Let us further assume that the hourly cost of a technician is $100 and that the average repair requires replacement parts costing 10% of the system cost; both of these assumptions are generously high. Still, for a cluster of servers costing $2,000 each, we arrive at an annual cost per server of 5% ∗ ($100 + 10% ∗ $2,000) = $15, or 0.75% per year. In other words, keeping large clusters healthy can be quite affordable.

7.6 ACCOUNTING FOR FAULTS IN CLUSTER SYSTEM DESIGN

For services with mutable state, as previously noted, replicas in different clusters must now worry about consistent replicas of highly mutable data. However, most new services start serving with a small amount of traffic and are managed by small engineering teams. In such cases these services need only a small fraction of the capacity of a given cluster, but if successful, will grow over time. For these services, even if they do not have highly mutable state, managing multiple clusters imposes a significant overhead, both in terms of engineering time and resource cost; that is, N + 2 replication is expensive when N = 1. Furthermore, these services tend to be crowded out of clusters by the large services and by the fact that a large number of services leads to resource fragmentation. In all likelihood, the free resources are not where the service is currently running.

To address these problems, the system cluster design needs to include a unit of resource allocation and job schedulability with certain availability, and eliminate planned outages as a source of overhead for all but singly-homed services with the highest availability requirements.

Figure 7.5 shows a schematic of such a cluster stack.

Power: At the base of the stack, we tackle the delivery of power and cooling to machines. Typically, we target a fault tolerant and concurrently maintainable physical architecture for the power and cooling system. For example, as described in Chapter 4, power is distributed hierarchically at the granularity of the building and physical data center rows. For high availability, cluster scheduling purposely spreads jobs across the units of failure. Similarly, the required redundancy in storage systems is in part determined by the fraction of a cluster that may simultaneously fail as a result of a power event. Hence, larger clusters lead to lower storage overhead and more efficient job scheduling while meeting diversity requirements.

Networking: A typical cluster fabric architecture, like the Jupiter design described in Chapter 3, achieves high availability by redundancy in fabric and physical diversity in deployment. It also focuses on robust software for the necessary protocols and a reliable out-of-band control plane. Given the spreading requirement for the cluster scheduler as previously discussed, Jupiter also supports uniform bandwidth across the cluster and resiliency mechanisms in the network control plane in addition to data path diversity.

Scheduling: Applications that run on Borg (Google's large-scale cluster management system) are expected to handle scheduling-related failure events using techniques such as replication, storing persistent state in a distributed file system, and (if appropriate) taking occasional checkpoints. Even so, we try to mitigate the impact of these events. For example, Borg scheduling accounts for automatic rescheduling of evicted applications, on a new machine if necessary, and reduced correlated failures by spreading applications across failure domains such as machines, racks, and power domains.

Storage: We keep the overall storage system highly available with two simple yet effective strategies: fast recovery and replication/encoding. Details are listed in the reference [Ser17].

Together, these capabilities provide a highly available cluster suitable for a large number of big or small services. In cloud platforms such as Google Cloud Platform (GCP), high availability is exposed to customers using the concept of zones and regions [GCRZ]. A representative mapping of compute zones to regions is shown in Figure 7.6.

Google internally maintains a map of clusters to zones such that the appropriate product SLAs (Service Level Agreements) can be satisfied (for example, Google Compute Engine SLAs [GCE]). Cloud customers are encouraged to utilize the zone and region abstractions in developing highly available and fault-tolerant applications on GCP.

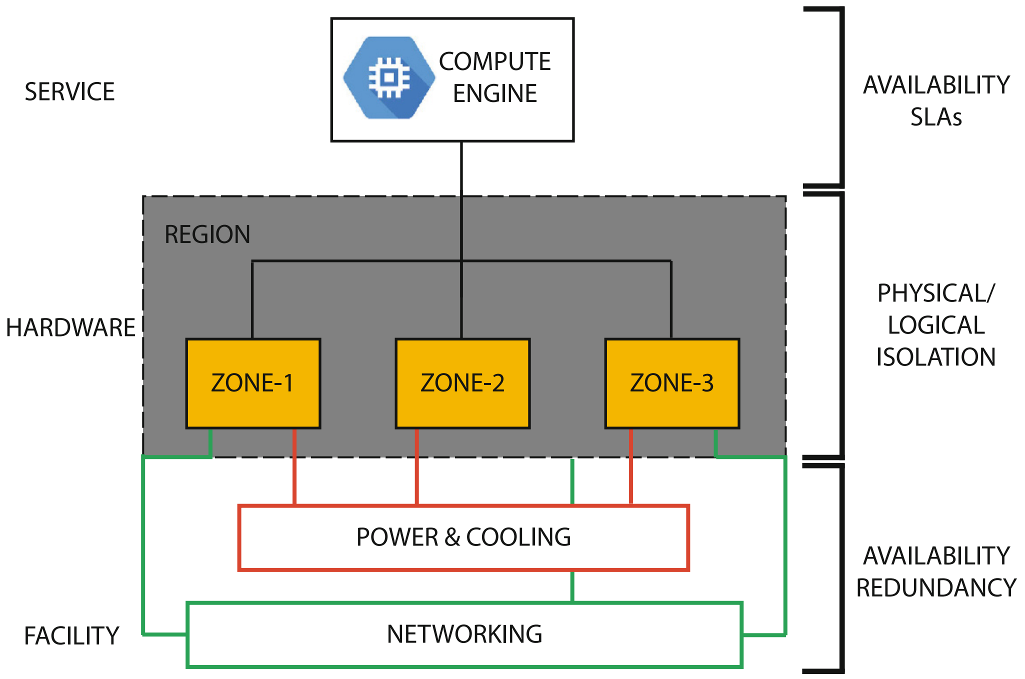

Figure 7.6: Mapping of Google Compute Engine zones to regions, with associated physical and logical isolation and customer facing availability properties.