Harvest, Yield, and Scalable Tolerant Systems

Armando Fox
Stanford University
fox@cs.stanford.edu

Eric A. Brewer
University of California at Berkeley
brewer@cs.berkeley.edu

Abstract

The cost of reconciling consistency and state management with high availability is highly magnified by the unprecedented scale and robustness requirements of today's Internet applications. We propose two strategies for improving overall availability using simple mechanisms that scale over large applications whose output behavior tolerates graceful degradation. We characterize this degradation in terms of harvest *and* yield*, and map it directly onto engineering mechanisms that enhance availability by improving fault isolation, and in some cases also simplify programming. By collecting examples of related techniques in the literature and illustrating the surprising range of applications that can benefit from these approaches, we hope to motivate a broader research program in this area.*

1. Motivation, Hypothesis, Relevance

Increasingly, infrastructure services comprise not only routing, but also application-level resources such as search engines [15], adaptation proxies [8], and Web caches [20]. These applications must confront the same 24×7 operational expectations and exponentially-growing user loads as the routing infrastructure, and consequently are absorbing comparable amounts of hardware and software. The current trend of harnessing commodity-PC clusters for scalability and availability [9] is reflected in the largest web server installations. These sites use tens to hundreds of PC's to deliver 100M or more read-mostly page views per day, primarily using simple replication or relatively small data sets to increase throughput.

The scale of these applications is bringing the well-known tradeoff between consistency and availability [4] into very sharp relief. In this paper we propose two general directions for future work in building large-scale robust systems. Our approaches tolerate partial failures by emphasizing simple composition mechanisms that promote fault containment, and by translating possible partial failure modes into engineering mechanisms that provide smoothly-degrading functionality rather than lack of availability of the service as a whole. The approaches were developed in the context of cluster computing, where it is well accepted [22] that one of the major challenges is the nontrivial software engineering required to automate partial-failure handling in order to keep system management tractable.

2. Related Work and the CAP Principle

In this discussion, *strong consistency* means single-copy ACID [13] consistency; by assumption a strongly-consistent system provides the ability to perform updates, otherwise discussing consistency is irrelevant. *High availability* is assumed to be provided through redundancy, e.g. data replication; data is considered highly available if a given consumer of the data can always reach *some* replica. *Partition-resilience* means that the system as whole can survive a partition between data replicas.

Strong CAP Principle. Strong **C**onsistency, High **A**vailability, **P**artition-resilience: Pick at most 2.

The CAP formulation makes explicit the trade-offs in designing distributed infrastructure applications. It is easy to identify examples of each pairing of CAP, outlining the proof by exhaustive example of the Strong CAP Principle:

- CA without P: Databases that provide distributed transactional semantics can only do so in the absence of a network partition separating server peers.
- CP without A: In the event of a partition, further transactions to an ACID database may be blocked until the partition heals, to avoid the risk of introducing merge conflicts (and thus inconsistency).
- AP without C: HTTP Web caching provides client-server partition resilience by replicating documents, but a client-server partition prevents verification of the freshness of an expired replica. In general, any distributed database problem can be solved with either expiration-based caching to get AP, or replicas and majority voting to get PC (the minority is unavailable).

In practice, many applications are best described in terms of *reduced* consistency or availability. For example, weakly-consistent distributed databases such as Bayou [5] provide specific models with well-defined consistency/availability tradeoffs; disconnected filesystems such as Coda [16] explicitly argued for availability over strong consistency; and expiration-based consistency mechanisms such as leases [12] provide fault-tolerant consistency management. These examples suggest that there is a **Weak CAP Principle** which we have yet to characterize precisely: The stronger the guarantees made about any two of strong consistency, high availability, or resilience to partitions, the weaker the guarantees that can be made about the third.

3. Harvest, Yield, and the CAP Principle

Both strategies we propose for improving availability with simple mechanisms rely on the ability to broaden our notion of "correct behavior" for the target application, and then exploit the tradeoffs in the CAP principle to improve availability at large scale.

We assume that clients make queries to servers, in which case there are at least two metrics for correct behavior: *yield*, which is the probability of completing a request, and *harvest*, which measures the fraction of the data reflected in the response, i.e. the completeness of the answer to the query. Yield is the common metric and is typically measured in "nines": "four-nines availability" means a completion probability of 0.9999. In practice, good HA systems aim for four or five nines. In the presence of faults there is typically a tradeoff between providing no answer (reducing yield) and providing an imperfect answer (maintaining yield, but reducing harvest). Some applications do not tolerate harvest degradation because any deviation from the single well-defined correct behavior renders the result useless. For example, a sensor application that must provide a binary sensor reading (presence/absence) does not tolerate degradation of the output.[1] On the other hand, some applications tolerate graceful degradation of harvest: online aggregation [14] allows a user to explicitly trade running time for precision and confidence in performing arithmetic aggregation queries over a large dataset, thereby smoothly trading harvest for response time, which is particularly useful for approximate answers and for avoiding work that looks unlikely to be worthwhile based on preliminary results.

At first glance, it would appear that this kind of degradation applies only to queries and not to updates. However, the model can be applied in the case of "single-location" updates: those changes that are localized to a single node (or technically a single partition). In this case, updates that affect reachable nodes occur correctly but have limited visibility (a form of reduced harvest), while those that require unreachable nodes fail (reducing yield). These localized changes are consistent exactly because the new values are not available everywhere. This model of updates fails for global changes, but it is still quite useful for many practical applications, including personalization databases and collaborative filtering.

[1] This is consistent with the use of the term *yield* in semiconductor manufacturing: typically, each die on a wafer is intolerant to harvest degradation, and yield is defined as the fraction of working dice on a wafer.

4. Strategy 1: Trading Harvest for Yield—Probabilistic Availability

Nearly all systems are probabilistic whether they realize it or not. In particular, any system that is 100% available under single faults is probabilistically available overall (since there is a non-zero probability of multiple failures), and Internet-based servers are dependent on the best-effort Internet for true availability. Therefore availability maps naturally to probabilistic approaches, and it is worth addressing probabilistic systems directly, so that we can understand and limit the impact of faults. This requires some basic decisions about what needs to be available and the expected nature of faults.

For example, node faults in the Inktomi search engine remove a proportional fraction of the search database. Thus in a 100-node cluster a single-node fault reduces the harvest by 1% during the duration of the fault (the overall harvest is usually measured over a longer interval). Implicit in this approach is graceful degradation under multiple node faults, specifically, linear degradation in harvest. By randomly placing data on nodes, we can ensure that the 1% lost is a random 1%, which makes the average-case and worst-case fault behavior the same. In addition, by replicating a high-priority subset of data, we reduce the probability of losing that data. This gives us more precise control of harvest, both increasing it and reducing the practical impact of missing data. Of course, it is possible to replicate all data, but doing so may have relatively little impact on harvest and yield despite significant cost, and in any case can never ensure 100% harvest or yield because of the best-effort Internet protocols the service relies on.

As a similar example, transformation proxies for thin clients [8] also trade harvest for yield, by degrading results on demand to match the capabilities of clients that might otherwise be unable to get results at all. Even when the 100%-harvest answer is useful to the client, it may still be preferable to trade response time for harvest when client-to-server bandwidth is limited, for example, by intelligent degradation to low-bandwidth formats [7].

5. Strategy 2: Application Decomposition and Orthogonal Mechanisms

Some large applications can be decomposed into subsystems that are independently intolerant to harvest degradation (i.e. they fail by reducing yield), but whose independent failure allows the overall application to continue functioning with reduced utility. The application as a whole is then tolerant of harvest degradation. A good decomposition has at least one actual benefit and one potential benefit.

The actual benefit is the ability to provision each subsystem's state management separately, providing strong consistency or persistent state only for the subsystems that need it, not for the entire application. The savings can be significant if only a few small subsystems require the extra complexity. For example, a typical e-commerce site has a read-only subsystem (user-profile-driven content generation from a static corpus), a transactional subsystem (billing), a subsystem that manages state that must be persistent over the course of a session but not thereafter (shopping cart), and a subsystem that manages truly persistent but read-mostly/write-rarely state (user personalization profile). Any of these subsystems, except possibly billing, can fail without rendering the whole service useless. If the user profile store fails, users may still browse merchandise but without the benefit of personalized presentation; if the shopping cart mechanism fails, one-at-a-time purchases are still possible; and so on.

Traditionally, the boundary between subsystems with differing state management requirements and data semantics has been characterized via narrow interface layers; we propose that in some cases it is possible to do even better, if we can identify *orthogonal mechanisms*. Unlike a layered mechanism, which sits above or below the next layer, an orthogonal mechanism is *independent* of other mechanisms, and has essentially *no runtime interface* to the other mechanisms (except possibly a configuration interface). Since Brooks [1] reveals that the complexity of a software project grows as the square of the number of engineers and Leveson [17] cites evidence that most failures in complex systems result from unexpected inter-component interaction rather than intra-component bugs, we conclude that less machinery is (quadratically) better. The ability to exploit orthogonal mechanisms therefore constitutes a second (potential) advantage of decomposition.

5.1. Programming With Orthogonal Mechanisms

Somewhat to our surprise, we have found that orthogonal mechanisms are not as limiting in practice as their description suggests. For example, the cluster-based Scalable Network Server (SNS) [9] is a deployed example of the orthogonal mechanisms approach. SNS is a software layer that provides high availability and incremental scaling on a cluster of PC's, but provides no persistent state management facilities or data consistency guarantees. In fact, a programming requirement for SNS-hosted applications, which are structured as composable subsystems as described above, is that each application module be restartable at essentially arbitrary times. Although this constraint is nontrivial, it allows SNS to use simple orthogonal mechanisms such as timeouts, retries, and sandboxing to automatically handle a variety of transient faults and load imbalances in the cluster and keep application modules available while doing a reasonable job of automatic load balancing.

Despite the restartability constraint and lack of state maintenance in SNS, we used it to deploy a group-state application: MediaPad [19], an adaptation proxy for the desktop *mediaboard* application that allows a PalmPilot to participate in a multi-user shared whiteboard session. MediaBoard and MediaPad use SRM (Scalable Reliable Multicast) [6] as the underlying communication protocol. In SRM applications, there are no hard copies of group or session state, but a soft copy is maintained by each peer in the session, and a multicast-based repair mechanism provides the basis for collaborative state maintenance. Crash recovery is based on refreshing the soft state via the repair mechanism. This behavior is compatible with the SNS constraint of restartable workers, and state maintenance is orthogonal to SNS, since no interfaces or behaviors were added or modified in SNS to support SRM applications. Similar techniques have been used to prototype a real-time streaming media server using soft-state protocol modules [23] running on SNS.

5.2. Related Uses of Orthogonal Mechanisms

Composition of orthogonal subsystems shifts the burden of checking for possibly harmful interactions from runtime to compile time, and deployment of orthogonal guard mechanisms improves robustness for the runtime interactions that do occur, by providing improved fault containment. The practical implication of these effects is that application writers need not concern themselves directly with the provision of incremental scaling (replication and load management) and high availability: the simple mechanisms in SNS perform these functions for all applications.

Neither use of orthogonality is new. Various forms of sandboxing, including stack-overrun guarding [3], system-call monitoring [11], and software fault isolation [24], constitute good examples of orthogonal safety. Orthogonal privacy and data integrity is exemplified by the Secure Socket Layer (SSL) protocol [10]: an initial out-of-band handshake establishes a secure channel, which can then be used as the substrate of any stream connection. Orthogonal approaches are particularly useful in adding operational fea-

tures such as security or robustness to legacy applications that were designed without these features in mind, without requiring special changes to the core application code. The safety-critical systems community has also been using orthogonal mechanisms for some time: the mechanical interlocks removed from the Therac-25 radiation therapy machine unmasked a software race condition that ultimately led to patient fatalities [18]. This and similar examples constitute abundant (if anecdotal) support for the design principle of simple failsafe mechanisms with small state spaces; our contribution is the identification of this collection of techniques and the potential synergy of pairing them with compile-time orthogonal composition as strategies for improving robustness.

6. Discussion and Research Agenda

We presented two implemented examples of mapping harvest degradation onto specific mechanisms that provide engineering tractability, availability through redundancy, or some other desired *operational* feature. In the case of the Inktomi search engine, per-node timeout constraints keep the overall system yield constant at the expense of probabilistic harvest degradation. In general, it maps faults to degradation in harvest rather than yield, thus providing probabilistically good answers essentially all the time (although the yield cannot be 100%). In the case of the SNS cluster-based application server, the constraint that application modules must be restartable allows the use of simple scaling and reliability mechanisms, including orthogonal mechanisms such as timeouts and retries, and the restartability constraint is addressed by composing the applications with orthogonal state maintenance mechanisms such as SRM. Specific mechanisms we have been able to exploit to simplify engineering and improve robustness or scalability include:

- Simple mechanisms with small state spaces whose behaviors are easy to reason about: timeout-based partial failure handling, guard timers, orthogonal security, etc., inspired by orthogonal mechanisms in safety-critical systems.

- The orthogonalization of these mechanisms with respect to application logic, separating the application functionality from the provision of high availability. The composition of SNS and SRM provide a good illustration of this approach.

- The replacement of hard state with refreshable soft state, which often has the beneficial side effect of making the recovery code the same as the mainline code. The load balancing manager in SNS works this way [2], using refreshable soft state mechanisms inspired by IP multicast routing and SRM state repair.

- Overall tractability of large-scale engineering involving hardware replication and redundancy. Only a few very expensive specialized systems, such as Teradata's 768-node data mining cluster [21], really compare in size and aggregate capacity to cluster-based Internet services.

It remains to formally characterize applications that tolerate graceful harvest degradation, including as a special case those applications composed of degradation-intolerant and possibly orthogonal subsystems. We expect that a formal characterization will induce a programming model that provides first-class abstractions for manipulating degraded results. A formally-characterizable framework for deploying such applications would then amount to a constructive proof of the Weak CAP Principle.

Traditionally, an application and system designed with incremental scalability and high availability in mind have differed from their counterparts designed without respect to these constraints. We have found that despite their simplicity, the engineering techniques we used in the design and construction of the above example applications have afforded surprising flexibility in the range of applications that can be built. Simple techniques were chosen in order to simplify the formidable programming task, and techniques with good fault isolation were favored in order to preserve the fault isolation advantages already inherent in clusters. In particular, the SNS server showed that it is possible to separate scalability and availability concerns from the design of mainline applications if the application structure can be reconciled with the design constraints imposed by the use of simple and orthogonal mechanisms.

We would like to motivate a broader research effort that extends these observations, resulting in a set of design guidelines for the construction of large-scale robust applications spanning the range from ACID to BASE [9]. We offer the initial observations here as a first step in that direction.

We thank the anonymous reviewers for their comments on the first draft and our colleagues at Stanford and Berkeley for being sounding boards for these early ideas.

References

[1] F. Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Addison–Wesley, 1975 (revised 1995).

[2] Y. Chawathe and E. A. Brewer. System support for scalable and fault tolerant internet service. In *IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98)*, Lake District, UK, Sep 1998.

[3] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Seventh USENIX Security Symposium (Security 98)*, San Antonio, TX, January 1998.

[4] S. B. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in partitioned networks. *ACM Computing Surveys*, 17(3), 1985.

[5] A. Demers, K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and B. Welch. The Bayou architecture: Support for data sharing among mobile users. In *Proceedings of the 1994 Workshop on Mobile Computing Systems and Applications*, December 1994.

[6] S. Floyd, V. Jacobson, C. Liu, and S. McCanne. A Reliable Multicast Framework for Light-Weight Sessions and Application Level Framing. In *ACM SIGCOMM '95*, pages 342–356, Boston, MA, Aug 1995.

[7] A. Fox and E. A. Brewer. Reducing WWW Latency and Bandwidth Requirements via Real-Time Distillation. In *Fifth International World Wide Web Conference (WWW-5)*, Paris, France, May 1996. WOrld WIde WEb Consortium.

[8] A. Fox, S. D. Gribble, Y. Chawathe, and E. A. Brewer. Adapting to network and client variation using active proxies: Lessons and perspectives. *IEEE Personal Communications (invited submission)*, Aug 1998. Special issue on adapting to network and client variability.

[9] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-Based Scalable Network Services. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, St.-Malo, France, October 1997.

[10] A. O. Freier, P. Karlton, and P. C. Kocher. SSL version 3.0, March 1996. Internet Draft, available at `http://home.netscape.com/eng/ssl3/ssl-toc.html`.

[11] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications: confining the wily hacker. In *1996 USENIX Security Symposium*, 1996.

[12] C. G. Gray and D. R. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. Technical Report CSL-TR-90-409, Stanford University Dept. of Computer Science, January 1990.

[13] J. Gray. The transaction concept: Virtues and limitations. In *Proceedings of VLDB*, Cannes, France, September 1981.

[14] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *ACM–SIGMOD International Conference on Management of Data*, Tucson, AZ, May 1997.

[15] Inktomi Corporation. The Inktomi technology behind HotBot, May 1996. `http://www.inktomi.com/whitepap.html`.

[16] J. J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Transactions on Computer Systems*, 10(1):3–25, February 1992.

[17] N. G. Leveson. *Safeware: System Safety and Computers*. Addison-Wesley, 1995.

[18] N. G. Leveson and C. S. Turner. An investigation of the therac-25 accidents. *IEEE Computer*, July 1993.

[19] S. McCanne, E. Brewer, R. Katz, L. Rowe, E. Amir, Y. Chawathe, A. Coopersmith, K. Mayer-Patel, S. Raman, A. Schuett, D. Simpson, A. Swan, T.-L. Tung, D. Wu, , and B. Smith. Toward a common infrastructure for multimedia networking middleware. In *7th Intl. Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV 97)*, St. Louis, MO, May 1997. Invited paper.

[20] National Laboratory for Applied Network Research. The Squid internet object cache. http://squid.nlanr.net.

[21] NCR Corp. Teradata scalable RDBMS. `http://www3.ncr.com/teradata`.

[22] G. F. Pfister. *In Search of Clusters (revised ed.)*. Addison-Wesley, 1998.

[23] A. Schuett, S. Raman, Y. Chawathe, S. McCanne, and R. Katz. A soft state protocol for accessing multimedia archives. In *NOSSDAV 97*, 1997.

[24] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, 1993.