

DARE: Surviving the Inevitable Failure of Fault-Tolerance

Abstract—A fault tolerant system is designed to operate correctly under a well-defined system model, but fails to provide any service when the assumptions in the system model are invalidated due to unexpected massive failures.

We advocate a new paradigm for building systems by augmenting fault tolerance with the properties of graceful Degradation, self-Awareness, and self-REstoration (the DARE properties). In particular, we believe that a storage subsystem with DARE properties could be a new and fundamental building block for scalable and reliable distributed systems and sketch a design for one such system.

I. INEVITABLE FAILURE OF FAULT TOLERANCE

Reliable distributed systems are typically designed to be *fault tolerant*. Fault tolerance mechanisms provably ensure system correctness, but only with respect to a system model that specifies the type and extent of failures. Most of the time, the system exists in a *normal* state, with no faulty components or by tolerating the failures of some components. Unfortunately, systems can sometimes suffer excessive failures that go beyond what is allowed in the system model. In these cases, fault tolerance mechanisms enter an *abnormal* state, are unable to mask failures, and cause “reliable” systems to fail.

Typical reliable distributed systems exhibit an *availability cliff* [15], where a system is available in any normal state, but becomes completely unavailable after transitioning to an abnormal state. For example, a system that adopts the replicated state machine approach [8], [14] tolerates a minority of machine failures, but makes no progress when a majority is unavailable. Figure I depicts the availability cliff.

It might seem that with sufficient replication one could virtually eliminate the probability of a system entering an abnormal state. However, such an argument hinges on the assumption that failures are independent. This assumption is often invalidated in some unexpected ways in practice (e.g., due to subtle software bugs [5], a certain batch of disks being defective, or accidental triggering of the power off button [2].) Furthermore, excessive replication could incur prohibitively high resource cost

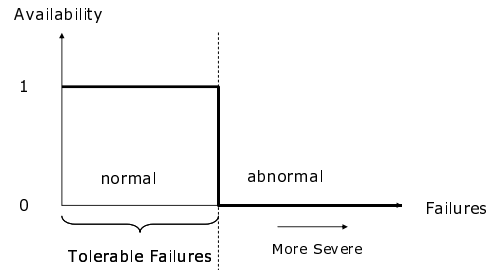


Fig. 1. Availability Cliff.

and significantly reduce system performance, thereby making it an impractical choice.

II. AUGMENTING FAULT TOLERANCE WITH DARE

Our goal is to make systems more useful by augmenting fault tolerance with **DARE** (Graceful Degradation, Self-Awareness, and Self-REstoration) properties described below.

Graceful Degradation. Complete system unavailability in an abnormal state is undesirable. There is often a natural spectrum of system degradation from complete availability to complete unavailability. Such degradation could manifest itself either as a reduction in the functions that can be performed or as a reduction in the amount and usefulness of accessible data. Ideally, the degradation should be *graceful* as more and more components fail. The degree of degradation should be proportional to the severity of failures experienced in an abnormal state. Figure II depicts a fault tolerant system that also exhibits graceful degradation.

Self-Awareness. Even with graceful degradation, system behavior in an abnormal state deviates from the original (normal state) system specification. Such deviation could be obvious to clients when certain functionality becomes unavailable; but in cases of data deterioration, the system might need to inform clients of such degradation, so that clients can interpret the data appropriately. We call this property *self-awareness*.

Self-Restoration. When individual components of the system recover, the system could transition from an

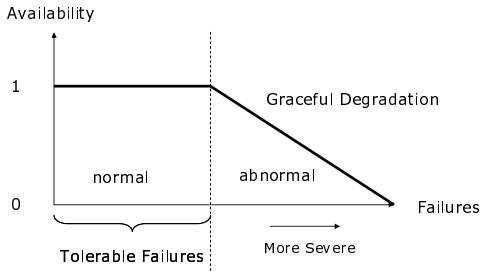


Fig. 2. Graceful Degradation.

abnormal state back into a normal one. Because non-degraded service is preferable to clients, the system should restore its non-degraded service when feasible. We refer to this ability as *self-restoration*.

III. STORAGE ABSTRACTIONS WITH DARE

The value of the DARE properties is evident for a distributed storage system on a cluster of commodity machines. Fault tolerance is a necessity for such a system because the system employs a large number of failure-prone commodity components. Because a storage system centers on data, data reliability and availability are the main goals for fault tolerance. Typically, fault tolerance is achieved by having each piece of data replicated on a *replication group* comprising a set of machines. A large-scale distributed storage system usually consists of a number of such replication groups. New groups can be added when the system expands incrementally with new machines; groups can also be reconfigured to replace faulty machines or for re-balancing.

There are two types of degradation in such a large-scale distributed storage system.

- Intra-group degradation: degradation of a replication group under excessive failures within the group.
- Inter-group degradation: degradation of system operations involving multiple replication groups when some of those groups are unavailable or degraded.

This paper focuses on inter-group degradation and will touch on intra-group degradation in Section III-E.

A. Existing systems with no graceful degradation

The existence of multiple replications groups might seem to lead to inter-group degradation naturally: However, the availability of a partial set of data on the available replication groups does not necessarily translate into graceful degradation from a client’s point of view because the partial set might not constitute a meaningful view for a client. For example, Ji et al. [6] reports that a

clustered file system built on a virtualized storage layer, such as xFS [1] and Frangipani [17], is expected to lose 63.8% of data with the loss of only 3.1% of storage units—the equivalent of a replication group. Such non-graceful degradation occurs because the unavailability of metadata stored on an unavailable replication group could render inaccessible certain data on an available replication group; for example, the access to a file might require the availability of the directory of that file.

This behavior is not peculiar to file systems; other virtualized distributed storage abstractions could also suffer from a similar problem. For example, Boxwood [10] provides the abstraction of distributed B-trees, where each tree node is stored on some replication group. When a replication group storing certain internal nodes of a B-tree fails, those internal nodes become unavailable. Consequently, all the descendants of those internal nodes become inaccessible even when their replication groups operate correctly. Figure III-A shows a simple view of how a B-tree is stored on a distributed storage system. Each rectangle represents a tree node. The circles inside a rectangle show the replication group, consisting of a pair of machines in this case, on which that tree node is stored. Node N1 is stored on machines C and D, while node N2 is stored on machines E and F. When machines C, D, and E fail, as illustrated in the figure, node N2 remains available because N2 is replicated on machine F. However, N1 becomes unavailable because both copies are unavailable, a scenario that puts the system in an abnormal state. Ideally, with graceful degradation, clients should be able to access information of all B-tree nodes stored on available replication groups. This is unfortunately not the case because the child nodes of node N1 are inaccessible as a result of N1’s unavailability.

For a distributed storage system, graceful degradation is particularly valuable when excessive permanent disk failures occur, because graceful degradation minimizes the amount of actual data loss and allows fast recovery from the available data.

B. Separating hard data from soft structure

The absence of graceful degradation in a distributed file system or a distributed B-tree occurs because these systems specify a rigid structure on their data that cannot be maintained in the presence of massive failures. One approach to achieving graceful degradation is to co-locate metadata with the data whose accessibility depends on the metadata, as done in some file systems (e.g., Archipelago [6], Pangaea [12], and D-GRAID [15]). For a B-tree, this would require that a node and all its

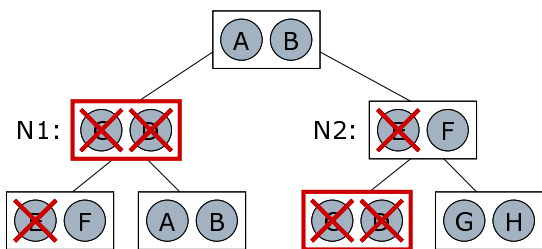


Fig. 3. Non-Graceful Degradation of a Distributed B-Tree. The unavailability of node N1 due to failures of machines C and D also causes the child nodes of N1 to be inaccessible, although at least one copy of each node is on a machine that has not failed.

ancestor nodes be co-located on the same replication group, imposing a constraint that is hard to maintain especially in the face of dynamic changes.

A different approach is to remove such dependencies by separating data elements from reconstructable global index structures. In this approach, we store the data elements on a collection of machines. The data elements may be thought of as a simple *set*. We mediate access to the data through one or more *index* structures. The index may be used to fetch only the necessary data and possibly to cache the results.

A variety of index structures may be used on a single set of data depending on the type of application that we are constructing. These index structures are *soft*, in that they can be reconstructed even when only part of the set (the *hard* state) is available.

Each element of a data set is stored on a replication group; elements of a data set could span multiple replication groups, referred to as the *group collection* for the set. This arrangement is often necessary for a large data set that does not fit on a single group. It also facilitates better load balancing and enables parallel operations on multiple groups for better performance. Each replication group maintains a local index (as hard state) that enumerates the elements stored locally.

We call the entity maintaining the global index structures the *index server* for the set. The index server could be a single machine or a set of replicated machines when replication is needed to reduce chances of index server unavailability. It is desirable to have one of the replication groups in the group collection act as the index server because a partial index is already stored locally. In case of an index server failure, (soft) global index structures can be reconstructed incrementally from the local indices stored on the replication groups in the group collection.

C. Achieving self-awareness and self-restoration

To ensure self-awareness, an index server for a data set must know whether it maintains up-to-date index structures for the set. This requires that it know the current membership of the data set’s group collection and maintain index structures that reflects the up-to-date state on every replication group in that collection.

The membership of a data set’s group collection can be maintained by a consensus service. The service employs a consensus protocol, such as Paxos [9]. A consensus service might become unavailable under excessive failures. It is desirable for the system to maximize the set of allowable operations when the consensus service is unavailable. Our use of the consensus service is described below.

When a data set is created and the first replication group selected for the elements of that data set, the consensus service is invoked to ensure that the data set has not been created on another replication group earlier and to record that replication group as the *anchor* of the data set’s group collection.¹ We disallow creation of new data sets when the consensus service is not present. However, we do allow additions to group collections even without the consensus service. This is done by having the replication groups in the group collection form a connected cluster with bi-directional links, where a link from a group *A* to group *B* indicates that *A* knows that *B* is in the same collection. A new replication group can be added only when a bi-directional link between at least one existing replication group and the new replication group are established. The consensus service ensures that there is no partition in the cluster by establishing a unique anchor for the cluster. Similarly, deletion of a replication group from a group collection can also be performed on the cluster without the consensus service, but deletion of a group *r* does require all groups directly linked to *r*, as well as *r*, be present and must ensure the cluster remain connected without *r* (by forming new links among those groups if needed) before links to and from *r* are removed. The consensus service is involved only when the data set or its anchor is deleted.

With a connected cluster for the group collection, an index server knows whether it has all replication groups in the group collection by starting at any replication group and following the links.

Once an index server has the current membership of the group collection, it exchanges periodic heartbeats

¹Alternatively, a deterministic hash function can be used to map a data set *id* to the anchor *id* for that set.

with each replication group in the collection. If the heartbeats from a replication group stop, the index server will enter a degraded mode. When an index server is unavailable due to failure or network partition, new index servers can be initiated.

Any replication group in the group collection can act as an index server. In cases of network partitions, multiple index servers for the same set might co-exist. Assuming a unique *id* for each replication group and a total ordering for those *ids*, we can give the group with the lowest *id* the highest priority in becoming the index server. If a replication group does not hear heartbeats from any lower-*id* replication group in the group collection, it will elect itself as the index server and send heartbeats to all the higher-*id* replication groups in the same group collection. It will resign if it receives heartbeats from lower-*id* replication groups. This pre-defined ordering ensures the convergence to a unique index server without a consensus service when the network partition heals.

D. Motivating applications

While the notion of DARE represents a departure from the way traditional storage applications are designed, quite a few applications fit well within this framework. Two examples include a mail service and an archival storage or data retention service. For mail service, each mailbox consists of a set of mail messages. Mail messages of a mailbox are stored on multiple replication groups. An index for message headers in a mailbox can be built to facilitate message sorting, searching, and browsing, thereby reducing traffic for fetching entire messages. When degraded, a client can access only those mail messages that are stored on available replication groups. An index server failure leads to no further degradation because the index can be reconstructed elsewhere.

From a client’s perspective, an archival storage maintains a set of documents for each account, just as a mail service. The system might provide a subset of documents when degraded. This is useful when the documents a client needs are in this subset despite excessive failures in the system. Knowing whether the results returned by the archival storage system are complete, as required by self-awareness, might be of paramount importance in some cases; for example, when the documents related to a subject are being collected for legal purposes.

It is important to note that our scheme is quite flexible, because the index structure overlaying the data set can be made arbitrarily complicated based on the needs of applications. For example, traditional notions

of hierarchies (e.g., as found in a file system) can be accommodated by associating pathname labels with each element of the hard state (e.g., files), as done in the Google file system [4], and co-locating the labels with the element. A table in a relational database is in fact a set with appropriate index structures for database operations, although support for operations that involve multiple tables (sets) is necessary in this case. It is an interesting future direction to understand how more complicated storage applications can be retrofitted to support the DARE properties.

E. Degradation within a replication group

Graceful degradation in a replication group has been extensively studied in optimistic replication [13]. Optimistic replication is proven effective in improving availability by allowing disconnected operations, a common scenario in mobile computing and in collaborative applications where each client maintains a local replica. Systems that employ optimistic replication (e.g., Coda [7] and Bayou [16]) allow operations to be performed on a minority of replication servers, resulting in divergence among the servers, and employ convergence mechanisms that ensure eventual consistency.

Self-awareness requires that the system knows when consistency is restored. If any replica can apply changes unilaterally, the complete view of the system state can be guaranteed only if the states of all replicas have been inspected. This means that the replication group enters a degraded mode as soon as a single replica is unavailable. In general, given any set S of replicas that can optimistically apply changes to the state and any set R of replicas from which a complete state can be reconstructed, there must exist at least one replica in the intersection of S and R . This poses an interesting design tradeoff.

A replication group can also incorporate optimistic replication mechanisms that bound inconsistency among the replicas (e.g., TACT [18]) for applications where such bounded inconsistency is meaningful.

IV. RELATED WORK

Gribble argues eloquently in [5] that any complex system that is designed to work only in the predicted conditions—often the case for fault tolerant systems—is destined to be fragile.

Fox and Brewer [3] propose the notion of *harvest* as a measure for graceful degradation. They identify good engineering principles, such as failure isolation through orthogonal mechanisms and replacing hard state

with soft state—many of them applicable to our context. Probabilistic availability, another proposal in the paper, corresponds to a gracefully degradable system with no fault tolerance.

File systems such as Archipelago [6], Pangaea [12], and D-GRAID [15] support graceful degradation. They do this by co-locating entire subtrees up to the root in the same fault domain. These subtrees serve as *rigid* structures over the actual data, in contrast with our approach. Also, a scheme specific to file systems appears to be less general than our proposal.

Porcupine [11] is a mail system that supports graceful degradation, but it does not support self-awareness. This is because Porcupine achieves eventual consistency only and clients do not know whether the set of messages returned by the system is complete. Also, Porcupine does not have to maintain any index structure because of the simple POP interface it supports.

V. CONCLUDING REMARKS

Distributed systems have typically adopted one of two independent design principles to deal with failures. One approach is to build in a high degree of redundancy and to assume that failures never go beyond this degree. Systems built using this approach typically provide strict consistency guarantees and full data access as long as there are limited failures. A second approach is to assume the worst as the common case: that there will always be failures that prevent strict consistency guarantees or complete access to all replicas. Systems built using this approach typically employ some form of optimistic replication and conflict resolution methods.

This paper advocates a third approach to designing reliable distributed systems. We believe that massive failures are inevitable, albeit infrequent, in real-world, large scale deployments, and feel that distributed system design principles must embrace this reality.

We believe a system should provide degradable service only when there are massive failures, must work efficiently with strict guarantees when there are few failures, must be able to transition between these operating regimes, and be able to recognize in which of these regimes it is currently operating. We have sketched the design of a particular instance of this principle as applied to a distributed storage system.

REFERENCES

- [1] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. Serverless network file systems. In *Proc. 15th Symp. Operating Systems Principles (SOSP)*, pages 109–126, December 1995.
- [2] B. Fitzpatrick. Power-loss post-mortem, January 2005. http://www.livejournal.com/community/lj_dev/670215.html.
- [3] A. Fox and E. A. Brewer. Harvest, yield and scalable tolerant systems. In *Proc. of the 7th Workshop on Hot Topics in Operating Systems*, pages 174–178, March 1999.
- [4] S. Ghemawat, H. Gobioff, and S. Leung. The Google file system. In *Proc. 19th Symp. Operating Systems Principles (SOSP)*, pages 29–43, December 2003.
- [5] S. D. Gribble. Robustness in complex systems. In *Proc. of the 8th Workshop on Hot Topics in Operating Systems*, pages 21–26, May 2001.
- [6] M. Ji, E. Felten, R. Wang, and J. Singh. Archipelago: An island-based file system for highly available and scalable Internet services. In *Proc. 4th USENIX Windows Systems Symposium*, August 2000.
- [7] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. In *Proc. of the 13th ACM Symposium on Operating Systems Principles*, pages 213–225, 1991.
- [8] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.
- [9] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.
- [10] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *Proc. of the 6th Symposium on Operating Systems Design and Implementation*, pages 105–120, December 2004.
- [11] Y. Saito, B. N. Bershad, and H. M. Levy. Manageability, availability and performance in Porcupine: A highly scalable, cluster-based mail service. *ACM Transactions on Computer Systems*, 18(3):298–332, August 2000.
- [12] Y. Saito, C. Karamonolis, M. Karlsson, and M. Mahalingam. Taming aggressive replication in the Pangaea wide-area file system. In *Proc. of the 5th Symposium on Operating Systems Design and Implementation*, pages 15–30, December 2002.
- [13] Y. Saito and M. Shapiro. Optimistic replication. Technical Report MSR-TR-2003-60, Microsoft, September 2003.
- [14] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.
- [15] M. Sivathanu, V. Prabhakaran, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Improving storage system availability with D-GRAID. In *Proc. of the 3rd USENIX Conference on File and Storage Technologies*, pages 15–30, March 2004.
- [16] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proc. of the 15th ACM Symposium on Operating Systems Principles*, pages 172–183, December 1995.
- [17] C. A. Thekkath, T. P. Mann, and E. K. Lee. Frangipani: A scalable distributed file system. In *Proc. 16th Symp. Operating Systems Principles (SOSP)*, pages 224–237, October 1997.
- [18] H. Yu and A. Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *Proc. of the 4th Symposium on Operating Systems Design and Implementation*, pages 305–318, 2000.