Graceful Degradation Via Versions: Specifications and Implementations

Lidong Zhou, Vijayan Prabhakaran, Venugopalan Ramasubramanian,
Roy Levin, and Chandramohan A. Thekkath
Microsoft Research Silicon Valley
Mountain View, CA, USA
{lidongz,vijayanp,rama,roylevin,thekkath}@microsoft.com

ABSTRACT

Correctness of a fault-tolerant system hinges on the failure model, which typically constrains the number of concurrent failures in the system. These assumptions are sometimes violated in practice, inevitably leading to degraded system behavior that deviates from the system's specification and even causing complete unavailability of the system.

This paper advocates the notion of *graceful degradation* as a complementary mechanism to fault tolerance in the design of highly available distributed systems. It provides three specifications for meaningful system behavior under degradation. The different specifications capture different tradeoffs between the *gracefulness* of degradation and the semantics preserved by a degraded view. The paper further demonstrates the practical relevance of the specifications by presenting three designs of versioned distributed storage systems that implement the specifications.

Categories and Subject Descriptors

D.4.5 [**Operating Systems**]: Reliability—*Fault-tolerance*; H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Distributed systems*

General Terms

Design, Reliability

Keywords

graceful degradation, fault tolerance, version, linearizability

1. INTRODUCTION

Fault tolerant systems are traditionally designed to operate correctly under a limited failure model. They guarantee correct system behavior as long as the number and type of failures are within the assumed limits. However, when the assumptions are violated, a fault-tolerant system either simply stops functioning or operates incorrectly. For example, replicated storage systems often maintain a small number of replicas of each data item in order to tolerate failures, but risk data loss if the number of failures exceeds the anticipated threshold.

It might seem that one could reduce the probability of such violations by increasing the level of redundancies in the system. Unfortunately, unanticipated and correlated failures do occur in practice. For example, subtle software bugs might get triggered at nodes running the same implementation of the software [5], a manufacturing defect might affect a large batch of devices such as disks, and a network or power outage might take down a large number of nodes. Tolerating such large-scale failures through over replication is impractical due to the prohibitively high cost it incurs in terms of not only storage and I/O resources, but also high complexity in performing consistent updates.

This paper advocates the notion of *graceful degradation* for system behavior under excessive, unanticipated failures. Although any system can claim to degrade gracefully by just not shutting itself off completely when facing excessive failures, it is crucial that the system continues to provide meaningful behavior in its degraded states.

A key contribution of this paper is to provide a series of specifications that define different notions of "correct" behavior in the degraded mode. These specifications apply to general systems modeled as deterministic state machines, rather than to systems with specific application semantics; they are natural extensions to the well-known notion of *linearizability* [6]. The specifications offer meaningful degradation semantics to clients, so that the clients can make use of the system even when it is degraded. Moreover, they define a measure of *gracefulness*, which quantifies the extent of degradation in the system.

The paper discusses the practical significance of these specifications in the context of a degradable versioned distributed storage system. It presents a set of system designs for implementing the specifications of degraded behavior in a versioned distributed storage system and proves that the designs meet the specifications. The semantics provided by these degradable systems have important practical implications and reveal an interesting tradeoff between gracefulness of degradation and usefulness of the semantics.

This paper is organized as follows. Section 2 defines the three specifications of degraded behavior, while Section 3 applies these specifications to a versioned distributed storage system. Further discussions on graceful degradation appear in Section 4. The paper surveys the related work in Section 5 and concludes in Section 6.

PODC'07, August 12–15, 2007, Portland, Oregon, USA.

2. GRACEFUL DEGRADATION: SPECIFICATIONS

Deterministic state machines are a well-known approach for modeling the semantics of a system. In a state machine model, the system starts with an initial state s_0, processes a sequence of commands serially, produces a response to each command, and transitions to a new state if the command demands it. We call the commands that cause state transitions *updates* and the commands that do not *queries*. The system state consists of a set of *objects*. Each command c reads a set of objects, denoted as its *read set* $c.rs$, and updates a set of objects, denoted as its *write set* $c.ws$.

Clients issue commands and might do so concurrently. We assume that the system's execution of commands is *serializable* [6, 12, 16]. For an execution $\mathcal{E}$ consisting of a set C of commands, a total order $\prec$ on commands in C defines a *serialization* $\mathcal{E}_s$, which represents a serial execution of all commands in C in the total order. A serialization $\mathcal{E}_s$ of $\mathcal{E}$ is *valid* if and only of $\mathcal{E}_s$ and $\mathcal{E}$ lead to identical final states and each command $c \in C$ reads the same values and produces the same response in the two executions. We use a sequence of commands $(c_1, c_2, \ldots, c_n)$ to represent an execution that processes $c_1, c_2, \ldots, c_n$ serially in this order.

Linearizability imposes an additional real-time ordering constraint. For an execution $\mathcal{E}$ consisting of a set of commands C, a *valid linearization* of $\mathcal{E}$ is a valid serialization of $\mathcal{E}$ with a total order $\prec$ that satisfies the following property: if a command $c_1 \in C$ completes before another command $c_2 \in C$ starts in $\mathcal{E}$, then $c_1 \prec c_2$ holds.

In this paper, we focus on systems that achieve linearizability when non-degraded. Consequently, each execution $\mathcal{E}$ has a valid linearization $\mathcal{E}_l$ that defines an equivalent serial execution. We therefore use $\mathcal{E}_l$ to represent $\mathcal{E}$ for convenience. For example, $(c_1, c_2, \ldots, c_n)$ describes an execution that has a valid linearization with a total order $c_1 \prec c_2 \prec \ldots \prec c_n$. We further use $M(c_1, c_2, \ldots, c_i)$ to denote the state of the system implementing state machine M after executing $c_1, c_2, \ldots, c_i$ serially. As a special case, for an empty sequence, $M(\phi) = s_0$, the initial state of the system.

Given an execution $\mathcal{E} = (c_1, c_2, \ldots, c_n)$ starting with an initial state s_0 and resulting in a state s_n, any command issued to the system after the completion of $\mathcal{E}$ will be executed on state s_n. However, under excessive failures, the latest values of the objects required by a command might not be available, and traditional fault tolerant systems would simply fail the command.

Rather than having the system fail the command, we propose a notion of *graceful degradation*, where the system could execute commands on some coherent intermediate state when the latest state is unavailable. We call this coherent intermediate state a *valid degraded state*. In this paper, we provide three specifications of valid degraded states for queries. Our specifications of valid degraded states enable the system to provide "meaningful" responses to queries even under excessive failures. Allowing degraded updates raises challenging issues, which we discuss in Section 4.

Common to all the specifications is the intuition that a valid degraded state reflects some partial history of the system execution. Natural definitions of a partial history include a prefix or a subsequence of an equivalent serial execution. However, not every subsequence constitutes a reasonable partial history: A partial history must preserve the semantics of its operations in the actual execution in that the commands in the partial history should read the same values from the system state, produce the same output, and make the same updates as in the original execution. We define the notion of *valid subsequence* as follows:

DEFINITION 2.1. **(Valid Subsequence)** *For an execution $\mathcal{E} = (c_1, c_2, \ldots, c_n)$, a subsequence $\mathcal{E}' = (c_{j_1}, c_{j_2}, \ldots, c_{j_k})$ with $1 \leq j_1 < j_2 < \ldots < j_k \leq n$ is* valid *if and only if for each c_{j_i} with $1 \leq i \leq k$ and for each object $o \in c_{j_i}.ws \cup c_{j_i}.rs$, the value of o before the execution of c_{j_i} in $\mathcal{E}$ is the same as before the execution of c_{j_i} in $\mathcal{E}'$; that is, let l be the index of c_{j_i} in $\mathcal{E}$, the value of o is the same after executing $c_1, c_2, \ldots, c_{l-1}$ as the value of o after executing $c_{j_1}, c_{j_2}, \ldots, c_{j_{(i-1)}}$.*

The following presents the precise definitions of the specifications: *Prefix Linearizability* captures any state that is the result of executing a prefix of the linearization; *Prefix Serializability* captures any state that is the result of executing a prefix of any valid serialization; *Subsequence Linearizability* captures any state that is the result of executing a valid subsequence of the linearization.

DEFINITION 2.2. **(Prefix Linearizability)** *For execution $\mathcal{E} = (c_1, c_2, \ldots, c_n)$, any state $M(c_1, c_2, \ldots, c_k)$ for $1 \leq k \leq n$ is considered a valid degraded state.*

DEFINITION 2.3. **(Prefix Serializability)** *Given an execution $\mathcal{E} = (c_1, c_2, \ldots, c_n)$ and a valid serialization $(c_{i_1}, c_{i_2}, \ldots, c_{i_n})$ of $\mathcal{E}$, any state $M(c_{i_1}, c_{i_2}, \ldots, c_{i_k})$ for $1 \leq k \leq n$ is considered a valid degraded state.*

DEFINITION 2.4. **(Subsequence Linearizability)** *Given an execution $\mathcal{E} = (c_1, c_2, \ldots, c_n)$, any state $M(c_{j_1}, c_{j_2}, \ldots, c_{j_k})$ for a valid subsequence $(c_{j_1}, c_{j_2}, \ldots, c_{j_k})$ with $k \geq 0$ and $1 \leq j_1 < j_2 < \ldots < j_k \leq n$ is considered a valid degraded state.*

We could have defined Subsequence Serializability as yet another definition, but it turns out to be the same as Subsequence Linearizability.

It is clear that Prefix Linearizability is stronger than both Prefix Serializability and Subsequence Linearizability as any valid degraded state with respect to Prefix Linearizability is valid with respect to Prefix Serializability and Subsequence Linearizability. This is because the valid linearization is a valid serialization and its prefix is by definition a valid subsequence.

The difference between Prefix Serializability and Subsequence Linearizability is subtle. Any command in a valid subsequence has the same effect as in the original execution; the same is true for any command in a valid serialization (or any of its prefixes.) However, unlike a valid serialization, the valid subsequence does not impose any constraints on commands that are excluded from the subsequence: it is as if those excluded commands were undone from the original execution. It might not be feasible to extend a valid subsequence into a valid serialization.

In all three specifications, $n - k$ indicates the number of commands the valid degraded state fails to reflect. We call $n - k$ the *degradation degree* and use it to quantify the gracefulness of a degraded state. This metric facilitates applications to evaluate different implementations of graceful degradation.

3. GRACEFUL DEGRADATION VIA VERSIONS

A specification is useful only if interesting new systems that conform to the specification can be designed. This section presents three different designs, each for a particular one of the three specifications.

The intuition behind the specifications for valid degraded states is to provide clients with some state reflecting a partial execution (hence the use of the execution prefix and subsequence) when the most up-to-date view is unavailable. The maintenance of multiple versions for each piece of data makes it possible to expose such a state. Our designs are for such versioned systems; as will be noted in Section 5, there are many such practical versioned systems.

In a versioned system, each update creates new versions of objects, rather than overwriting the current versions. Version v of an object o is referred to as object version $\langle o, v\rangle$. In a distributed versioned system, different versions of the same object could reside on different sets of storage servers to reduce the chances of all versions of an object being unavailable due to server failures.

Valid Degraded State and Coherency: A degraded state consists of a set of object versions, one for each object. It might seem that one could simply define the degraded state to include all the most recent available object versions, one for each object. This could lead to states that are clearly unreasonable. For example, consider an execution $\mathcal{E} = (c_1, c_2)$, where c_1 creates $\langle o_1, 1\rangle$ and $\langle o_2, 1\rangle$, and c_2 creates $\langle o_1, 2\rangle$ and $\langle o_2, 2\rangle$. Consider the case where $\langle o_1, 2\rangle$ becomes unavailable. A degraded state consisting of the most recent available versions would be $\{\langle o_1, 1\rangle, \langle o_2, 2\rangle\}$; this state is problematic because it reflects a partial completion of c_2, and violates the atomicity of command c_2.

From a practical point of view, when some latest object versions become unavailable, for a query, the system wants to find the most recent and available versions of the requested objects that are *coherent*. With a specification for valid degraded states, a set of object versions is coherent if and only if all these object versions co-exist in some valid degraded state.

At a given system state, there might be multiple coherent sets for a given set of objects. We define a *more-recent* relation among the coherent sets for the same system state and the same set of objects. The discussions in the rest of the paper are always on a particular system state and often on a particular set of objects, so we omit mentioning those when the context is clear. Given two coherent sets S_1 and S_2 for a set O of objects, we define S_1 to be *more recent* than S_2 if and only if the following holds: for any $o \in O$, let $\langle o, v_1\rangle \in S_1$ and $\langle o, v_2\rangle \in S_2$ be the corresponding object versions in the two coherent sets, $v_1 \geq v_2$ holds. Given a set O of objects at a given system state, a coherent set is called *most recent* if no other coherent sets are more recent. There might not be a unique most recent set because more-recent relation for a given set of coherent sets might not constitute a total order. We further assume that each object has an initial version (version number 0) when the system starts. Any set of those initial object versions is defined to be coherent.

In the rest of this section, we present three designs; for each design, we present a scheme that allows the system to check whether a set of object versions are coherent, prove that the coherency is consistent with one of the three specifications, and show how to find a most recent coherent set.

3.1 Timestamp-Based Design

The most natural design for checking the coherency of an object version set is to see whether those object versions co-exist at some (real) time in the past. This can easily be achieved if each object version records one timestamp to indicate when it is created and another timestamp to indicate when a newer version of the same object is created. The two timestamps specify the time interval during which the object version exists. A set of object versions are coherent as long as the time intervals of all object versions overlap; that is, there exists a time that falls into the interval of every object version in the set.

How the timestamps are assigned demands care. We cannot simply use the real-time values at which object versions are created even if we assume that all servers in the system have perfectly synchronized clocks. For example, consider an execution of two commands. Command c_1 creates object version $\langle o_2, 1\rangle$ at time t_2, while command c_2 creates object version $\langle o_1, 1\rangle$ at t_1 and $\langle o_2, 2\rangle$ (without reading $\langle o_2, 1\rangle$) at t_3. With $t_1 < t_2 < t_3$, (c_1, c_2) forms a valid linearization. Using those timestamps, $\{\langle o_1, 1\rangle, \langle o_2, 1\rangle\}$ might be considered coherent because they co-exist at t_2. However, the set reflects a partial execution of c_2 and violates atomicity. The problem arises because t_2 is in the middle of the execution of c_2. So, timestamps must be assigned to reflect the linearization order, but also make sure that they do not divide the execution of any command.

The timestamp-based concurrency control techniques [1], a well-known approach to serialization in database systems, can be used for creating such timestamps. In the basic scheme, each transaction is assigned a timestamp. The execution of committed transactions must be equivalent to an execution of those in the serialized order based on the assigned timestamps. This can be enforced if the system maintains a read timestamp $o.rt$ and a write timestamp $o.wt$ for each object o, where the read (resp. write) timestamp tracks the highest timestamp among the transactions read (resp. write) the object. Any transaction with timestamp t, denoted by T_t, will be aborted if it attempts to write to an object o with $o.rt > t$ or read an object o with $o.wt > t$.

Multiversion concurrency control (MVCC) [17] is an improvement over the basic scheme by maintaining multiple versions for each object. For each object version $\langle o, v\rangle$, $\langle o, v\rangle.rt$ records the timestamps of all transactions reading that object version and $\langle o, v\rangle.wt$ records the timestamp of the transaction that creates that object version. A transaction T_t reading an object o will read the highest object version $\langle o, v\rangle$ with $\langle o, v\rangle.wt < t$. A write by a transaction T_t will be rejected if there exists a read timestamp $t_r \in \langle o, v\rangle.rt$ for some v, such that $t_r > t$ and $t_r < \langle o, v'\rangle.wt$ for the lowest version $\langle o, v'\rangle$ with $\langle o, v'\rangle.wt > t$. Using the timestamps at the starting time for each transaction, MVCC yields a valid linearization of all committed transactions based on the timestamps.

We can extend MVCC to define coherency as follows. For an object version $\langle o, v\rangle$, denote its previous and next versions as $\langle o, v\rangle.prev$ and $\langle o, v\rangle.succ$ respectively. The system stores with each object version $\langle o, v\rangle$ not only $\langle o, v\rangle.wt$ but also $\langle o, v\rangle.succ.wt$; we set $\langle o, v\rangle.succ.wt$ to ∞ for any $\langle o, v\rangle.succ$ that does not yet exist.

DEFINITION 3.1. **(Timestamp-Based Coherency)** *A set S of object versions is coherent if and only if there exists a timestamp t, such that for each $\langle o, v\rangle \in S$, condition $\langle o, v\rangle.wt \leq t < \langle o, v\rangle.succ.wt$ holds.*

It is straightforward to show that this definition of coherency satisfies Prefix Linearizability. The time t in the definition can be associated with a coherent set; it is obvious that the most recent coherent set is the one with the highest timestamp t.

THEOREM 3.1. *Timestamp-Based Coherency satisfies Prefix Linearizability.*

PROOF. If a set S of object versions satisfies Timestamp-Based Coherency, then there exists a timestamp t such that, for each $\langle o, v\rangle \in S$, condition $\langle o, v\rangle.wt \leq t < \langle o, v\rangle.succ.wt$ holds. For the valid linearization based on the timestamps, consider the prefix p that includes all commands (transactions) with timestamps up to t. For each $\langle o, v\rangle \in S$, the command creating this version has been executed, but the command creating its next version has not. Therefore, $\langle o, v\rangle$ is in the state after executing p. □

Finding the Most Recent Coherent Set: To find the most recent coherent set for a set O of objects, starting with a set S of the most recent available versions of the objects in O. Let $\langle o, v\rangle$ be the object version in S with the highest timestamp t. If there exists an $\langle o', v'\rangle \in S$ such that $t \geq \langle o', v'\rangle.succ.wt$, replace $\langle o, v\rangle$ in S with o's highest available version that has a timestamp lower than t. Repeat this process until no such $\langle o', v'\rangle$ is found. This is guaranteed to terminate because each object has a finite number of versions and each iteration reduces the version of an object in O.

Timestamp-Based Coherency conveys a simple degradation semantics: it provides a state that existed sometime in the past. The system simply lets clients "time-travel" [14] to the latest point in the past where the requested view is completely preserved. This is a well-defined semantics that clients can easily make use of: clients clearly understand the semantics of the state because it was the up-to-date state sometime in the past.

However, Timestamp-Based Coherency is rather strong. Consider the example in Figure 1, which shows an execution $(c_1, c_2, c_3, c_4, c_5)$. For each command, Figure 1 shows the set of object versions it reads, referred to as the *read version set*, and the set of object versions it creates, referred to as the *write version set*. In the case where all object versions except $\langle o_1, 2\rangle$ are available, with Timestamp-Based Coherency, the most recent coherent set for all objects will be $\{\langle o_1, 1\rangle, \langle o_2, 0\rangle, \langle o_3, 1\rangle\}$; that is, the state after executing c_1 only. Command c_3 has to be excluded even though it operates on an unrelated object o_2; this could exacerbate data loss when the system tries to recover to a coherent state after suffering permanent loss of some object versions.

Consider a concrete example with a system that maintains the source code tree of a large piece of software (e.g., Linux or Cygwin.) If the latest version of a file created in March of 2006 becomes unavailable and the last update was in February of 2006, Timestamp-Based Coherency will present the source tree view in February of 2006 to the clients. The clients will be able to build this source tree and obtain an old version of the software.

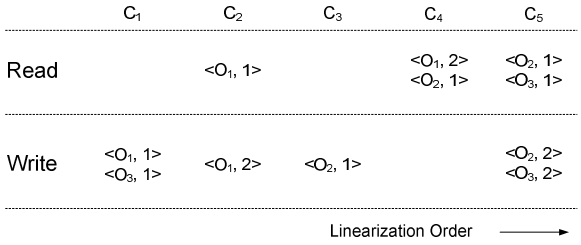

Figure 1: An example execution $(c_1, c_2, c_3, c_4, c_5)$. The read version set and the write version set are shown for each command.

There might be independent packages (e.g., packages for python and perl) in the source tree. Suppose the python package was updated in February and April of 2006, where the perl package was updated in January and March of 2006. The loss of the March-2006 version of the perl package ideally should not force the use of the old version of python in February of 2006, as Timestamp-Based Coherency mandates. Prefix Serialization does allow the view with the January 2006 version of perl and April 2006 version of python, thereby allowing a more graceful degradation. Next section shows how this can be realized.

3.2 Dependency-Based Design

In the example of Figure 1, Prefix Serialization allows a view that reflects the execution of more commands. Note that $(c_1, c_3, c_2, c_4, c_5)$ constitutes a valid serialization. With the loss of $\langle o_1, 2\rangle$, a prefix (c_1, c_3) of the serialization yields a state with $\{\langle o_1, 1\rangle, \langle o_2, 1\rangle, \langle o_3, 1\rangle\}$. Observe that a valid serialization can be obtained by swapping c_2 and c_3 in the linearized execution. This is because c_2 and c_3 are *independent* as defined below.

We use $c.rs$ and $c.ws$ to denote the set of objects that a command c reads and writes. We further use $c.rvs$ (read version set) and $c.wvs$ (write version set) for the object versions c reads and writes. Two commands c and c' $(c \prec c')$ are defined to be *conflicting* if $c.rs \cap c'.ws \neq \phi$ (Read-Write Conflict), $c.ws \cap c'.ws \neq \phi$ (Write-Write Conflict), or $c.ws \cap c'.rs \neq \phi$ (Write-Read Conflict) holds. Otherwise, they are *independent*.

It is easy to see that, starting with a valid linearization of an execution, swapping two consecutive independent commands one or more times leads to a valid serialization. This is because each command in the new sequence will be reading the same set of object versions and creating the same set of object versions as in the original execution.

In order to check whether a set of object versions satisfies the coherency consistent with Prefix Serialization, for each object version ov, the system maintains two sets $ov.Dep$ and $ov.Equiv$ during an execution $\mathcal{E}$ as follows.

When executing a command c in $\mathcal{E}$, for each object version in its write version set, its *Dep* is set to be the union of the following: c's write version set, the *Dep* sets of the object versions in c's read version set, the *Equiv* sets of the previous versions of those in c's write version set, and the *Dep* sets of the previous versions of those in c's write version set. The *Equiv* for each object version in c's write set is set to empty.

$$\forall ov \in c.wvs,\ ov.Dep := c.wvs \quad (1)$$
$$\cup \bigcup_{rov \in c.rvs} (rov.Dep) \quad (2)$$
$$\cup \bigcup_{wov \in c.wvs} (wov.prev.Equiv) \quad (3)$$
$$\cup \bigcup_{wov \in c.wvs} (wov.prev.Dep) \quad (4)$$
$$ov.Equiv := \phi \quad (5)$$

For each object version in c's read version set, set its *Equiv* set to be the union of the current value of its *Equiv* set, the *Dep* sets for the object versions in c's read version set, and the *Dep* sets for the previous versions of those in c's write version set.

$$\forall ov \in c.rvs,\ ov.Equiv := ov.Equiv \quad (6)$$
$$\cup \bigcup_{rov \in c.rvs} (rov.Dep) \quad (7)$$
$$\cup \bigcup_{wov \in c.wvs} (wov.prev.Dep) \quad (8)$$

Intuitively, $ov.Dep$ captures the set of object versions that ov depends on: any lower versions for those objects can never co-exist with ov. Also, note that both *Dep* and *Equiv* are reflective. We define the coherency condition as follows.

DEFINITION 3.2. **(Dependency-Based Coherency)** *A set S of object versions is coherent with respect to an execution $\mathcal{E}$ if and only if for any object versions $\langle o_1, v_1\rangle \in S$ and $\langle o_2, v_2\rangle \in S$, the following holds: if $\exists v$, $\langle o_1, v\rangle \in \langle o_2, v_2\rangle.Dep$, then $v \leq v_1$ holds; similarly, if $\exists v$, $\langle o_2, v\rangle \in \langle o_1, v_1\rangle.Dep$, then $v \leq v_2$ holds.*

Before we prove that Dependency-Based Coherency satisfies Prefix Serializability, we first prove a lemma that relates *Dep* with valid serializations.

LEMMA 3.1. *Given an execution $\mathcal{E}$, consider any valid serialization $\mathcal{E}_s$ of $\mathcal{E}$ that is the result of swapping consecutive independent commands one or more times, if $\mathcal{E}_s$ contains a consecutive sequence of commands $c_1, c_2, \ldots, c_k$[1] with $k \geq 1$, such that c_i and c_{i+1} are conflicting for any $1 \leq i < k$, then the following holds:*
$\forall ov \in c_1.wvs,\ \forall ov' \in c_k.wvs,\ ov \in ov'.Dep$

PROOF. Prove by induction on k.
Induction Basis: Consider the case with $k = 1$; that is, $c_1 = c_k$ holds. For any $ov \in c_1.wvs$ and $ov' \in c_k.wvs$, condition $ov \in ov'.Dep$ holds.[2]

$$\begin{aligned} ov \in c_1.wvs &\quad \text{(Assumption)} \\ = c_k.wvs &\quad (k = 1) \\ \subseteq ov'.Dep. &\quad (1) \end{aligned}$$

[1]Note that $c_1, c_2, \ldots, c_k$ is a continuous subsequence of $\mathcal{E}_s$, but not necessarily in the original execution $\mathcal{E}$.
[2]In the proof, we provide the justifications for each step (by referring to the definitions of *Dep* and *Equiv* or the previous assumptions) in the parenthesis on the right-hand side of the proof steps.

Induction Step: assume that the condition holds for any sequence with length less than k for some $k > 1$, we want to show that the condition holds for a sequence of length k.

We look at the last two commands c_{k-1} and c_k in any sequence of length k. By definition, the two commands are conflicting. We therefore consider the following three cases.
Case I (Write/Read Conflict:) c_{k-1} and c_k are conflicting because $c_{k-1}.ws \cap c_k.rs \neq \phi$. Because c_{k-1} and c_k are consecutive in a valid serialization that is obtained by swapping consecutive independent operations one or more times, there exists ov, such that the following holds:

$$ov \in c_{k-1}.wvs \quad (9)$$
$$ov \in c_k.rvs \quad (10)$$

For any $ov_1 \in c_1.wvs$ and any $ov_k \in c_k.wvs$,

$$\begin{aligned} ov_1 \in ov.Dep &\quad \text{(Induction Hypothesis, 9)} \\ \subseteq ov_k.Dep. &\quad (10, 2) \end{aligned}$$

Case II (Write/Write Conflict:) c_{k-1} and c_k are conflicting because $c_{k-1}.ws \cap c_k.ws \neq \phi$. Because c_{k-1} and c_k are consecutive in a valid serialization that is obtained by swapping consecutive independent commands one or more times, there exists ov, such that the following holds:

$$ov \in c_k.wvs \quad (11)$$
$$ov.prev \in c_{k-1}.wvs \quad (12)$$

For any $ov_1 \in c_1.wvs$ and any $ov_k \in c_k.wvs$,

$$\begin{aligned} ov_1 \in (ov.prev).Dep &\quad \text{(Induction Hypothesis, 12)} \\ \subseteq ov_k.Dep &\quad (11, 4) \end{aligned}$$

Case III (Read/Write Conflict:) c_{k-1} and c_k are conflicting because $c_{k-1}.rs \cap c_k.ws \neq \phi$. Because c_{k-1} and c_k are consecutive in a valid serialization that is obtained by swapping consecutive independent operations one or more times, there exists ov, such that the following holds:

$$ov.prev \in c_{k-1}.rvs \quad (13)$$
$$ov \in c_k.wvs \quad (14)$$

We consider two subcases depending on whether $c_{k-1}.wvs = \phi$ holds.
Subcase III.A: Consider the case where $c_{k-1}.wvs = \phi$ holds. In this case, the only way that c_{k-1} and c_{k-2} can be conflicting is because there exists ov', such that the following holds (Write/Read Conflict:)

$$ov' \in c_{k-2}.wvs \quad (15)$$
$$ov' \in c_{k-1}.rvs \quad (16)$$

Given any $ov_1 \in c_1.wvs$ and any $ov_k \in c_k.wvs$,

$$\begin{aligned} ov_1 \in ov'.Dep &\quad \text{(Induction Hypothesis, 15)} \\ \subseteq ov.prev.Equiv &\quad (13, 16, 6, 7) \\ \subseteq ov.Dep &\quad (14, 3) \\ \subseteq ov_k.Dep &\quad (14, 1) \end{aligned}$$

Subcase III.B: Consider the case where $c_{k-1}.wvs = \phi$ does not hold. Choose an arbitrary $ov_{k-1} \in c_{k-1}.wvs$.

Given any $ov_1 \in c_1.wvs$ and any $ov_k \in c_k.wvs$,

$$\begin{aligned} ov_1 &\in ov_{k-1}.Dep \qquad \text{(Induction Hypothesis)} \\ &\subseteq ov.prev.Equiv \qquad (13, 6, 8) \\ &\subseteq ov_k.Dep \qquad (14, 3) \end{aligned}$$

This concludes the induction step. □

THEOREM 3.2. *Dependency-Based Coherency satisfies Prefix Serializability.*

PROOF. It suffices to show that, if two object versions ov_1 and ov_2 are coherent with respect to an execution $\mathcal{E}$, then there exists a valid serialization of that execution, such that the two object versions co-exist after executing a prefix of that serialization.

Proof by contradiction: If there does not exist a valid serialization such that the two object versions co-exist after executing a prefix of that serialization, we want to show that ov_1 and ov_2 are not coherent.

Let c be the command creating ov_1 and c' be the command creating ov_2. Without loss of generality, we assume that c precedes c' in $\mathcal{E}$. Let c_s be the command creating $ov_1.succ$. According to our assumption, because $\mathcal{E}$ is a valid serialization, ov_1 and ov_2 do not co-exist after executing any prefix of $\mathcal{E}$. Therefore, c_s must precede c' in $\mathcal{E}$.

We then claim that there must exist a valid serialization $\mathcal{E}_s$ that contains a consecutive sequence $c_s = c_{j1}, c_{j2}, \ldots, c_{jk} = c'$ for some $k \geq 1$, such that c_{ji} and $c_{j(i+1)}$ are conflicting, for any $1 \leq i \leq k-1$. This is because, otherwise, a series of swapping between consecutive independent commands on $\mathcal{E}$ will lead to a valid serialization with c' preceding c_s (but remain after c). In that valid serialization, executing the prefix up to c' will yield a state with both ov_1 and ov_2 because the update to ov_1 in c_s is after c'. This creates a contradiction to our assumption.

Applying Lemma 3.1 on the consecutive sequence in $\mathcal{E}_s$, we have $ov_1.succ \in ov_2.Dep$ holds, which implies that ov_1 and ov_2 are not coherent. Contradiction. □

Finding a Most Recent Coherent Set: To find a most recent coherent set for a given set O of objects, start with the set S of the most recent available versions for objects in O, and perform the following step repeatedly until a coherent set is found: If there exists $\langle o_1, v_1\rangle \in S$, $\langle o_2, v_2\rangle \in S$, and $\langle o_1, v\rangle \in \langle o_2, v_2\rangle.Dep$ such that $v_1 < v$, replace $\langle o_2, v_2\rangle$ with the highest available version of o_2 that is lower than v_2.

Note that different orders of choosing conflicting pairs could lead to different results; that is, there could be multiple most recent coherent sets that are not comparable.

3.3 Weak-Dependency-Based Design

Dependency-Based Coherency can be weakened; interestingly, one weakened definition achieves Subsequence Linearizability and has practical relevance. The weakened coherency definition uses the following Dep_w instead of Dep.

$$\begin{aligned} \forall ov \in c.wvs,\ ov.Dep_w := c.wvs \\ \cup \bigcup_{rov \in c.rvs} (rov.Dep_w) \\ \cup \bigcup_{wov \in c.wvs} (wov.prev.Dep_w) \end{aligned}$$

By definition of Dep_w, the following lemma trivially holds.

LEMMA 3.2. *Given a command c in an execution $\mathcal{E}$, for any $ov \in (c.wvs \cup c.rvs)$ and $ov' \in c.wvs$, $ov.Dep_w \subseteq ov'.Dep_w$ holds.*

We define Weak-Dependency-Based Coherency as follows.

DEFINITION 3.3. **(Weak-Dependency-Based Coherency)** *For a given execution $\mathcal{E}$, a set S of object versions is coherent if and only if for any object versions $\langle o_1, v_1\rangle$ and $\langle o_2, v_2\rangle$ in S, the following holds: if $\exists v$, $\langle o_1, v\rangle \in \langle o_2, v_2\rangle.Dep_w$, then $v \leq v_1$ holds; if $\exists v$, $\langle o_2, v\rangle \in \langle o_1, v_1\rangle.Dep_w$, then $v \leq v_2$ holds.*

To show that Weak-Dependency-Based Coherency achieves Subsequence Linearizability, we further introduce an *undo* construction to create valid subsequences. Intuitively, the *undo-subsequence* of ov removes every command with an object version in its read version set or its write version set that depends on ov based on Dep_w.

DEFINITION 3.4. **(Undo-Subsequence)** *An* Undo-Subsequence *with respect to an execution $\mathcal{E}$ and an object version ov is a subsequence of $\mathcal{E}$ that excludes every command c' with $ov \in ov'.Dep_w$ for some $ov' \in (c'.wvs \cup c'.rvs)$.*

LEMMA 3.3. *Any prefix of an Undo-Subsequence is a valid subsequence.*

PROOF. To show that a subsequence of an execution $\mathcal{E}$ is valid, it suffices to show that, for each command c in the subsequence, any object version $rov \in c.rvs$[3] is in the set of object versions generated in a previous command in the subsequence, and for any object version $wov \in c.wvs$, $wov.prev$ is in the set of object versions created in a previous command in the subsequence.

Proof by induction on the length of a prefix of an undo-subsequence:

Induction Basis: Any undo-subsequence prefix with length 0 is by definition valid.

Induction Step: Assume that any undo-subsequence prefix of length less than $k \geq 1$ is valid. We show that an undo-subsequence prefix of length k is valid. Assume otherwise. Let the k^{th} command on the offending subsequence be c_k. Due to induction hypothesis, the prefix of length $k-1$ is valid. Therefore, c_k is the command that causes the prefix of length k not to be valid.

There are two cases for c_k to cause the violation. In Case I, one of the object versions in c_k's read version set is generated by a command that is excluded from the undo-subsequence. That is, there exists an $ov' \in c_k.rvs$ such that the command c with $ov' \in c.wvs$ is excluded from the subsequence during construction. Because c is excluded, we have $ov \in ov''.Dep_w$ for some $ov'' \in (c.wvs \cup c.rvs)$. Because of $ov' \in c.wvs$, by construction of Dep_w, $ov''.Dep_w \subseteq ov'.Dep_w$ holds. Therefore, $ov \in ov'.Dep_w$ holds and c_k should be removed from the undo-subsequence. Contradiction.

In Case II, the command that generates the previous version of some object version in c_k's write version set is excluded from the undo-subsequence. That is, there exists $ov' \in c_k.wvs$ such that the command c with $ov'.prev \in c.wvs$ is removed from the subsequence during construction. Because c does not exist in the undo-subsequence, we have

[3]Note that $c.rvs$ and $c.wvs$ are defined with respect to the original execution $\mathcal{E}$, not the execution of the subsequence.

$ov \in ov''.Dep_w$ for some $ov'' \in (c.wvs \cup c.rvs)$. Due to Lemma 3.2, $ov''.Dep_w \subseteq ov'.prev.Dep_w$ holds. By definition of Dep_w, we also have $ov'.prev.Dep_w \subseteq ov'.Dep_w$. Again, c_k should be removed from the undo-subsequence. Contradiction. □

To show that Weak-Dependency-Based Coherency satisfies Subsequence Linearizability, we construct a valid subsequence that is obtained by "undoing" the commands that create the object versions that are unavailable due to failures. A formal proof follows.

THEOREM 3.3. *Weak-Dependency-Based Coherency satisfies Subsequence Linearizability.*

PROOF. It suffices to show that, given an execution $\mathcal{E}$, if a pair of object versions ov_1 and ov_2 satisfies Weak-Dependency-Based Coherency, then there exists a valid subsequence whose execution will yield a state containing the pair of object versions.

Without loss of generality, we assume that the command c_1 that creates ov_1 precedes the command c_2 that creates ov_2. If $ov_1.succ$ does not exist, then ov_1 is the last version for the object. Therefore, ov_1 and ov_2 will co-exist after executing the prefix of $\mathcal{E}$ up to c_2.

If $ov_1.succ$ does exist, let s be the undo-subsequence with respect to $\mathcal{E}$ and $ov_1.succ$. Clearly, s contains c_1. Because ov_1 and ov_2 satisfy Weak-Dependency-Based Coherency, $ov_1.succ \notin ov_2.Dep_w$ holds. Due to Lemma 3.2 and the definition of undo-subsequence, s also contains c_2. Let $ov_1 = \langle o_1, v_1 \rangle$. By definition of Dep_w, s excludes any commands that create object version $\langle o_1, v \rangle$ with $v > v_1$.

Consider the prefix p of s up to command o_2. By Lemma 3.3, p is a valid subsequence. We now show that ov_1 and ov_2 co-exist in the state after executing p. The state contains ov_1 because p contains o_1, but not commands creating higher versions of the same object. The state also contains ov_2 because o_2, the last command in the sequence, creates ov_2. □

Finding a Most Recent Coherent Set: We can use a similar algorithm as in Section 3.2, replacing *Dep* with Dep_w. To find the most recent coherent set for a given set O of objects, start with the set S of the most recent available versions for objects in O and performs the following step repeatedly until a coherent set is found: If there exists $\langle o_1, v_1 \rangle \in S$, $\langle o_2, v_2 \rangle \in S$, and $\langle o_1, v \rangle \in \langle o_2, v_2 \rangle.Dep_w$ such that $v_1 < v$, replace $\langle o_2, v_2 \rangle$ with the highest available version of o_2 that is lower than v_2.

3.4 A Comparison of Three Approaches

Because Prefix Linearizability is stronger than Prefix Serializability and Subsequence Linearizability, Timestamp-Based Coherency is stronger than Dependency-Based Coherency and Weak-Dependency-Based Coherency: a set of object versions that satisfies Timestamp-Based Coherency is guaranteed to satisfy Dependency-Based Coherency and Weak-Dependency-Based Coherency for the same execution.

It is also clear that, by the constructions of *Dep* and Dep_w, for the same execution $\mathcal{E}$ and for any object version ov, $ov.Dep_w \subseteq ov.Dep$ holds. Therefore, Dependency-Based Coherency is stronger than Weak-Dependency-Based Coherency.

Consequently, given any system state, there exists a most recent coherent set S_1 for Weak-Dependency-Based Coherency, a most recent coherent set S_2 for Dependency-Based Coherency, and the most recent coherent set S_3 for Timestamp-Based Coherency, such that S_1 is more recent than S_2, which is in turn more recent than S_3. If the protocols for finding a most recent coherent set for Dependency-Based Coherency or Weak-Dependency-Based Coherency always choose to roll back object versions with the highest timestamp and break ties in the same deterministic way, then the resulting most recent coherent sets will obey the more-recent relation. We assume the use of such protocols in the rest of this section.

A stronger coherency condition translates into a higher degradation degree. This is illustrated in the example shown in Figure 1. When object version $\langle o_1, 2 \rangle$ becomes unavailable, the most recent coherent set according to Timestamp-Based Coherency is $\{\langle o_1, 1 \rangle, \langle o_2, 0 \rangle, \langle o_3, 1 \rangle\}$, reflecting only c_1, and therefore a degradation degree of 4. The most recent coherent set according to Dependency-Based Coherency is $\{\langle o_1, 1 \rangle, \langle o_2, 1 \rangle, \langle o_3, 1 \rangle\}$, reflecting c_1 and c_3, and therefore a degradation degree of 3. The most recent coherent set according to Weak-Dependency-Based Coherency is $\{\langle o_1, 1 \rangle, \langle o_2, 2 \rangle, \langle o_3, 3 \rangle\}$, reflecting c_1, c_3, and c_5, and therefore a degradation degree of 2.

The difference between the set for Dependency-Based Coherency and the set for Weak-Dependency-Based Coherency is due to c_4, which conflicts with c_2 and c_5. This makes it impossible to move c_5 before c_2 in any valid serialization. But for Weak-Dependency-Based Coherency, both c_2 and c_5 are removed from the subsequence and the coherency condition has no obligation for the subsequence to be extended to include c_2 and c_5 with the guarantee that they execute in the same states as in the original execution.

It is evident that there is an inherent tradeoff between the semantics ensured by the valid degraded states and the degradation degree. Usually, the stronger the semantics, the higher the degradation degree. A client can choose the weakest semantics that it can live with in order to get a less degraded view of the system.

For Timestamp-Based Coherency, the system provides an accurate time-travel capability. It provides a strong semantics, but at the expense of potentially higher degradation degree.

For Dependency-Based Coherency, the system provides a *virtual* time-travel capability in that the system has the freedom to assign virtual timestamps to decide in which order commands are executed, while preserving the responses from the system and the final state of the system. Going back to the example of maintaining a source code tree for a large piece of software, clients might notice that the system returns an April version of python, but with the old January version of perl. Those two versions never co-existed as the up-to-date versions in the history of the source code tree. Any real dependencies not captured by *Dep* might lead to anomalies: for example, the April version of python might implicitly depend on new features that are only available in the March version of perl. In this case, the resulting coherent set according to Prefix Serializability might not be consistent in clients' view.

For Weak-Dependency-Based Coherency, the system provides an undo capability to remove the effects of certain commands. In the example of maintaining the source code tree, further assume that at the end of March a command c is executed to collect statistics about the entire source code tree. Such a command will make the April version of

python depend on the March version of perl according to *Dep*, because the execution of the command has the February version of python and March version of perl in its read version set. Even Prefix Serializability will force the state to contain only the February version of python. However, if we use Weak-Dependency-Based Coherency, that command c can simply be "undone", making it possible to return the April version of python. Certainly, clients might observe more anomalies in this case; in particular, the statistics collected by c might not make sense any more; among other things, the statistics might indicate a higher version of perl with a lower version of python.

From a practical point of view, there is also a consideration of overhead in maintaining the metadata to check whether a set of object versions are coherent or not. In all three schemes we present, the metadata for an object version is maintained when the object version is created or read. Dependency-Based Coherency requires the most amount of metadata, which is a superset of what Weak-dependency-Based Coherency requires. The Timestamp-Based Coherency requires only two timestamps. It is therefore possible to support all three at about the same overhead as the one for Dependency-Based Coherency, giving clients the flexibility to choose the right semantics at different occasions.

4. DISCUSSIONS

In this section, we discuss additional issues related to graceful degradation.

4.1 Supporting Degraded Updates

We have so far limited our attention to degraded queries. Extending the semantics to include degraded updates would provide clients with greater functionality in the degraded mode. However, allowing updates on valid degraded states introduces challenges.

The characteristics of failures that lead to degradation influence how updates are handled during degradation. If the failures that lead to degradation are known to be permanent, irrecoverable data loss occurs. In this case, the system can use the mechanisms described in Section 3 to find a most-recent coherent state with respect to a chosen specification. The system then accepts the loss, continues with this coherent state as the current non-degraded state, and proceeds to take updates.

If the failures are transient and data loss is recoverable, degraded updates on older versions of data might create conflicting versions. This leads to new branches from the original linear sequence of versions. Conflicting branches can be merged into a new version when the system recovers from the degraded mode. The merge operation could be application dependent; for example, an application can choose to make the latest version on one of the branches as the new version. In general, the state after a merge operation can be assumed to represent the result of applying a series of updates on some previous valid state. The natural effect of a merge operation is to restore the linear progression of versions by eliminating conflicting branches.

A consequence of branching is that version numbers no longer form a total order, but a partial order. More precisely, versions of an object form a partial order $\prec_p$, where $v_1 \prec_p v_2$ indicates that v_2 evolves from v_1. Partial ordering is transitive. That is, $v_1 \prec_p v_2$ holds if there exists a v such that $v_1 \prec_p v$ and $v \prec_p v_2$ hold. However, If neither $v_1 \prec_p v_2$ nor $v_2 \prec_p v_1$ holds, then v_1 and v_2 belong to two conflicting branches.

We propose a mechanism for representing branched versions and the associated partial order $\prec_p$. Implicitly, we assume that $\prec_p$ is transitive.

1. The first version of an object has version number 1.
2. A new version created from v as the i^{th} branch has the version number $v \circ i$. Here $v \prec_p v \circ i$ holds.
3. A new version created from merging a list of versions $v_1, v_2, \ldots, v_k$ has the version number $[v_1, v_2, \ldots, v_k]$. Here, $v_i \prec_p [v_1, v_2, \ldots, v_k]$ holds for any $1 \leq i \leq k$.

Supporting branched versions have been extensively studied in the context of optimistic replication schemes, such as Coda [11] and Bayou [24]. *Version vectors* are a well-known mechanism to capture branched versions created concurrently by different replicas, enable detection of conflicts, and represent merged branches. However, version vectors are not suitable for representing branched versions arising out of degraded updates, rather than concurrent updates. In optimistic replication, branching happens due to concurrency: two users on two servers might be updating the same file without knowing the action of the other party. Therefore, using version vectors, with one element reserved for each participating server, it is sufficient to encode versions for conflict detection. In our case, arbitrary branching might occur, depending on how massive failures happen and how users choose to proceed. Since the number of dimensions is unknown and unbounded, version vectors are no longer appropriate.

Furthermore, in our case, branching occurs only when degraded. Branches should be merged and resolved appropriately when the system returns to a normal state. In contrast, optimistic replication comes with a weak consistency guarantee.

4.2 Practical Considerations

This paper has focused mainly on the theoretical aspect of the notion of graceful degradation and has attempted to lay out a foundation for further investigation. We discuss some of the practical considerations in this section.

We have looked at how the semantics of valid degraded states could influence the gracefulness of degradation. How we store different versions of objects and how we co-locate object versions also have significant impact on the gracefulness of the degradation. For example, different versions of an object should often be placed on different set of servers, so that it is unlikely that they become unavailable at the same time. Also, object versions that are created by the same command are better co-located because a coherent set often cannot contain only a subset of those object versions while not also including the others. A placement strategy that takes into account those considerations and finds a reasonable compromise is important for achieving better graceful degradation in practice.

Maintaining multiple versions of each object, as well as the extra information associated with each object version, introduces overhead to a system. Depending on the applications and how they are configured, such overhead could be significant and could influence the practicality of the approach. While we have not yet been able to evaluate the

overhead, some recent work in practical (database) systems gives us reasons to believe that our proposed approaches can be made practical.

The most relevant work is on Immortal databases [14, 15]. An immortal database is a transaction time database that retains and provides access to prior states of a database by maintaining old versions of database records. The system is designed to support "time travel" and allow queries on records that are current at some time in the past. As in MVCC, versions are stamped with the times of their updating transactions and transactions are linearized based on the timestamp order.

Immortal databases have been demonstrated to be practical and can be incorporated into a commercial database. Supporting graceful degradation with Timestamp-Based Coherency in Immortal databases is straightforward, assuming the availability of the various index structures. It is also possible to incorporate Dependency-Based and Weak-Dependency-Based Coherency into Immortal databases. In fact, the recent extension [15] to Immortal databases that allow operations to be undone is the same as Weak-Dependency-Based Coherency, although it was proposed to undo "bad" transactions rather than for graceful degradation.

Despite the optimism, there are challenges that remain to be addressed. For example, a truly gracefully degradable system must cope with the unavailability of not only the recent object versions, but also the system metadata that allows the access to object versions. Examples of such metadata include any index structure that maps objects to their versions and their locations. Databases tend to maintain more index structures for performance reasons. Care must be taken so that the system can still offer the normal or degraded semantics when such system metadata becomes unavailable due to excessive failures.

4.3 Degradation and Transaction Aborts

Transactional systems usually have a mechanism to abort transactions. This has the effect of removing any changes related to aborted transactions. Graceful degradation might seem similar to transactional aborts; for example, for Subsequence Linearizability, a coherent state corresponds to one where a set of transactions are undone. However, there is a fundamental difference between them. For graceful degradation, undone transactions during excessive failures have already been *committed*, whereas for traditional transaction mechanisms only uncommitted transactions can be aborted.

5. RELATED WORK

Recognizing the importance of graceful degradation for complex programs, Herlihy and Wing [7] studied the problem of specifying graceful degradation. Their work characterizes graceful degradation as relaxations of application-dependent constraints that the system satisfies. A relaxation lattice method is proposed for specifying the behavior of graceful degradation. Each specification in the lattice is parameterized by a set of constraints, where stronger constraints are more restrictive. The method is used to specify programs that can tolerate faults, timing anomalies, synchronization conflicts, and security breaches.

Graceful degradation has also been considered in many different contexts. Often, it refers to degradation under heavy load, as noted by Brewer [2]. Fox and Brewer [4] propose the notion of *harvest* (the fraction of data reflected in a response) and *yield* (the probability of completing a request) as measures for graceful degradation and discuss the tradeoffs between them. They identify good engineering principles, such as failure isolation through orthogonal mechanisms and replacing hard state with soft state. Our work can also be considered as offering new types of tradeoffs with respect to the fundamental CAP principle, which makes explicit tradeoffs among **C**onsistency, **A**vailability and **P**artition-resilience, while designing distributed systems.

Weaker semantics can sometimes be provided for better system availability. Yu and Vahdat [26] propose the *continuous consistency model*. The model can be thought of as providing guaranteed bounded degradation. Numerical error, order error, and staleness are three types of degradation studied. BFT2F [13] offers graceful degradation with a weaker consistency model called the *Fork** consistency, when the number of Byzantine failures exceeds f, but not $2f$, in a system with $3f+1$ servers. In both cases, the notion of consistency applies to the states on a set of replicas.

Besides database systems that maintain multiple versions, as mentioned in Section 4, creating multiple file versions to improve availability has been in use for decades. Schroeder et al. describe a programming environment with multiple file versions, where they employ clever techniques to reuse a deleted version of a file to create a new one [21]. Over the years, work has been done on versioned file systems [20], whole system snapshots [9], software configuration management [8, 25], and metadata minimization in versioned systems [23].

Orthogonal to versioning, fault containment and isolation (e.g., [18, 10, 19, 22]) are well-known techniques for graceful degradation. For example, Archipelago [10] is a file system consisting of self-contained file servers called islands. Its design is based on the one-island principle, which states that as many operations as possible should involve exactly one island. Similarly, D-GRAID [22] uses file-system semantics to fault-isolate logically related data on separate disks and replicate critical data on multiple disks, resulting in a storage system that gracefully degrades with the number of disk failures.

6. CONCLUDING REMARKS

The study of graceful degradation represents a different mind set than the traditional approach to fault tolerance, where a system is assumed to work within the specified system model. It forces system designers to understand system behaviors when assumptions are invalidated; the result is often a more robust system. For example, with a system that has the notion of valid degraded states, it is easy for the system to recover from permanent loss of some data and proceed with a coherent state. This paper lays out a theoretical foundation for specifying meaningful semantics for graceful degradation in a general distributed system, and provides practical designs that conform to the specifications.

Combining graceful degradation with traditional fault-tolerance mechanisms introduces a new dimension to the design of highly available distributed systems. Graceful degradation removes the cliff that a fault-tolerant system experiences when facing excessive failures. Moreover, it offers the opportunity to weaken semantics for reduced overhead and better performance. For example, one could choose to reduce the level of redundancies in the fault-tolerant mechanism to gain better performance. At one extreme, one could

eliminate redundancies entirely and maintain one copy of each object version, thereby simplifying the system design. The benefits come at the cost of weakened semantics: when failures happen, the system might provide a degraded view.

Graceful degradation is a natural next step to fault tolerance in the design of highly available distributed systems. We believe that it opens up new and potentially fruitful research avenues both in theory and in practice.

7. ACKNOWLEDGMENTS

We would like to thank Phil A. Bernstein, Mihai Budiu, John MacCormick, Dahlia Malkhi, and Nick Murphy for interesting discussions related to the topic of this paper. We also thank John Douceur and the anonymous reviewers for their comments on the paper.

8. REFERENCES

[1] P. A. Bernstein and N. Goodman. Concurrency Control in Distributed Database Systems. *ACM Computing Surveys (CSUR)*, 13(2):185–221, 1981.

[2] E. Brewer. Lessons from giant-scale services. vol. 5, no. 4. pp. 46-55. july/august 2001. *IEEE Internet Computing*, 5(4):46–55, July/August 2001.

[3] B. Fitzpatrick. Power-loss post-mortem, January 2005. http://www.livejournal.com/community/lj_dev/670215.html.

[4] 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.

[5] S. D. Gribble. Robustness in complex systems. In *Proc. of the 8th Workshop on Hop Topics in Operating Systems*, pages 21–26, May 2001.

[6] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[7] M. P. Herlihy and J. M. Wing. Specifying graceful degradation. *IEEE Transactions on Parallel and Distributed Systems*, 2(1):93–104, July 1991.

[8] A. Heydon, R. Levin, T. Mann, and Y. Yu. *Software Configuration Management Using Vesta*. Monographs in Computer Science. Springer Verlag, February 2006.

[9] D. Hitz, J. Lau, and M. Malcolm. File system design for an NFS file server appliance. In *Proc. of the USENIX Winter 1994 Technical Conference*, pages 235–246, 1994.

[10] 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.

[11] 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.

[12] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computer*, C-28(9):690–691, September 1979.

[13] J. Li and D. Mazières. Beyond one-third faulty replicas in byzantine fault tolerant systems. In *Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation (NSDI'2007)*, 2007.

[14] D. Lomet, R. Barga, M. F. Mokbel, G. Shegalov, R. Wang, and Y. Zhu. Immortal DB: transaction time support for SQL server. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 939–941, New York, NY, USA, 2005. ACM Press.

[15] D. Lomet, Z. Vagena, and R. Barga. Recovery from "bad" user transactions. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 337–346, New York, NY, USA, 2006. ACM Press.

[16] C. H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, October 1979.

[17] D. P. Reed. *Naming and Synchronization in a Decentralized Computer System*. Ph.D. Dissertation, Massachusetts Institute of Technology, Cambridge, MA, USA, 1978.

[18] Y. Saito, B. Bershad, and H. M. Levy. Manageability, availability and performance in Porcupine: A highly scalable, cluster-based mail service. In *Proc. of the 17th ACM Symposium on Operating Systems Principles*, pages 1–15, December 1999.

[19] 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.

[20] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, and J. Ofir. Deciding when to forget in the Elephant file system. In *Proc. of the 17th ACM Symposium on Operating Systems Principles*, pages 110–123, 1999.

[21] M. D. Schroeder, D. K. Gifford, and R. M. Needham. A caching file system for a programmer's workstation. In *Proc. of the 10th ACM Symposium on Operating Systems Principles*, pages 25–34, 1985.

[22] 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.

[23] C. A. N. Soules, G. R. Goodson, J. D. Strunk, and G. R. Ganger. Metadata Efficiency in a Comprehensive Versioning File System. In *Proc. of the 2nd FAST*, 2003.

[24] 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.

[25] W. F. Tichy. RCS – A system for version control. *Software Practice and Experience*, 15(7):637–654, 1985.

[26] H. Yu and A. Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Trans. Comput. Syst.*, 20(3):239–182, 2002.