Consistency in Non-Transactional Distributed Storage Systems

PAOLO VIOTTI, EURECOM
MARKO VUKOLIĆ, IBM Research - Zurich

Over the years, different meanings have been associated with the word *consistency* in the distributed systems community. While in the '80s "consistency" typically meant *strong consistency*, later defined also as *linearizability*, in recent years, with the advent of highly available and scalable systems, the notion of "consistency" has been at the same time both weakened and blurred.

In this article, we aim to fill the void in the literature by providing a structured and comprehensive overview of different consistency notions that appeared in distributed systems, and in particular *storage* systems research, in the last four decades. We overview more than 50 different consistency notions, ranging from linearizability to eventual and weak consistency, defining precisely many of these, in particular where the previous definitions were ambiguous. We further provide a partial order among different consistency predicates, ordering them by their semantic "strength," which we believe will be useful in future research. Finally, we map the consistency semantics to different practical systems and research prototypes.

The scope of this article is restricted to non-transactional semantics, that is, those that apply to single storage object operations. As such, our article complements the existing surveys done in the context of transactional, database consistency semantics.

CCS Concepts: • **General and reference** → **Surveys and overviews**; • **Software and its engineering** → **Consistency**; Memory management; • **Information systems** → *Parallel and distributed DBMSs*; *Distributed storage*; • **Theory of computation** → *Invariants*; *Program specifications;*

Additional Key Words and Phrases: Consistency, distributed storage

ACM Reference Format:
Paolo Viotti and Marko Vukolić. 2016. Consistency in non-transactional distributed storage systems. ACM Comput. Surv. 49, 1, Article 19 (June 2016), 34 pages.
DOI: http://dx.doi.org/10.1145/2926965

1. INTRODUCTION

Faced with the inherent challenges of failures, communication/computation asynchrony, and concurrent access to shared resources, distributed system designers have continuously sought to hide these fundamental concerns from users by offering abstractions and semantic models of various strength. At first glance, the ultimate goal of a distributed system is seemingly simple, as it should ideally be just a fault-tolerant and more scalable version of a centralized system. Namely, an ideal distributed system should leverage distribution and replication to boost availability by masking failures, provide scalability, and/or reduce latency, but maintain the simplicity of use of a centralized system—and, notably, its *consistency*—providing the illusion of sequential access.

This research was supported in part by the EU projects CloudSpaces (FP7-317555) and SUPERCLOUD (Horizon 2020 program, grant no. 643964).
Authors' addresses: P. Viotti, EURECOM, 450 Route des Chappes, 06904 Biot Sophia Antipolis cedex, France; email: viotti@eurecom.fr; M. Vukolić, IBM Research - Zurich, Säumerstrasse 4, CH-8803 Rüschlikon, Switzerland; email: mvu@zurich.ibm.com.

DOI: http://dx.doi.org/10.1145/2926965

Such *strong* consistency criteria can be found in early seminal works that paved the way for modern storage systems (e.g., Lamport [1978, 1986a]), as well as in the subsequent advances in defining general, practical correctness conditions, such as *linearizability* [Herlihy and Wing 1990].

Unfortunately, the goals of high availability and strong consistency, in particular linearizability, have been identified as mutually conflicting in many practical circumstances. Negative theoretical results and lower bounds, such as the FLP impossibility result [Fischer et al. 1985] and the CAP theorem [Gilbert and Lynch 2002], shaped the design space of distributed systems. As a result, distributed system designers have to either give up the idealized goals of scalability and availability or relax consistency.

In recent years, the rise of commercial Internet-scale wide-area computing caused system designers to prefer availability over consistency, leading to the advent of weak and eventual consistency [Terry et al. 1994; Saito and Shapiro 2005; Vogels 2008]. Consequently, much research has been focusing on attaining a better understanding of those weaker semantics [Bailis and Ghodsi 2013], but also on adapting [Bailis et al. 2014] or dismissing and replacing stronger ones [Helland 2007]. Along this line of research, tools have been conceived in order to deal with consistency at the level of programming languages [Alvaro et al. 2011], data objects [Shapiro et al. 2011a; Burckhardt et al. 2012], or data flows [Alvaro et al. 2014].

Today, however, after roughly four decades of intensive and exciting research on various flavors of consistency, we lack a structured and comprehensive overview of different consistency notions that appeared in distributed systems research, and *storage* systems research in particular.

This article aims to help fill this void by giving an overview of over 50 different consistency notions, ranging from linearizability to eventual and weak consistency, defining precisely many of these, in particular where the previous definitions were ambiguous. We further provide a partial order among different consistency notions, ordering them by their semantic "strength," which we believe will be useful in further research. Finally, we map the consistency semantics to different practical systems and research prototypes. The scope of this article is restricted to non-transactional semantics that apply to single storage object operations. We focus on non-transactional storage systems as they have become increasingly popular in recent years due to their simple implementations and good scalability. As such, our article complements the existing surveys done in the context of transactional, database consistency semantics (see, e.g., Adya [1999]), which we omit for space limitations.

This survey is organized as follows. In Section 2, we define our model of a distributed system and set up the framework for reasoning about different consistency semantics. In order to ensure the broadest coverage of our work, we model the distributed system as asynchronous, that is, without predefined constraints on timing of computation and communication. Our framework, which we derive from the work of Burckhardt [2014], captures the dynamic aspects of a distributed system through *histories* and *abstract executions* of such systems. We define an execution as a set of actions (i.e., *operations*) invoked by some processes on the storage objects through their interface. To analyze executions, we adopt the notion of history, that is, the set of operations of a given execution. Leveraging the information attached to the history, we are able to properly capture the intrinsic complexity of executions. Namely, we can group and relate operations according to their features (e.g., by the processes and objects they refer to, and by their timings) or by the dynamic relationships established during executions (e.g., causality). Additionally, abstract executions augment histories with orderings of operations that account for the resolution of write conflicts and their propagation within the storage system.

Section 3 brings the main contribution of our article: a survey of more than 50 different consistency semantics proposed in the context of non-transactional distributed

storage systems.[1] We define many of these models employing the framework specified in Section 2, that is, using declarative compositions of logic predicates over graph entities. In turn, these definitions enable us to establish the hierarchical partial order of consistency semantics according to their semantic strengths (given in Figure 1 of Section 3). For better readability, we also loosely classify consistency semantics into 10 *families*, which group them by their common traits.

We discuss our work in the context of related consistency surveys in Section 4 and conclude in Section 5. We further complement our survey with a summary of all consistency predicates defined in this work (Appendix A). In addition, for all consistency models mentioned in this work, we provide references to their original, primary definitions, as well as pointers to research papers that propose related implementations (Appendix B). Specifically, we reference implementations that appeared in recent proceedings of the most relevant venues. We believe that this is a useful contribution on its own, as it will allow distributed systems researchers and, in particular, students to navigate more easily through the very large number of research papers that deal with different subtleties of consistency.

2. SYSTEM MODEL

In this section, we specify the main notions behind the reasoning about consistency semantics carried out in the rest of this article. We rely on the concurrent objects abstraction, as presented by Lynch and Tuttle [1989] and by Herlihy and Wing [1990], for the definitions of fundamental "static" elements of the system, such as objects and processes. Moreover, to reason about dynamic behaviors of the system (i.e., executions), we build upon the mathematical framework laid out in Burckhardt [2014].

2.1. Preliminaries

Objects and Processes. We consider a distributed system consisting of a finite set of *processes*, modeled as I/O automata [Lynch and Tuttle 1989], interacting with *shared* (or *concurrent*) *objects* through a fully connected asynchronous communication network. Unless stated otherwise, processes and shared objects (or, simply, objects) are *correct*; that is, they do not fail. Each process and object is identified by a unique identifier. We define *ProcessIds* as the set of all process identifiers and *ObjectIds* as the set of all object identifiers.

Additionally, each object has a unique *object type*. Depending on the type, the object can assume *values* belonging to a defined domain denoted by *Values*,[2] and it supports a set of primitive *operation types* (i.e., $OpTypes = \{rd, wr, inc, \ldots\}$) that provide the only means to manipulate that object. For simplicity and without loss of generality, unless specified otherwise, in this work we further classify operations as either *reads* (*rd*) or *writes* (*wr*). Namely, we model as a write (or *update*) any operation that modifies the value of the object, while, conversely, reads return to the caller the current value held by the object's replica without causing any change to it. We adopt the term *object replicas*, or simply *replicas*, to refer to the different copies of a same named shared object maintained in the storage system for fault tolerance or performance enhancement. Ideally, replicas of the same shared object should hold the same data at any time. The coordination protocols among replicas are, however, determined by the implementation of the shared object.

[1]Note that, while this article focuses on a survey of consistency semantics proposed in the context of distributed storage, our approach maintains generality as our consistency definitions are applicable to other replicated data structures beyond distributed storage.

[2]For readability, we adopt a notation in which a set *Values* is implicitly parameterized by object type.

Time. Unless specified otherwise, we assume an asynchronous computation and communication model, with no bounds on computation and communication latencies. However, when describing certain consistency semantics, we will be using terms such as *recency* or *staleness*. Such terms relate to the concept of *real time*, that is, an ideal and global notion of time that we use to reason about histories a posteriori, although it is not accessible by processes during executions. We refer to the real-time domain as *Time*, which we model as the set of positive real numbers, that is, $\mathcal{R}^+$.

2.2. Operations, Histories, and Abstract Executions

Operations. We describe an operation issued by a process on a shared object as the tuple $(proc, type, obj, ival, oval, stime, rtime)$, where:

—$proc \in ProcessIds$ is the id of the process invoking the operation.
—$type \in OpTypes$ is the operation type.
—$obj \in ObjectIds$ is the id of the object on which the operation is invoked.
—$ival \in Values$ is the operation input value.
—$oval \in Values \cup \{\nabla\}$ is the operation output value, or ∇ if the operation does not return.
—$stime \in Time$ is the operation invocation time.
—$rtime \in Time \cup \{\Omega\}$ is the operation return time, or Ω if the operation does not return.

By convention, we use the special value $\sqcup \in Values$ to represent the input value (i.e., $ival$) of reads and, possibly, the return value (i.e., $oval$) of writes. For simplicity, given operation op, we will use the notation $op.par$ to access its parameter named par as expressed in the tuple (e.g., $op.type$ represents its type, and $op.ival$ its input value).

Histories. A *history* H is a set of operations. Intuitively, a history contains all operations invoked in a given execution. We further denote by $H|_{wr}$ ($H|_{rd}$, respectively) the set of write (read, respectively) operations of a given history H (e.g., $H|_{wr} = \{op \in H : op.type = wr\}$).

We further define the following relations on elements of a history:[3]

—rb (*returns-before*) is a natural partial order on H based on real-time precedency. Formally: $rb \triangleq \{(a, b) : a, b \in H \wedge a.rtime < b.stime\}$.
—ss (*same-session*) is an equivalence relation on H that groups pairs of operations invoked by the same process—we say such operations belong to the same *session*. Formally: $ss \triangleq \{(a, b) : a, b \in H \wedge a.proc = b.proc\}$.
—so (*session order*) is a partial order defined as $so \triangleq rb \cap ss$.
—ob (*same-object*) is an equivalence relation on H that groups pairs of operations invoked on the same object. Formally: $ob \triangleq \{(a, b) : a, b \in H \wedge a.obj = b.obj\}$.
—$concur$ is the symmetric binary relation designating all pairs of real-time *concurrent* operations invoked on the same object. Formally: $concur \triangleq ob \setminus rb$.

For $(a, b) \in rel$, we sometimes alternatively write $a \xrightarrow{rel} b$. We further denote by rel^{-1} the inverse relation of rel. For the sake of a more compact notation, we use binary relation projections. For instance, $rel|_{wr \to rd}$ identifies all pairs of operations belonging to rel consisting of a write and a read operation. Furthermore, if rel is an equivalence relation, we adopt the notation $a \approx_{rel} b \triangleq [a \xrightarrow{rel} b]$. We recall that an equivalence relation rel on set H partitions H into equivalence classes $[a]_{rel} = \{b \in H : b \approx_{rel} a\}$. We write $H/\approx_{rel}$ to denote the set of equivalence classes determined by rel.

[3]For better readability, we implicitly assume relations are parameterized by a history.

We complement the *concur* relation with the function $Concur : H \rightarrow 2^H$ to denote the set of write operations concurrent with a given operation:

$$Concur(a) \triangleq \{b \in H|_{wr} : (a, b) \in concur\}. \quad (1)$$

Abstract Executions. We model system executions using the concept of *abstract execution*, following Burckhardt [2014]. An abstract execution is a multigraph $A = (H, vis, ar)$ built on a given history H, which it complements with two relations on elements of H, that is, vis and ar. Whereas histories describe the observable outcomes of executions, vis and ar, intuitively, capture the nondeterminism of the asynchronous environment (e.g., message delivery order), as well as implementation-specific constraints (e.g., conflict resolution policies). In other words, vis and ar determine the relations between pairs of operations in a history that explain and justify its outcomes. More specifically:

—vis (*visibility*) is an acyclic natural relation that accounts for the propagation of write operations. Intuitively, a is visible to b (i.e., $a \xrightarrow{vis} b$) means that the effects of a are visible to the process invoking b (e.g., b may read a value written by a). Two write operations are *invisible* to each other if they are not ordered by vis.
—ar (*arbitration*) is a *total* order on operations of the history that specifies how the system resolves conflicts due to concurrent and invisible operations. In practice, such total order can be achieved in various ways: through the adoption of a distributed timestamping [Lamport 1978] or consensus protocol [Birman et al. 1991; Hadzilacos and Toueg 1994; Lamport 2001], using a centralized serializer, or using a deterministic conflict resolution policy.

Depending on constraints expressed by vis, during an execution, processes may observe different orderings of write operations, which we call *serializations*.

We further define the *happens-before* order (hb) as the transitive closure of the union of so and vis, denoted by

$$hb \triangleq (so \cup vis)^+. \quad (2)$$

2.3. Replicated Data Types and Return Value Consistency

Rather than defining the current system state as a set of values held by shared objects, following Burckhardt [2014], we employ a graph abstraction called (operation) *context*, which encodes the information of an abstract execution A, taking a projection on visibility (vis) with respect to a given operation op. Formally, given $\mathcal{C}$ as the set of contexts of all operations in a given abstract execution A, we define the context of an operation op as

$$C = cxt(A, op) \triangleq A|_{op, vis^{-1}(op), vis, ar}. \quad (3)$$

Further, we adopt the concept of *replicated data type* [Burckhardt 2014] to define the type of shared object implemented in the distributed system (e.g., read/write register, counter, set, queue, etc.). For each replicated data type, a function $\mathcal{F}$ specifies the set of intended return values of an operation $op \in H$ in relation to its context, that is, $\mathcal{F}(op, cxt(A, op))$. Using $\mathcal{F}$, we can define *return value consistency* as

$$\text{RVAL}(\mathcal{F}) \triangleq \forall op \in H : op.oval \in \mathcal{F}(op, cxt(A, op)). \quad (4)$$

Essentially, return value consistency is a predicate on abstract executions that guarantees that the return value of any given operation of that execution will belong to the set of its intended return values.

Given operation $b \in H$ and its context $cxt(A, b)$, we let $a = prec(b)$ be the (unique) latest operation preceding b in ar, such that $a.oval \neq \nabla \wedge a \in H|_{wr} \cap vis^{-1}(b)$. In other

words, $prec(b)$ is the last write visible to b according to the ordering specified by ar. If no such preceding operation exists (e.g., if b is the first operation of the execution according to ar), by convention $prec(b).ival$ is a default value equal to $\perp$.

In this article, we adopt the read/write register (i.e., read/write storage) as reference replicated data type, which is defined by the following intended return value function:

$$\mathcal{F}_{reg}(op, cxt(A, op)) = prec(op).ival. \tag{5}$$

Note that, while the focus of this survey is on read/write storage, the consistency predicates defined in this article take $\mathcal{F}$ as a parameter, and therefore directly extend to other replicated data types.

2.4. Consistency Semantics

Following Burckhardt [2014], we define *consistency semantics*, sometimes also called *consistency guarantees*, as conditions on attributes and relations of abstract executions, expressed as first-order logic predicates. We write $A \models \mathcal{P}$ if the consistency predicate $\mathcal{P}$ is true for abstract execution A. Hence, defining a consistency model amounts to collecting all the required consistency predicates and then specifying that histories must be justifiable by at least an abstract execution that satisfies them all.

Formally, given history H and $\mathcal{A}$ as the set of all possible abstract executions on H, we say that history H satisfies some consistency predicates $\mathcal{P}_1, \ldots \mathcal{P}_n$ if it can be extended to some abstract execution that satisfies them all:

$$H \models \mathcal{P}_1 \wedge \cdots \wedge \mathcal{P}_n \Leftrightarrow \exists A \in \mathcal{A} : \mathcal{H}(A) = H \wedge A \models \mathcal{P}_1 \wedge \cdots \wedge \mathcal{P}_n. \tag{6}$$

In the previous notation, given the abstract execution $A = (H, vis, ar)$, $\mathcal{H}(A)$ denotes H.

3. NON-TRANSACTIONAL CONSISTENCY SEMANTICS

In this section, we analyze and survey the consistency semantics of systems that adopt single operations as their primary operational constituent (i.e., non-transactional consistency semantics). The consistency models described in the rest of the article appear in Figure 1, a comprehensive graph that proposes a partial ordering of consistency semantics according to their semantic strength, as well as a more loosely defined clustering into *families* of consistency models. This classification draws both from strength of different consistency semantics and from the underlying common factors that underpin their definitions.

In the remainder of this section, we examine each family of consistency semantics. Section 3.1 introduces linearizability and other strong consistency models, while in Section 3.2 we consider eventual and weak consistency. Next we analyze PRAM and sequential consistency (Section 3.3) and, in Section 3.4, the models based on the concept of session. Section 3.5 proposes an overview of consistency semantics explicitly dealing with causality, while in Section 3.6 we study staleness-based models. This is followed by an overview of fork-based models (Section 3.7). Sections 3.8 and 3.9 respectively deal with tunable and per-object semantics. Finally, we survey the family of consistency models based on synchronization primitives (Section 3.10).

3.1. Linearizability and Related "Strong" Consistency Semantics

The gold standard and the central consistency model for non-transactional systems is **linearizability**, defined by Herlihy and Wing [1990]. Roughly speaking, linearizability is a correctness condition that establishes that each operation shall appear to be applied instantaneously at a certain point in time between its invocation and its response.

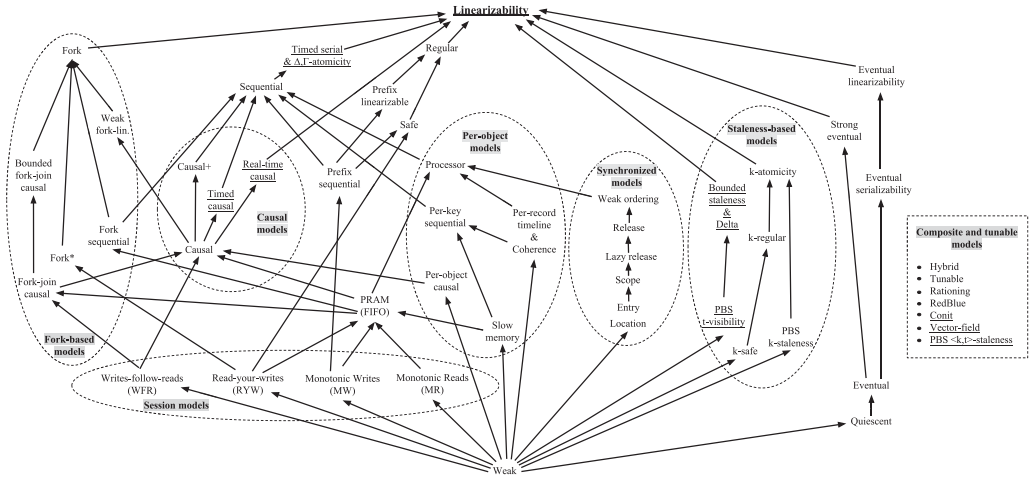

Fig. 1. Hierarchy of non-transactional consistency models. A directed edge from consistency semantics A to consistency semantics B means that any execution that satisfies B also satisfies A. Underlined models explicitly reason about timing guarantees.

Linearizability, often informally dubbed *strong consistency*,[4] has been for long regarded as the ideal correctness condition at which distributed storage implementations should aim. Linearizability features a *locality* property: a composition of linearizable objects is itself linearizable—hence, linearizability enables modular design and verification.

Although very intuitive to understand, the strong semantics of linearizability make it challenging to implement. In this regard, Gilbert and Lynch [2002] formally proved the *CAP* theorem, an assertion informally presented in previous works [Johnson and Thomas 1975; Davidson et al. 1985; Coan et al. 1986; Brewer 2000] that binds linearizability to the ability of a system to maintain a nontrivial level of availability when confronted with network partitions. In a nutshell, the *CAP* theorem states that in the presence of network partitions, a distributed storage system has to sacrifice either availability or linearizability.

Burckhardt [2014] breaks down linearizability into three components:

$$\textsc{Linearizability}(\mathcal{F}) \triangleq \textsc{SingleOrder} \wedge \textsc{RealTime} \wedge \textsc{RVal}(\mathcal{F}), \tag{7}$$

where

$$\textsc{SingleOrder} \triangleq \exists H' \subseteq \{op \in H : op.oval = \nabla\} : vis = ar \setminus (H' \times H) \tag{8}$$

and

$$\textsc{RealTime} \triangleq rb \subseteq ar. \tag{9}$$

In other words, SingleOrder imposes a single global order that defines both vis and ar, whereas RealTime constrains arbitration (ar) to comply to the returns-before partial ordering (rb). Finally, RVal($\mathcal{F}$) specifies the return value consistency of a replicated data type. We recall that, as per Equation (5), in case of read/write storage, this is the value written by the last write (according to ar) visible to a given read operation rd.

A definition tightly related to that one of linearizability had been previously provided by Lamport [1986b] for the **atomic** register semantic. Lamport describes a

[4]Note that the adjective "strong" has also been used in the literature to identify indistinctly *linearizability* and *sequential consistency* (which we define in Section 3.3), as they both entail single-copy semantics and require that a single ordering of operations be observed by all processes.

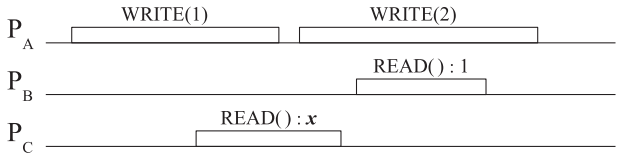

Fig. 2. An execution exhibiting read-write concurrency (real-time flows from left to right). The register is initialized to 0. Atomic (linearizable) semantics allow x to be 0 or 1. Regular semantics allow x to be 0, 1, or 2. With safe semantics x may be any value.

single-writer multireader (SWMR) shared register to be atomic if and only if each read operation not overlapping a write returns the last value actually written on the register, and the read values are the same as if the operations had been performed sequentially (i.e., without overlapping). Essentially, this definition implies the existence of a point in time (the *linearization point*) at which each operation is actually applied on the shared register.[5] It is easy to show that atomicity and linearizability are equivalent for read-write registers. However, linearizability is a more general condition designed for generic shared data structures that allow for a broader set of operational semantics than those offered by registers. Besides atomic registers, Lamport [1986b] defines two slightly weaker semantics for SWMR registers: **safe** and **regular**. In the absence of read-write concurrency, they both guarantee that a read returns the last written value, exactly like the atomic semantics. The difference between the three resides in the allowed set of return values for a read operation concurrent with a write. Namely, with a safe register, a read concurrent with some write may return any value. On the other hand, with a regular register, a read operation concurrent with some writes may return either the value written by the most recent complete write or a value written by a concurrent write. This difference is illustrated in Figure 2.

Formally, regular and safe semantics can be defined as follows:

$$\textsc{Regular}(\mathcal{F}) \triangleq \textsc{SingleOrder} \wedge \textsc{RealTimeWrites} \wedge \textsc{RVal}(\mathcal{F}) \tag{10}$$

$$\textsc{Safe}(\mathcal{F}) \triangleq \textsc{SingleOrder} \wedge \textsc{RealTimeWrites} \wedge \textsc{SeqRVal}(\mathcal{F}), \tag{11}$$

where

$$\textsc{RealTimeWrites} \triangleq rb|_{wr \to op} \subseteq ar \tag{12}$$

is a restriction of real-time ordering only for writes (preceding reads or other writes), and

$$\textsc{SeqRVal}(\mathcal{F}) \triangleq \forall op \in H : Concur(op) = \emptyset \Rightarrow op.oval \in \mathcal{F}(op, cxt(A, op)), \tag{13}$$

which restricts the return value consistency only to read operations that are not concurrent with any write.

3.2. Weak and Eventual Consistency

At the opposite end of the consistency spectrum lies **weak** consistency. Although this term has been traditionally used in the literature to identify any consistency model weaker than sequential consistency, recent works [Vogels 2008; Bermbach and Kuhlenkamp 2013] associate it to a more specific albeit rather vague definition: a weakly consistent system does not guarantee that reads return the most recent value written, and several (often underspecified) requirements have to be satisfied for a value

[5]The existence of an instant at which each operation becomes atomically *visible* had originally been postulated by Lamport [1983].

to be returned. In effect, weak consistency does not provide ordering guarantees—hence, no synchronization protocol is actually required. Even though this model might seem to have limited usability, it is in fact implemented in situations in which having a synchronization protocol would be too costly and a fortuitous exchange of information between replicas can be good enough. For example, a typical use case for weak consistency is the relaxed caching policies that can be applied across various tiers of a web application, or even the cache implemented in web browsers.

Eventual consistency is a slightly stronger notion than weak consistency. Namely, under eventual consistency, replicas converge toward identical copies in the absence of further updates. In other words, if no new write operations are invoked on the object, *eventually* all reads will return the same value. Eventual consistency was first defined by Terry et al. [1994] and then further popularized more than a decade later by Vogels [2008] with the advent of highly available storage systems (i.e., *AP* systems in the CAP theorem parlance). Eventual consistency is especially suited in contexts where coordination is not practical or too expensive (e.g., in mobile and wide area settings) [Saito and Shapiro 2005]. Despite its wide adoption, eventual consistency leaves to the application programmer the burden of dealing with transient *anomalies* (i.e., behaviors deviating from that of an ideal linearizable execution). Hence, a quite large body of recent work has been aiming to achieve a better understanding of its subtle implications [Bermbach and Tai 2011; Bernstein and Das 2013; Bailis and Ghodsi 2013; Bailis et al. 2014]. At its core, eventual consistency constrains replicas' eventual state (i.e., their *convergence*): in fact, it does not provide any guarantees about recency and ordering of operations. Burckhardt [2014] proposes a formal definition of eventual consistency:

$$\begin{aligned}&\textsc{EventualConsistency}(\mathcal{F})\\ &\quad\triangleq \textsc{EventualVisibility} \wedge \textsc{NoCircularCausality} \wedge \textsc{RVal}(\mathcal{F}),\end{aligned} \tag{14}$$

where

$$\begin{aligned}\textsc{EventualVisibility} \triangleq \forall a \in H, \forall [f] \in H/\approx_{ss}:\\ |\{b \in [f] : (a \xrightarrow{rb} b) \wedge (a \overset{vis}{\not\rightarrow} b)\}| < \infty\end{aligned} \tag{15}$$

and

$$\textsc{NoCircularCausality} \triangleq \mathit{acyclic(hb);} \tag{16}$$

that is, the acyclic projection of hb, defined in Equation (2). EventualVisibility mandates that, eventually, operation op will be visible to another operation op' invoked after the completion of op.

In an alternative attempt at clarifying the definition of eventual consistency, Shapiro et al. [2011a] identify the following properties from replicas' viewpoint:

—*Eventual delivery*: if some correct replica applies a write operation op, op is eventually applied by all correct replicas.
—*Convergence*: all correct replicas that have applied the same write operations eventually reach equivalent state.
—*Termination*: all operations complete.

To this definition of eventual consistency, Shapiro et al. [2011a] add the following constraint:

—*Strong convergence*: all correct replicas that have applied the same write operations *have* equivalent state.

In other words, this last property guarantees that any two replicas that have applied the same (possibly unordered) set of writes will hold the same data. A storage system enforcing both eventual consistency and strong convergence is said to implement **strong eventual** consistency.

We capture strong convergence from the perspective of read operations by requiring that reads that have the identical sets of visible writes return the same values:

$$\textsc{StrongConvergence} \triangleq \forall a, b \in H|_{rd} : vis^{-1}(a)|_{wr} = vis^{-1}(b)|_{wr} \Rightarrow a.oval = b.oval. \quad (17)$$

Then, strong eventual consistency can be defined as

$$\textsc{StrongEventualConsistency}(\mathcal{F}) \triangleq \textsc{EventualConsistency}(\mathcal{F}) \wedge \textsc{StrongConvergence}. \quad (18)$$

Quiescent consistency [Herlihy and Shavit 2008] requires that if an object stops receiving updates (i.e., becomes quiescent), then the execution is equivalent to some sequential execution containing only complete operations. Although this definition resembles eventual consistency, it does not guarantee termination: a system that does not stop receiving updates will not reach quiescence, thus preventing replica convergence. Following Burckhardt [2014], we formally define quiescent consistency as

$$\textsc{QuiescentConsistency}(\mathcal{F}) \triangleq |H|_{wr}| < \infty \Rightarrow \exists C \in \mathcal{C} : \forall [f] \in H/\approx_{ss} : |\{op \in [f] : op.oval \notin \mathcal{F}(op, C)\}| < \infty. \quad (19)$$

3.3. PRAM and Sequential Consistency

Pipeline RAM (**PRAM** or FIFO) consistency [Lipton and Sandberg 1988] prescribes that all processes see write operations issued by a given process in the same order as they were invoked by that process. On the other hand, processes may observe writes issued by different processes in different orders. Thus, no global total ordering is required. However, the writes from any given process (session) must be serialized in order, as if they were in a pipeline—hence the name. We define PRAM consistency by requiring the visibility partial order to be a superset of session order:

$$\text{PRAM} \triangleq so \subseteq vis. \quad (20)$$

As proved by Brzezinski et al. [2003], PRAM consistency is ensured if and only if the system provides read-your-write, monotonic reads, and monotonic writes guarantees, which we will introduce in Section 3.4.

In a storage system implementing **sequential** consistency, all operations are serialized in the same order on all replicas, and the ordering of operations determined by each process is preserved. Formally:

$$\textsc{SequentialConsistency}(\mathcal{F}) \triangleq \textsc{SingleOrder} \wedge \text{PRAM} \wedge \textsc{RVal}(\mathcal{F}). \quad (21)$$

Thus, sequential consistency, first defined in Lamport [1979], is a guarantee of ordering rather than recency. Like linearizability, sequential consistency enforces a common global order of operations. Unlike linearizability, sequential consistency does not require real-time ordering of operations across different sessions: only the real-time ordering of operations invoked by the same process is preserved (as in PRAM consistency).[6] A quantitative comparison of the power and costs involved in the

[6]In Section 3.10, we present *processor* consistency: a model whose semantic strength stands between those of PRAM and sequential consistency.

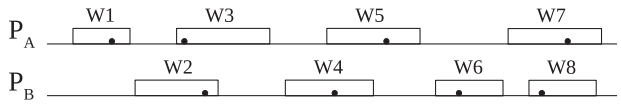

Fig. 3. An execution with processes issuing write operations on a shared object. Black spots are the chosen linearization points.

implementation of sequential consistency and linearizability is presented by Attiya and Welch [1994].

Figure 3 shows an execution featuring two processes issuing write operations on a shared object. Let us suppose that the two processes also continuously perform read operations. Each process will observe a certain serialization of the write operations. If we were to assume that the system respects PRAM consistency, those two processes might observe, for instance, the following two serializations:

$$S_{P_A}: \quad W1 \quad W2 \quad W3 \quad W5 \quad W4 \quad W7 \quad W6 \quad W8 \tag{S.1}$$

$$S_{P_B}: \quad W1 \quad W3 \quad W5 \quad W7 \quad W2 \quad W4 \quad W6 \quad W8. \tag{S.2}$$

If the system implemented sequential consistency, then S_{P_A} would be equal to S_{P_B} and it would respect the ordering of operations imposed by each writing process. Thus, any of (S.1) or (S.2) would be acceptable. On the other hand, assuming the system implements linearizability, and assigning linearization points as indicated by the points in Figure 3, (S.3) would be the only allowed serialization:

$$S_{Lin}: \quad W1 \quad W3 \quad W2 \quad W4 \quad W5 \quad W6 \quad W8 \quad W7. \tag{S.3}$$

3.4. Session Guarantees

Session guarantees were first described by Terry et al. [1994]. Although originally defined in connection to *client* sessions, session guarantees may as well apply to situations in which the concept of session is more loosely defined and it just refers to a specific process's point of view on the execution. We note that previous works in the literature have classified session guarantees as *client-centric models* [Tanenbaum and van Steen 2007].

Monotonic reads states that successive reads must reflect a non-decreasing set of writes. Namely, if a process has read a certain value v from an object, any successive read operation will not return any value written before v. Intuitively, a read operation can be served only by those replicas that have executed all write operations whose effects have already been observed by the requesting process. In effect, we can represent this by saying that, given three operations $a, b, c \in H$, if $a \xrightarrow{\text{vis}} b$ and $b \xrightarrow{\text{so}} c$, where b and c are read operations, then $a \xrightarrow{\text{vis}} c$; that is, the transitive closure of vis and so is included in vis:

$$\begin{aligned}\textsc{MonotonicReads} &\triangleq \forall a \in H, \forall b, c \in H|_{rd} : a \xrightarrow{\text{vis}} b \wedge b \xrightarrow{\text{so}} c \Rightarrow a \xrightarrow{\text{vis}} c \\ &\triangleq (vis; so|_{rd \to rd}) \subseteq vis. \end{aligned} \tag{22}$$

The **Read-your-writes** guarantee (also called read-my-writes [Terry et al. 2013; Burckhardt 2014]) requires that a read operation invoked by a process can only be carried out by replicas that have already applied all writes previously invoked by the same process:

$$\begin{aligned}\textsc{ReadYourWrites} &\triangleq \forall a \in H|_{wr}, \forall b \in H|_{rd} : a \xrightarrow{\text{so}} b \Rightarrow a \xrightarrow{\text{vis}} b \\ &\triangleq so|_{wr \to rd} \subseteq vis. \end{aligned} \tag{23}$$

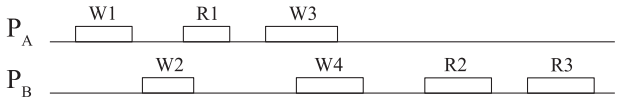

Fig. 4. An execution with processes issuing read and write operations on a shared object.

Let us assume that two processes issue read and write operations on a shared object as in Figure 4.

Given such execution, P_A and P_B could observe the following serializations, which satisfy the read-your-write guarantee but not PRAM consistency:

$$S_{P_A}: \quad W1 \quad W3 \quad W4 \quad W2 \tag{S.4}$$

$$S_{P_B}: \quad W2 \quad W4 \quad W3 \quad W1. \tag{S.5}$$

We note that some works in the literature refer to *session consistency* as a special case of read-your-writes consistency that can be attained through *sticky* client sessions, that is, those sessions in which the process always invokes operations on a given replica.

In a system that ensures **monotonic writes,** a write is only performed on a replica if the replica has already performed all previous writes of the same session. In other words, replicas shall apply all writes belonging to the same session according to the order in which they were issued:

$$\textsc{MonotonicWrites} \triangleq \forall a, b \in H|_{wr} : a \xrightarrow{\text{so}} b \Rightarrow a \xrightarrow{\text{ar}} b \triangleq so|_{wr \to wr} \subseteq ar. \tag{24}$$

Writes-follow-reads, sometimes called *session causality*, is somewhat the converse concept of read-your-write guarantee as it ensures that writes made during the session are ordered after any writes made by any process on any object whose effects were seen by previous reads in the same session:

$$\textsc{WritesFollowReads} \triangleq \forall a, c \in H|_{wr}, \forall b \in H|_{rd} : a \xrightarrow{\text{vis}} b \wedge b \xrightarrow{\text{so}} c \Rightarrow a \xrightarrow{\text{ar}} c$$
$$\triangleq (vis; so|_{rd \to wr}) \subseteq ar. \tag{25}$$

We note that some of the session guarantees embed specific notions of causality, and that in fact, as proved by Brzezinski et al. [2004], causal consistency, which we describe next, requires and includes them all.

3.5. Causal Models

The commonly accepted notion of potential causality in distributed systems has been enclosed in the definition of the *happened-before* relation introduced by Lamport [1978]. According to this relation, two operations a and b are ordered if (1) they are both part of the same thread of execution, (2) b reads a value written by a, or (3) they are related by a transitive closure leveraging (1) and/or (2). This notion, originally defined in the context of message passing systems, has been translated to a consistency condition for shared-memory systems by Hutto and Ahamad [1990]. The potential causality relation establishes a partial order over operations, which we represent as *hb* in Equation (2). Hence, while operations that are potentially causally[7] related must be seen by all processes in the same order, operations that are not causally related (i.e., causally concurrent) may be observed in different orders by different processes. In other words, **causal** consistency dictates that all replicas agree on the ordering of causally related

[7]While the most appropriate terminology would be "potential causality," for simplicity, hereafter we will use "causality."

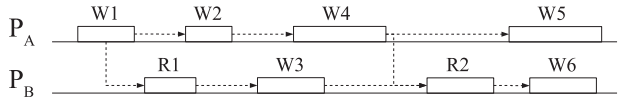

Fig. 5. An execution with processes issuing operations on a shared object. Arrows express causal relationships between operations.

operations [Hutto and Ahamad 1990; Ahamad et al. 1995; Mahajan et al. 2011]. This can be expressed as the conjunction of two predicates [Burckhardt 2014]:

—CausalVisibility $\triangleq hb \subseteq vis$
—CausalArbitration $\triangleq \overline{hb} \subseteq ar$

Hence, causal consistency is defined as

$$\text{Causality}(\mathcal{F}) \triangleq \text{CausalVisibility} \wedge \text{CausalArbitration} \wedge \text{RVal}(\mathcal{F}). \tag{26}$$

Figure 5 represents an execution with two processes writing and reading the value of a shared object, with arrows indicating the causal relationships between operations. Assuming the execution respects PRAM but not causal consistency, we might have the following serializations:

$$S_{P_A}: \quad W1 \quad W2 \quad W4 \quad W5 \quad W3 \quad W6 \tag{S.6}$$

$$S_{P_B}: \quad W3 \quad W6 \quad W1 \quad W2 \quad W4 \quad W5. \tag{S.7}$$

Otherwise, with causal consistency (which implies PRAM), we could have obtained these serializations:

$$S_{P_A}: \quad W1 \quad W3 \quad W2 \quad W4 \quad W5 \quad W6 \tag{S.8}$$

$$S_{P_B}: \quad W1 \quad W2 \quad W3 \quad W4 \quad W6 \quad W5. \tag{S.9}$$

Recent work by Bailis et al. [2012] promotes the use of explicit application-level causality, which is a subset of potential causality,[8] for building highly available distributed systems that would entail less overhead in terms of coordination and metadata maintenance. Furthermore, an increasing body of research has been drawing attention on causal consistency, considered as an optimal tradeoff between user-perceived correctness and coordination overhead, especially in mobile or geo-replicated applications [Lloyd et al. 2011; Bailis et al. 2013; Zawirski et al. 2015].

Causal+ (or convergent causal) consistency [Lloyd et al. 2011] mandates, in addition to causal consistency, that all replicas should eventually and independently agree on conflict resolution. In fact, causally concurrent write operations may generate conflicting outcomes that in convergent causal consistent systems are handled in the same way by commutative and associative functions. Essentially, causal+ strengthens causal consistency with strong convergence (see Equation (17)), which mandates that all correct replicas that have applied the same write operations have equivalent state. In a sense, causal+ consistency augments causal consistency with strong convergence, in the vein that strong eventual consistency [Shapiro et al. 2011a] strengthens eventual consistency. Hence, causal+ consistency can be expressed as

$$\text{Causal+}(\mathcal{F}) \triangleq \text{Causality}(\mathcal{F}) \wedge \text{StrongConvergence}. \tag{27}$$

[8]As argued in Bailis et al. [2012], the application-level causality graph would be smaller in fanout and depth with respect to the traditional causal one, because it would only enclose relevant causal relationships, hinging on application-level knowledge and user-facing outcomes.

Real-time causal consistency has been defined in Mahajan et al. [2011] as a stricter condition than causal consistency that enforces an additional condition: causally concurrent write operations that do not overlap in real time must be applied according to their real-time order:

$$\textsc{RealTimeCausality}(\mathcal{F}) \triangleq \textsc{Causality}(\mathcal{F}) \wedge \textsc{RealTime}, \tag{28}$$

where RealTime is defined as in Equation (9).

We note that although [Lloyd et al. 2011] classifies real-time causal consistency as stronger than causal+ consistency, they are actually incomparable, as real-time causality—as defined in Mahajan et al. [2011]—does not imply strong convergence. Of course, one can devise a variant of real-time causality that respects strong convergence as well.

Attiya et al. [2015] define *observable causal* consistency as a strengthening of causal consistency for multi-value registers (MVRs) that enforces the exposure of concurrency between operations when this concurrency may be inferred by processes from their observations. Observable causal consistency has also been proved to be the strongest consistency model satisfiable for a certain class of highly available data stores implementing MVRs.

3.6. Staleness-Based Models

Intuitively, staleness-based models allow reads to return old, *stale* written values. They provide stronger guarantees than eventually consistent semantics, but weak enough to allow for more efficient implementations than linearizability. In the literature, two common metrics are employed to measure staleness: (real) time and data (object) versions.

To the best of our knowledge, the first formalization of a consistency model explicitly dealing with time-based staleness was proposed by Singla et al. [1997] as **delta** consistency. According to delta consistency, writes are guaranteed to become visible at most after $t + delta$ time units. Moreover, delta consistency is defined in conjunction with an ordering criterion (which is reminiscent of the *slow memory* consistency model, which we postpone to Section 3.9): writes to a given object by the same process are observed in the same order by all processes, but no global ordering is enforced for writes to a given object by different processes.

In an analogous way, *timed consistency* models, as defined by Torres-Rojas et al. [1999], restrict the sets of values that read operations may return by the amount of time elapsed since the preceding writes. Specifically, in a *timed serialization*, all reads occur *on time*; that is, they do not return stale values when there are more recent ones that have been available for more than Δ units of time—Δ being a parameter of the execution. In other words, similarly to delta consistency, if a write operation is performed at time t, the value written by this operation must be visible by all processes by time $t + \Delta$.

Mahajan et al. [2010] define a consistency condition named **bounded staleness** that at its core is very similar to that of timed and delta semantics: a write operation of a given process becomes visible to other processes no later than a fixed amount of time. However, this definition is also related to the use of a periodic message (i.e., a *beacon*) that allows each process to keep up with updates from other processes or *suspect* of missing updates. The differences among delta consistency, timed reads, and bounded staleness are in fact matter of subtle operational details that derive from the diverse contexts and practical purposes for which those models were developed. Hence, we can describe in formal terms the core semantics expressed by delta consistency, timed

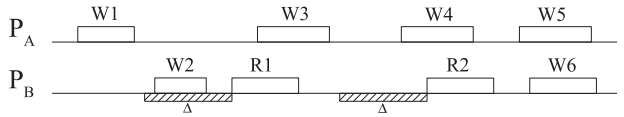

Fig. 6. An execution with processes issuing operations on a shared object. Hatched rectangles highlight the Δ parameter of staleness-based read operations.

consistency models, and bounded staleness as the following condition:

$$\textsc{TimedVisibility}(\Delta) \triangleq \forall a \in H|_{wr}, \forall b \in H,$$
$$\forall t \in \mathit{Time}: a.\mathit{rtime} = t \wedge b.\mathit{stime} = t + \Delta \Rightarrow a \xrightarrow{\text{vis}} b. \quad (29)$$

Timed causal consistency [Torres-Rojas and Meneses 2005] guarantees that each execution respects the partial ordering of causal consistency and that all reads are *on time*, with tolerance Δ:

$$\textsc{TimedCausality}(\mathcal{F}, \Delta) \triangleq \textsc{Causality}(\mathcal{F}) \wedge \textsc{TimedVisibility}(\Delta). \quad (30)$$

As depicted in Figure 1, due to the timed visibility term, timed causal is a semantic condition stronger than causal consistency.

Similarly, **timed serial** consistency [Torres-Rojas and Meneses 2005] combines the real-time global ordering guarantee with the timed serialization constraint. Hence, a timed serial consistent execution with $\Delta = 0$ would in fact be linearizable.

Golab et al. [2011] describe **Δ-atomicity**, a semantic condition that is in fact equivalent to timed serial consistency. Namely, according to Δ-atomicity, read operations may return either the value written by the last preceding write or the value of a write operation returned up to Δ time units ago. In a follow-up work [Golab et al. 2014], the same authors propose a novel metric called Γ, which entails fewer assumptions and is more robust than Δ against clock skews. The corresponding consistency semantics, **Γ-atomicity**, expresses, as Δ-atomicity, a "deviation" in time of a given execution from a linearizable one having the same operations' outcomes.

We express the core notion of Δ-atomicity, Γ-atomicity, and timed serial consistency in the following predicate:

$$\textsc{TimedLinearizability}(\mathcal{F}, \Delta) \triangleq \textsc{SingleOrder} \wedge \textsc{TimedVisibility}(\Delta) \wedge \textsc{RVal}(\mathcal{F}). \quad (31)$$

Figure 6 illustrates an execution featuring read operations of which outcomes should depend on a fixed timing parameter Δ.

If we were to assume that, despite the timing parameter, P_A and P_B observed the following serialization:

$$S_{P_{A,B}}: \quad W2 \quad W6 \quad W1 \quad W3 \quad W4 \quad W5, \quad (S.10)$$

then such execution would be sequentially consistent but it would not satisfy timed serial consistency requirements. Thus, this execution serves as a hint of the relative strengths of sequential and timed serial consistency models as represented in Figure 1.

Prefix consistency [Terry et al. 1995; Terry 2013], also dubbed **timeline** consistency [Cooper et al. 2008], grants readers the guarantee of observing an ordered sequence of writes that nonetheless may not contain the most recent ones. So it expresses a constraint in matter of ordering rather than recency of writes: the read value is the result of a specific sequence of writes upon whose order all replicas have agreed. This pre-established order is supposedly reminiscent of that one imposed by sequential consistency. Thus, we could rename prefix consistency as *prefix sequential* consistency,

whereas a version abiding real-time constraints would be called *prefix linearizable* consistency. Formally, we describe prefix sequential consistency as:

$$\textsc{PrefixSequential}(\mathcal{F}) \triangleq \textsc{SingleOrder} \wedge \textsc{MonotonicWrites} \wedge \textsc{RVal}(\mathcal{F}), \tag{32}$$

where the term named MonotonicWrites implies that the ordering of writes belonging to the same session is respected, as defined in Equation (24). Similarly, we express prefix linearizable consistency as

$$\textsc{PrefixLinearizable}(\mathcal{F}) \triangleq \textsc{SingleOrder} \wedge \textsc{RealTimeWW} \wedge \textsc{RVal}(\mathcal{F}), \tag{33}$$

where

$$\textsc{RealTimeWW} \triangleq rb|_{wr \to wr} \subseteq ar. \tag{34}$$

In a study on quorum-based replicated systems with malicious faults, Aiyer et al. [2005] formalize relaxed semantics that tolerate limited version-based staleness. Substantially, ***K*-safe**, ***K*-regular**, and ***K*-atomic** (or ***K*-linearizability**) generalize the register consistency conditions previously introduced in Lamport [1986a] and described in Section 3.1, by permitting reads non-overlapping concurrent writes to return one of the latest K values written. For instance ***K*-linearizability** can be formalized as[9]

$$\begin{aligned}&\text{K-}\textsc{Linearizable}(\mathcal{F}, K)\\ &\quad\triangleq \textsc{SingleOrder} \wedge \textsc{RealTimeWW} \wedge \text{K-}\textsc{RealTimeReads}(K) \wedge \textsc{RVal}(\mathcal{F}),\end{aligned} \tag{35}$$

where

$$\begin{aligned}&\text{K-}\textsc{RealTimeReads}(K) \triangleq \forall a \in H|_{wr}, \forall b \in H|_{rd}, \forall PW \subseteq H|_{wr},\\ &\qquad \forall pw \in PW : |PW| < K \wedge a \xrightarrow{ar} pw \wedge pw \xrightarrow{rb} b \wedge a \xrightarrow{rb} b \Rightarrow a \xrightarrow{ar} b.\end{aligned} \tag{36}$$

Finally, Bailis et al. [2012] build on these results a series of probabilistic models to predict the staleness of reads performed on eventually consistent quorum-based stores. They provide definitions of Probabilistically Bounded Staleness (PBS) ***k*-staleness** and PBS ***t*-visibility**. While the first describes a probabilistic model that restricts the staleness of values returned by read operations, the latter limits probabilistically the time before a write becomes visible. The combination of these two models is named PBS **⟨*k*, *t*⟩-staleness**. In a sense, PBS k-staleness is a probabilistic weakening of K-atomicity, that is, the one that with probability equal to 1 becomes K-linearizability. Similarly, PBS t-visibility is a probabilistic weakening of timed visibility.

3.7. Fork-Based Models

Inherent trust limitations that arise in the context of outsourced storage and computations [Cachin et al. 2009b; Vukolić 2010] have revamped the research on algorithms and protocols expressly conceived to deal with Byzantine faults [Lamport et al. 1982], that is, faults that encompass arbitrary and malicious behavior. In the Byzantine fault model, faulty processes and shared objects may tamper with data (within the limits of cryptography) or perform other arbitrary operations in order to deliberately disrupt executions.

Together with these algorithms, new consistency models were defined that reshaped the correctness conditions in accordance to what is actually attainable when coping with such strong fault assumptions. Whereas in the context of several untrusted storage repositories Byzantine fault tolerance could be applied to mask certain fault patterns

[9]Strictly speaking, K-linearizability implicitly assumes K initial writes (i.e., writes with input value $\perp$) [Aiyer et al. 2005].

[Vukolić 2010; Bessani et al. 2013] and even implement strong consistency semantics (e.g., linearizability) [Bessani et al. 2014; Dobre et al. 2014], when dealing with a single untrusted storage repository, the situation is different and the consistency needs to be relaxed [Cachin et al. 2009b]. Feasible consistency semantics in the context of interactions of correct clients with untrusted (Byzantine) storage have been captured within the family of *fork-based* consistency models. In a nutshell, systems dealing with untrusted storage aim at providing linearizability when the storage is correct, but (gracefully) degrade to weaker consistency models, specifically, fork-based consistency models, when the storage exhibits a Byzantine fault.

The forefather of this family of models is **fork** (or fork-linearizable) consistency, introduced by Mazières and Shasha [2002]. In short, a fork-linearizable system guarantees that if the storage system causes the visible histories of two processes to differ even for just a single operation, they may never again observe each other's writes after that without the server being exposed as faulty. Specifically, any divergence in the histories observed by different groups of correct processes can be easily spotted by using any available communication protocol between them (e.g., out-of-band communication, gossip protocols, etc.). Fork-linearizability respects session order (PRAM semantics) and real-time arbitration, and thus can be expressed as follows:

$$\textsc{ForkLinearizability}(\mathcal{F}) \triangleq \text{PRAM} \wedge \textsc{RealTime} \wedge \textsc{NoJoin} \wedge \text{RVal}(\mathcal{F}), \tag{37}$$

where the NoJoin predicate stipulates that clients whose sequences of visible operations (also called *views*) have been forked by an adversary cannot be joined again:

$$\begin{aligned}\textsc{NoJoin} \triangleq \forall a_i, b_i, a_j, b_j \in H : a_i \not\approx_{ss} a_j \wedge (a_i, a_j) \in ar \setminus vis \wedge a_i \preceq_{so} b_i \wedge a_j \preceq_{so} b_j \\ \Rightarrow (b_i, b_j), (b_j, b_i) \notin vis.\end{aligned} \tag{38}$$

A subsequent model named **fork*** consistency was defined in Li and Mazières [2007] in order to allow the design of protocols that would offer better performance and liveness guarantees. Fork* consistency relaxes the conditions of fork consistency by allowing forked groups of processes to observe at most one common operation issued by a certain correct process:

$$\textsc{Fork*}(\mathcal{F}) \triangleq \textsc{ReadYourWrites} \wedge \textsc{RealTime} \wedge \textsc{AtMostOneJoin} \wedge \text{RVal}(\mathcal{F}), \tag{39}$$

where

$$\begin{aligned}\textsc{AtMostOneJoin} \triangleq \forall a_i, a_j \in H : a_i \not\approx_{ss} a_j \wedge (a_i, a_j) \in ar \setminus vis \Rightarrow \\ \wedge |\{b_i \in H : a_i \preceq_{so} b_i \wedge (\exists b_j \in H : a_j \preceq_{so} b_j \wedge b_i \xrightarrow{vis} b_j\}| \leq 1 \\ \wedge |\{b_j \in H : a_j \preceq_{so} b_j \wedge (\exists b_i \in H : a_i \preceq_{so} b_i \wedge b_j \xrightarrow{vis} b_i\}| \leq 1.\end{aligned} \tag{40}$$

Notice that, unlike fork-linearizability, fork* does not respect monotonicity of reads (and hence PRAM) [Cachin et al. 2011].

Fork-sequential consistency [Oprea and Reiter 2006; Cachin et al. 2009a] requires that whenever an operation becomes visible to multiple processes, all these processes share the same history of operations occurring before that operation. Therefore, whenever a process reads a certain value written by another process, the reader is guaranteed to share with the writer process the set of visible operations that precede that write operation. Essentially, similarly to sequential consistency, a global order of operations is ensured up to a common visible operation. Formally:

$$\textsc{ForkSequential}(\mathcal{F}) \triangleq \text{PRAM} \wedge \textsc{NoJoin} \wedge \text{RVal}(\mathcal{F}). \tag{41}$$

Mahajan et al. define **fork-join causal** consistency (FJC) as a weaker variant of causal consistency that can preserve safeness and availability in spite of Byzantine faults [Mahajan et al. 2010]. In a fork-join causal consistent storage system, if a write operation *op* issued by a correct process depends on a write operation op' issued by any process, then, at every correct process, op' becomes visible before *op*. In other words, FJC enforces causal consistency among correct processes. Besides, partitioned groups of processes are allowed to reconcile their histories through merging policies, since inconsistent writes by a Byzantine process are treated as concurrent writes by multiple *virtual processes*. **Bounded fork-join causal** [Mahajan et al. 2011] refines this clause by limiting the number of forks accepted from a faulty node and thus bounding the number of virtual nodes needed to represent each faulty node.

Finally, **weak fork-linearizability** [Cachin et al. 2011] relaxes fork-linearizability conditions in two ways: (1) after being partitioned in different groups, two processes may share the visibility of one more operation (i.e., *at-most-one-join*, as in fork* consistency) and (2) the real-time order of the last visible operation by each process may not be preserved (i.e., *weak real-time order*). These two conditions enable the design of protocols that allow for improved liveness guarantees (i.e., *wait freedom*). Weak fork-linearizability can be expressed as

$$\textsc{WeakForkLin}(\mathcal{F}) \triangleq \text{PRAM} \wedge \text{K-}\textsc{RealTime}(2) \wedge \textsc{AtMostOneJoin} \wedge \textsc{RVal}(\mathcal{F}), \tag{42}$$

where K-RealTime(2) predicate is equivalent K-RealTimeReads(2) defined in Equation (36), when generalized to all operations (i.e., when the predicate holds $\forall op \in H$). We note that weak fork-linearizability and fork* consistency are incomparable [Cachin et al. 2011].

3.8. Composite and Tunable Semantics

To bridge the gap between strongly consistent and efficient implementations, several works have proposed consistency models that entail the use of different semantics in an adaptive fashion according to the contingent tradeoffs of performance and correctness.[10]

The idea of distinguishing operations' consistency requirements by their semantics dates back to the shared-memory systems era. In that context, consistency models that employed different ordering constraints depending on operations' types (e.g., *acquire* and *release*, rather than read/write data accesses) were called *hybrid*, whereas those that did not operate distinctions were referred to as *uniform* [Mosberger 1993; Dubois et al. 1986; Gharachorloo et al. 1990].

A first formal definition that presents a similar diversification was proposed by Attiya and Friedman [1992] for shared-memory multiprocessors. **Hybrid** consistency is defined as a model requiring a concerted adoption of weak and strong consistency semantics. In a hybrid consistent system, *strong* operations are guaranteed to be seen in some sequential order by all processes (as in sequential consistency), while *weak* operations are designed to be fast, and they eventually become visible by all processes (much like in eventual consistency). Weak operations are only guaranteed to be ordered according to their interleaving with strong operations: if two operations belong to the same session and one of them is strong, then their relative order of invocation is respected and visible by all processes. In a similar manner, Ladin et al. [1992] tackle the tradeoff between performance and consistency by assigning to each operation an ordering type. *Causal* operations respect causality ordering among them, *forced* operations are delivered in the same order at all relicas, and *immediate* operations are

[10]We do not formulate formal definitions for tunable semantics considering that they can be expressed by combining the logical predicates reported in the rest of the article.

performed as they return and they are delivered by each replica in the same order with respect to all other operations.

Eventual serializability[11] is described in Fekete et al. [1996] as a condition that requires a partial ordering of operations that eventually settle to a total order. According to this model, operations might be *strict* or *non-strict*. Strict operations are required to be *stable* as soon as they obtain a response, while non-strict ones may be reordered afterward. An operation is said to be *stable* if the prefix of operations preceding it reached a final total order. Fekete et al. [1996] envision an implementation in which processes issue operations attaching to them both the list of identifiers of operations that must be ordered before the requested operation and a flag that indicates the type of operation (i.e., strict or non-strict). The final global and total order achieved by operations can be regarded as a sequential consistency ordering as no real-time notion is involved.

Similarly, Serafini et al. [2010] distinguish *strong* and *weak* operations. While strong operations are immediately linearized, weak ones are linearized only eventually. Weak operations are thus said to respect **eventual linearizability**. Weak operations are in fact designed to terminate despite failures, and can therefore violate linearizability for a finite period of time. Essentially, eventual linearizability mandates that operations must be ordered according to their real-time ordering, yet this applies only to operations invoked after a certain time t. Therefore, earlier operations may have observed inconsistent histories and can be temporarily ordered in an arbitrary manner. Ultimately, the operations in a system that implements eventual linearizability gravitate toward a total order that satisfies real-time constraints.

Krishnamurthy et al. [2002] propose a QoS model that allows client applications of a distributed storage system to express their consistency requirements. According to their requirements, clients are then directed by a middleware toward a specific group of replicas implementing synchronous or lazy replication schemes, thus applying strong or weak consistency semantics. This framework is said to provide **tunable** consistency.

In the same vein, Li et al. [2012] propose **RedBlue** consistency. With RedBlue consistency, operations are flagged as *blue* or *red* depending on several conditions such as their commutativity and the respect of invariants. According to such classification, operations are then executed locally and replicated in an eventually consistent manner, or serialized with respect to each other through synchronous coordination. In a follow-up work, Li et al. [2014] implement and evaluate a system that would relieve the programmer from having to choose the right consistency level for each operation by exploiting a combination of automatic static and dynamic code analyses.

Yu and Vahdat [2002] propose a continuous consistency spectrum based on three metrics: *staleness*, *order error,* and *numerical error*. Those metrics are embedded in a **conit** (portmanteau of "consistency unit"), which is a three-dimensional vector that quantifies the divergence from an ideal linearizable execution. Numerical error accounts for the number of write operations that are already globally applied but not yet propagated to a given replica of a certain object. Order error quantifies the number of writes at any replica that are subject to reordering, while staleness bounds the real-time delay of writes propagation among replicas. Those metrics are an attempt to capture the semantics of some fundamental dimensions of consistency, notably those related to the general requirements of agreement on state and update ordering. Note that, according to this model, and unlike timed consistency (see Section 3.6), time-based staleness is defined from the replicas' viewpoint rather than with respect to the timing of individual operations.

[11]We remark that despite the affinity of its name with those of popular transactional consistency models, eventual serializability has been conceived for non-transactional storage systems.

Similarly, Santos et al. [2007] aim at quantifying the divergence of data object replicas by using a three-dimensional consistency vector. Originally designed for distributed multiplayer games on ad hoc networks, **vector-field** consistency mandates for each object a vector $\kappa = [\theta, \sigma, \nu]$ that bounds its staleness in a particular *view* of the virtual world. In particular, the vector establishes the maximum divergence of replicas in time (θ), number of updates (σ), and object value (ν). Unlike conit, this model brings about a notion of locality awareness as it describes consistency as a vector field deployed throughout the gaming virtual environment.

Later works put forward tunable consistency as a suitable model for cloud storage, since it would enable more flexible quality-of-service (QoS) policies and service-level agreements (SLAs). Kraska et al. [2009] envision consistency **rationing**, which would entail adapting the consistency level at runtime by taking into account economic concerns. Similarly, Chihoub et al. [2012] explore the possibility of a self-adaptive protocol that dynamically adjusts consistency to meet the application needs. In a subsequent work, Chihoub et al. [2013] add the monetary cost to the equation and study its trade-offs with consistency in cloud settings. Terry et al. [2013] advocate the use of declarative consistency-based SLAs that would allow users of cloud key-value stores to attain a better awareness of the inherent performance-correctness tensions. This approach has been subsequently implemented as a declarative programming model for tunable consistency by Sivaramakrishnan et al. [2015].

In another attempt at providing stronger consistency semantics for geo-replicated storage, Balegas et al. [2015] introduce *explicit* consistency. Besides providing eventual consistency, a replicated store implementing explicit consistency ensures that application-specific correctness rules (i.e., *invariants*) be respected during executions. In a follow-up work, Gotsman et al. [2016] propose a proof rule to help programmers in the task of assigning fine-grained restrictions on operations in order to respect data integrity invariants.

Finally, in the context of combining different consistency models, it is also worth mentioning systems that turn eventual consistency of data (provided by modern commodity cloud storage services) into linearizability, by relying on comparably small volumes of metadata stored separately from data in linearizable storage. In independent efforts, this technique was recently proposed under the names of *consistency anchor* [Bessani et al. 2014] and *consistency hardening* [Dobre et al. 2014].

3.9. Per-Object Semantics

Per-object (or per-key) semantics have been defined to express consistency constraints on a per-object basis. Intuitively, per-object ordering semantics allow for more efficient implementations than global ordering semantics, that is, across invocations on all objects, taking advantage of techniques such as sharding and state partitioning.

Slow memory, defined by Hutto and Ahamad [1990], is a weaker variant of PRAM consistency. A shared-memory system implementing this condition requires that all processes see the writes of a given process to a given object in the same order. In other words, slow memory delineates a per-object weakening of PRAM consistency:

$$\textsc{PerObjectPRAM} \triangleq (so \cap ob) \subseteq vis. \tag{43}$$

An important concept in this family of semantics is that of **coherence** [Gharachorloo et al. 1990] (or cache consistency [Goodman 1989]), which was first introduced as a correctness condition of memory hierarchies in shared-memory multiprocessor systems [Dubois et al. 1986]. Coherence ensures that what has been written to a specific memory location becomes visible in some sequential order by all processors, possibly through their local caches. In other words, coherence requires operations to be globally ordered on a per-object basis. A very similar notion has been coined in recent works [Cooper

et al. 2008; Lloyd et al. 2011] as **per-record timeline** consistency. This condition, described in relation to replicated storage, ensures that for each individual key (or object), all processes observe the same ordering of operations. Formally, we capture such condition with the following predicate:

$$\textsc{PerObjectSingleOrder} \triangleq \exists H' \subseteq \{op \in H : op.oval = \nabla\} : ar \cap ob = vis \cap ob \setminus (H' \times H). \quad (44)$$

Moreover, a system in which executions respect ordering of operations by a certain process on each object and a global ordering of all operations invoked on each object would implement a semantic condition that we could name as *per-object sequential* consistency:

$$\textsc{PerObjectSequential}(\mathcal{F}) \triangleq \textsc{PerObjectSingleOrder} \wedge \textsc{PerObjectPRAM} \wedge \textsc{RVal}(\mathcal{F}). \quad (45)$$

Processor consistency, defined by Goodman [1989] and formalized by Ahamad et al. [1993], is expressed by two conditions: (1) writes issued by a process must be observed in the order in which they were issued, and (2) if there are two write operations to the same object, all processes observe these operations in the same order. Evidently, the two conditions just mentioned are in fact PRAM and per-record timeline consistency, and thus:

$$\textsc{ProcessorConsistency}(\mathcal{F}) \triangleq \textsc{PerObjectSingleOrder} \wedge \text{PRAM} \wedge \textsc{RVal}(\mathcal{F}). \quad (46)$$

In addition, few works in the literature (e.g., Moraru et al. [2013]) mention *per-object linearizability*, which is in fact equivalent to linearizability on a per-object basis, due to its *locality* property [Herlihy and Wing 1990].

We further note that one could compose other arbitrary consistency models by refining some of the predicates mentioned in this work to match only operations performed on individual objects. As a case in point, Burckhardt et al. [2014] describe **per-object causal** consistency as a restriction of causal consistency on a per-object basis, which leverages the *per-object happens-before* order, defined as $hbo \triangleq ((so \cap ob) \cup vis)^+$.

3.10. Synchronized Models

For completeness, in this section we overview semantic conditions defined in the '80s and early '90s in order to model the correctness of multiprocessor shared-memory systems. In order to exploit the computational parallelism of these systems and, at the same time, to cope with the different performance of the various components (e.g., memories, interconnections, processors, etc.), buffering and caching layers were adopted. Consequently, the fundamental challenge of this kind of architecture is making sure that all memories reflect a common, consistent view of shared data. Thus, system designers employed *synchronization variables*, that is, special shared objects that only expose two operations, named *acquire* and *release*. The synchronization variables are used as a generic abstraction for implementing logical fences meant to control concurrent accesses to shared data objects. In other words, synchronization variables protect the access to shared data through the implementation of mutual exclusion by means of low-level primitives (e.g., locks) or high-level language constructs (e.g., critical sections). While the burden of using such tools is left to the programmer, the system is supposed to distinguish the accesses to shared data from those to the synchronization variables, possibly by implementing and exposing specific low-level instructions.

Sequential consistency [Lamport 1979] (which we defined in Section 3.3) was initially adopted as the ideal correctness condition for multiprocessors' shared-memory

systems. **Weak ordering**[12] as described by Dubois et al. [1986] represents a convenient weakening of sequential consistency that brings about performance improvements. In a system that implements weak ordering, (1) all accesses to synchronization variables must be *strongly ordered*, (2) no access to a synchronization variable is allowed before all previous reads have been completed, and (3) processes cannot perform reads before issuing an access to a synchronization variable. In particular, Dubois et al. [1986] define operations as strongly ordered if they comply with two specific criteria that constrain the ordering of operations according to their session ordering and relatively to some special instructions supported by pipelined cache-based systems. Weak ordering has been subsequently redefined in terms of coordination requirements between software and hardware. Namely, Adve and Hill [1990] define a *synchronization model* as a set of constraints on memory accesses that specify how and when synchronization needs to be enforced. Given this definition, "a hardware is weakly ordered with respect to a given synchronization model if and only if it appears sequentially consistent to all software that obey the synchronization model."

Release consistency, presented by Gharachorloo et al. [1990], is a weaker extension of weak ordering that exploits further detailed information about synchronization operations (i.e., acquire and release) and non-synchronization accesses. Operations have to be labeled before execution by the programmer (or the compiler) as strong or weak. Hence, this widens the classification operated by weak ordering, which included just synchronization and non-synchronization labels. Similarly to hybrid consistency (see Section 3.8), strong operations are ordered according to processor or sequential consistency, whereas the ordering of weak operations is just restricted by the relative ordering with respect to the strong operations invoked by the same process.

Subsequently, several algorithms that slightly alter the original implementation of release consistency have been designed. For instance, **lazy release** consistency [Keleher et al. 1992] is a relaxed implementation of release consistency in which actions that enforce consistency are postponed from the release to the next acquire operation. The rationale of lazy release consistency is reducing the number of messages and the amount of data exchanged in a distributed shared-memory system implemented in software. On the same line, the protocol called *automatic update release consistency* [Iftode et al. 1996a] aims at improving performance substantially over software-only implementation of lazy release consistency by using an automatic update mechanism provided by a virtual memory mapped network interface.

Bershad and Zekauskas [1991] define **entry** consistency by strengthening the relation between synchronization objects and the data they guard. According to entry consistency, every object has to be guarded by a synchronization variable. Thus, in a sense, this model is a location-relative weakening of a consistency semantic, similar to the models surveyed in Section 3.9. Moreover, entry consistency operates a further distinction of the synchronization operations in exclusive and non-exclusive. Thanks to these features, reads can occur with a greater degree of concurrency, thus enabling better performance.

Scope consistency [Iftode et al. 1996b] claims to offer most of the potential performance advantages of entry consistency, without requiring explicit binding of data to synchronization variables. The key intuition of scope consistency is the use of an abstraction called *scope* to implicitly capture the relationship between data and synchronization operations. Consistency scopes can be derived automatically from the use of synchronization variables in the program, thus easing the work of programmers.

[12]Some works in the literature refer to weak ordering as "weak consistency." We chose to avoid this equivocation by adopting its original nomenclature.

With the definition of **location** consistency, Gao and Sarkar [2000] forwent the basic assumption of *memory coherence* [Gharachorloo et al. 1990], that is, the property that ensures that all writes to the same object are observed in the same order by all processes (see Section 3.9). Thus, they explored the possibility of executing multithreaded programs in a correct manner by just exploiting a partial order on writes to shared data. Similarly to entry consistency, in location consistency each object is associated to a synchronization variable. However, thanks to the relaxed underlying ordering constraint, Gao and Sarkar [2000] prove that location consistency can be more efficient and equivalently strong when it is applied to settings with low data contention between processes.

4. RELATED WORK

Several works in the literature have provided overviews on consistency models. In this section, we classify these works according to their different perspectives.

Shared-Memory Systems. Gharachorloo et al. [1990] proposed a classification of shared-memory access policies, specifically regarding their concurrency control semantics (e.g., synchronization operations vs. read/write accesses). Mosberger [1993] adopted this classification to conduct a study on the memory consistency models popular at that time and their implementation tradeoffs. Adve and Gharachorloo [1996] summarized in a practical tutorial the informal definitions and related issues of consistency models most commonly adopted in shared-memory multiprocessor systems.

Several subsequent works developed uniform frameworks and notations to represent consistency semantics defined in the literature [Adve and Hill 1993; Raynal and Schiper 1997; Bataller and Bernabéu-Aubán 1997]. Most notably, Steinke and Nutt [2004] provide a unified theory of consistency models for shared-memory systems based on the composition of a few fundamental declarative properties. In turn, this declarative and compositional approach outlines a partial ordering over consistency semantics. Similarly, a treatment of composability of consistency conditions had been carried out in Friedman et al. [2003].

While all these works proved to be valuable and formally sound, they represent only a limited portion of the consistency semantics relevant to modern non-transactional storage systems.

Distributed Storage Systems. In more recent years, researchers have been proposing categorizations of the most influential consistency models for modern storage systems. Namely, Tanenbaum and van Steen [2007] proposed the client-centric versus data-centric classification, while Bermbach and Kuhlenkamp [2013] expanded such classification and provided descriptions for the most popular models. While practical and instrumental in attaining a good understanding of the consistency spectrum, these works propose informal treatments based on a simple dichotomous categorization that falls short of capturing some important consistency semantics. With this survey, we aim at improving these works, as we adopt a formal model based on first-order logic predicates and graph theory. We derived this model from the one proposed in Burckhardt [2014], which we modified and expanded in order to enable the definition of a wider and richer range of consistency semantics. Moreover, whereas Burckhardt [2014] focuses mostly on session and eventual semantics, we cover a broader ground including more than 50 different consistency semantics.

Measuring Consistency. A concurrent research trend has been straining to design uniform and rigorous frameworks to measure consistency in both shared-memory systems and, more recently, distributed storage systems. Namely, while some works have proposed metrics to assess consistency [Yu and Vahdat 2002; Golab et al. 2014], others have devised methods to verify, given an execution, whether it satisfies a certain

consistency model [Misra 1986; Gibbons and Korach 1997; Anderson et al. 2010]. Finally, due to the loose definitions and opaque implementations of eventual consistency, recent research has tried to quantify its inherent anomalies as perceived from a client-side perspective [Wada et al. 2011; Patil et al. 2011; Bermbach and Tai 2011; Rahman et al. 2012; Lu et al. 2015]. In this regard, our work provides a more comprehensive and structured overview of the metrics that can be adopted to evaluate consistency.

Transactional Systems. Readers interested in pursuing a formal treatment of the most important consistency models for transactional storage systems may refer to Adya [1999]. Similarly, other works by Harris et al. [2010] and by Dziuma et al. [2014] complement this survey with overviews on models specifically designed for transactional memory systems. Finally, some recent research [Burckhardt et al. 2012; Cerone et al. 2015] adopted variants of the same framework used in this article to propose axiomatic specifications of transactional consistency models.

5. CONCLUSION

In this work, we presented an overview of the most relevant consistency models for non-transactional storage systems. Thanks to our methodical approach, we were able to highlight subtle yet meaningful differences among consistency models, thus helping scholars and practitioners attain a better understanding of the tradeoffs involved.

To describe consistency semantics, we adopted a mathematical framework based on graph theory and first-order logic. As a first contribution of this work, we developed such a formal framework as an extension of the one presented in Burckhardt [2014]. The framework is comprehensive and useful in capturing different factors involved in the executions of a distributed storage system.

We used this framework to formulate formal definitions for the most popular of the more than 50 consistency semantics we analyzed. For the rest of them, we presented informal descriptions that provide insights about their feature and relative strengths. Moreover, thanks to the axiomatic approach we adopted, we laid out a clustering of semantics according to criteria that account for their natures and common traits. In turn, both the clustering and the formal definitions helped us in building a partial ordering of consistency models (see Figure 1). We believe this partial ordering of semantics will prove convenient both in designing more precise and coherent models and in evaluating and comparing the correctness of systems already in place. Finally, as a further contribution, we provide in Appendix B an ordered list of all the models analyzed in this work, along with references to their definitions and main implementations in research literature.

APPENDIX

A. SUMMARY OF CONSISTENCY PREDICATES

Table I. Summary of Consistency Predicates Listed in the Article

Linearizability($\mathcal{F}$)	SingleOrder $\wedge$ RealTime $\wedge$ RVal($\mathcal{F}$)
SingleOrder	$\exists H' \subseteq \{op \in H : op.oval = \nabla\} : vis = ar \setminus (H' \times H)$
RealTime	$rb \subseteq ar$
Regular($\mathcal{F}$)	SingleOrder $\wedge$ RealTimeWrites $\wedge$ RVal($\mathcal{F}$)
Safe($\mathcal{F}$)	SingleOrder $\wedge$ RealTimeWrites $\wedge$ SeqRVal($\mathcal{F}$)
RealTimeWrites	$rb\vert_{wr \to op} \subseteq ar$
SeqRVal($\mathcal{F}$)	$\forall op \in H : Concur(op) = \emptyset \Rightarrow op.oval \in \mathcal{F}(op, cxt(A, op))$
EventualConsistency($\mathcal{F}$)	EventualVisibility $\wedge$ NoCircularCausality $\wedge$ RVal($\mathcal{F}$)

(Continued)

Table I. Continued

EventualVisibility	$\forall a \in H, \forall [f] \in H/\approx_{ss}: |\{b \in [f] : (a \xrightarrow{rb} b) \wedge (a \not\xrightarrow{vis} b)\}| < \infty$
NoCircularCausality	$acyclic(hb)$
StrongConvergence	$\forall a, b \in H|_{rd} : vis^{-1}(a)|_{wr} = vis^{-1}(b)|_{wr} \Rightarrow a.oval = b.oval$
StrongEventualCons.($\mathcal{F}$)	EventualConsistency($\mathcal{F}$) $\wedge$ StrongConvergence
QuiescentConsistency($\mathcal{F}$)	$|H|_{wr}| < \infty \Rightarrow \exists C \in \mathcal{C} : \forall [f] \in H/\approx_{ss}: |\{op \in [f] : op.oval \notin \mathcal{F}(op, C)\}| < \infty$
PRAM	$so \subseteq vis$
SequentialConsistency($\mathcal{F}$)	SingleOrder $\wedge$ PRAMConsistency $\wedge$ RVal($\mathcal{F}$)
MonotonicReads	$\forall a \in H, \forall b, c \in H|_{rd} : a \xrightarrow{vis} b \wedge b \xrightarrow{so} c \Rightarrow a \xrightarrow{vis} c \triangleq (vis; so|_{rd \to rd}) \subseteq vis$
ReadYourWrites	$\forall a \in H|_{wr}, \forall b \in H|_{rd} : a \xrightarrow{so} b \Rightarrow a \xrightarrow{vis} b \triangleq so|_{wr \to rd} \subseteq vis$
MonotonicWrites	$\forall a, b \in H|_{wr} : a \xrightarrow{so} b \Rightarrow a \xrightarrow{ar} b \triangleq so|_{wr \to wr} \subseteq ar$
WritesFollowReads	$\forall a, c \in H|_{wr}, \forall b \in H|_{rd} : a \xrightarrow{vis} b \wedge b \xrightarrow{so} c \Rightarrow a \xrightarrow{ar} c \triangleq (vis; so|_{rd \to wr}) \subseteq ar$
CausalVisibility	$hb \subseteq vis$
CausalArbitration	$hb \subseteq ar$
Causality($\mathcal{F}$)	CausalVisibility $\wedge$ CausalArbitration $\wedge$ RVal($\mathcal{F}$)
Causal+($\mathcal{F}$)	Causality($\mathcal{F}$) $\wedge$ StrongConvergence
RealTimeCausality($\mathcal{F}$)	Causality($\mathcal{F}$) $\wedge$ RealTime
TimedVisibility(Δ)	$\forall a \in H|_{wr}, \forall b \in H, \forall t \in Time : a.rtime = t \wedge b.stime = t + \Delta \Rightarrow a \xrightarrow{vis} b$
TimedCausality($\mathcal{F}$, Δ)	Causality($\mathcal{F}$) $\wedge$ TimedVisibility(Δ)
TimedLinearizability($\mathcal{F}$, Δ)	SingleOrder $\wedge$ TimedVisibility(Δ) $\wedge$ RVal($\mathcal{F}$)
PrefixSequential($\mathcal{F}$)	SingleOrder $\wedge$ MonotonicWrites $\wedge$ RVal($\mathcal{F}$)
PrefixLinearizable($\mathcal{F}$)	SingleOrder $\wedge$ RealTimeWW $\wedge$ RVal($\mathcal{F}$)
RealTimeWW	$rb|_{wr \to wr} \subseteq ar$
K-Linearizable($\mathcal{F}$, K)	SingleOrder $\wedge$ RealTimeWW $\wedge$ K-RealTimeReads(K) $\wedge$ RVal($\mathcal{F}$)
K-RealTimeReads(K)	$\forall a \in H|_{wr}, \forall b \in H|_{rd}, \forall PW \subseteq H|_{wr}, \forall pw \in PW : |PW| < K \wedge a \xrightarrow{ar} pw \wedge pw \xrightarrow{rb} b \wedge a \xrightarrow{rb} b \Rightarrow a \xrightarrow{ar} b$
ForkLinearizability($\mathcal{F}$)	PRAM $\wedge$ RealTime $\wedge$ NoJoin $\wedge$ RVal($\mathcal{F}$)
NoJoin	$\forall a_i, b_i, a_j, b_j \in H : a_i \not\approx_{ss} a_j \wedge (a_i, a_j) \in ar \setminus vis \wedge a_i \preceq_{so} b_i \wedge a_j \preceq_{so} b_j \Rightarrow (b_i, b_j), (b_j, b_i) \notin vis$
Fork*($\mathcal{F}$)	ReadYourWrites $\wedge$ RealTime $\wedge$ AtMostOneJoin $\wedge$ RVal($\mathcal{F}$)
AtMostOneJoin	$\forall a_i, a_j \in H : a_i \not\approx_{ss} a_j \wedge (a_i, a_j) \in ar \setminus vis \Rightarrow |\{b_i \in H : a_i \preceq_{so} b_i \wedge (\exists b_j \in H : a_j \preceq_{so} b_j \wedge b_i \xrightarrow{vis} b_j\}| \leq 1 \wedge |\{b_j \in H : a_j \preceq_{so} b_j \wedge (\exists b_i \in H : a_i \preceq_{so} b_i \wedge b_j \xrightarrow{vis} b_i\}| \leq 1$
ForkSequential($\mathcal{F}$)	PRAM $\wedge$ NoJoin $\wedge$ RVal($\mathcal{F}$)
WeakForkLin($\mathcal{F}$)	PRAM $\wedge$ K-RealTime(2) $\wedge$ AtMostOneJoin $\wedge$ RVal($\mathcal{F}$)
PerObjectPRAM	$(so \cap ob) \subseteq vis$
PerObjectSingleOrder	$\exists H' \subseteq \{op \in H : op.oval = \nabla\} : ar \cap ob = vis \cap ob \setminus (H' \times H)$
PerObjectSequential($\mathcal{F}$)	PerObjectSingleOrder $\wedge$ PerObjectPRAM $\wedge$ RVal($\mathcal{F}$)
ProcessorConsistency($\mathcal{F}$)	PerObjectSingleOrder $\wedge$ PRAM $\wedge$ RVal($\mathcal{F}$)
PerObjectHappensBefore	$hbo \triangleq ((so \cap ob) \cup vis)^+$

B. PRIMARY REFERENCES

Table II. Definitions of Consistency Semantics and Some of Their Implementations in Literature

Models	Definitions	Implementations[13]
Atomicity	Lamport [1986b]	[Attiya et al. 1995]
Bounded fork-join causal	[Mahajan et al. 2011]	-
Bounded staleness	[Mahajan et al. 2010]	-

(Continued)

[13]In case of very popular consistency semantics (e.g., causal consistency, atomicity/linearizability), we only cite a subset of known implementations.

Table II. Continued

Causal	[Lamport 1978; Hutto and Ahamad 1990; Ahamad et al. 1995; Mahajan et al. 2011]	[Ladin et al. 1992; Birman et al. 1991; Lakshmanan et al. 2001; Lloyd et al. 2013; Du et al. 2014; Zawirski et al. 2015; Lesani et al. 2016]
Causal+	[Lloyd et al. 2011]	[Petersen et al. 1997; Belaramani et al. 2006; Almeida et al. 2013]
Coherence	[Dubois et al. 1986]	-
Conit	[Yu and Vahdat 2002]	-
Γ-atomicity	[Golab et al. 2014]	-
Δ-atomicity	[Golab et al. 2011]	-
Delta	[Singla et al. 1997]	-
Entry	[Bershad and Zekauskas 1991]	-
Eventual	[Terry et al. 1994; Vogels 2008]	[Reiher et al. 1994; DeCandia et al. 2007; Singh et al. 2009; Bortnikov et al. 2010; Bronson et al. 2013]
Eventual linearizability	[Serafini et al. 2010]	-
Eventual serializability	[Fekete et al. 1996]	-
Fork*	[Li and Mazières 2007]	[Feldman et al. 2010]
Fork	[Mazières and Shasha 2002; Cachin et al. 2007]	[Li et al. 2004; Brandenburger et al. 2015]
Fork-join causal	[Mahajan et al. 2010]	-
Fork-sequential	[Oprea and Reiter 2006]	-
Hybrid	[Attiya and Friedman 1992]	-
K-atomic	[Aiyer et al. 2005]	-
K-regular	[Aiyer et al. 2005]	-
K-safe	[Aiyer et al. 2005]	-
k-staleness	[Bailis et al. 2012]	-
Lazy release	[Keleher et al. 1992]	-
Linearizability	[Herlihy and Wing 1990]	[Burrows 2006; Baker et al. 2011; Glendenning et al. 2011; Calder et al. 2011; Corbett et al. 2013; Han et al. 2015; Lee et al. 2015]
Location	[Gao and Sarkar 2000]	-
Monotonic reads	[Terry et al. 1994]	[Terry et al. 1995]
Monotonic writes	[Terry et al. 1994]	[Terry et al. 1995]
Observable causal	[Attiya et al. 2015]	-
PBS $\langle k, t \rangle$-staleness	[Bailis et al. 2012]	-
Per-object causal	[Burckhardt et al. 2014]	-
Per-record timeline	[Cooper et al. 2008; Lloyd et al. 2011]	[Andersen et al. 2009]
PRAM	[Lipton and Sandberg 1988]	-
Prefix	[Terry et al. 1995; Terry 2013]	-
Processor	[Goodman 1989]	-
Quiescent	[Herlihy and Shavit 2008]	-
Rationing	[Kraska et al. 2009]	-
Read-your-writes	[Terry et al. 1994]	[Terry et al. 1995]
Real-time causal	[Mahajan et al. 2011]	-
RedBlue	[Li et al. 2012]	-
Regular	[Lamport 1986b]	[Malkhi and Reiter 1998a; Guerraoui and Vukolic 2006]
Release	[Gharachorloo et al. 1990]	-
Safe	[Lamport 1986b]	[Malkhi and Reiter 1998b; Guerraoui and Vukolic 2006]

(Continued)

Table II. Continued

Scope	[Iftode et al. 1996b]	-
Sequential	[Lamport 1979]	[Rao et al. 2011]
Slow	[Hutto and Ahamad 1990]	-
Strong eventual	[Shapiro et al. 2011a]	[Shapiro et al. 2011b; Conway et al. 2012; Roh et al. 2011]
Timed causal	[Torres-Rojas and Meneses 2005]	-
Timed serial	[Torres-Rojas et al. 1999]	-
Timeline	[Cooper et al. 2008]	[Rao et al. 2011]
Tunable	[Krishnamurthy et al. 2002]	[Lakshman and Malik 2010; Wu et al. 2013; Perkins et al. 2015; Sivaramakrishnan et al. 2015]
t-visibility	[Bailis et al. 2012]	-
Vector-field	[Santos et al. 2007]	-
Weak	[Vogels 2008; Bermbach and Kuhlenkamp 2013]	-
Weak fork-linearizability	[Cachin et al. 2011]	[Shraer et al. 2010]
Weak ordering	[Dubois et al. 1986]	-
Writes-follow-reads	[Terry et al. 1994]	[Terry et al. 1995]

ACKNOWLEDGMENTS

We would like to thank Alysson Bessani, Christian Cachin, Marc Shapiro, and the anonymous reviewers for their helpful comments on this work.

REFERENCES

Sarita Adve and Mark D. Hill. 1990. Weak ordering—A new definition. In *International Symposium on Computer Architecture*. 2–14.

Sarita V. Adve and Kourosh Gharachorloo. 1996. Shared memory consistency models: A tutorial. *IEEE Computer* 29, 12 (1996), 66–76.

Sarita V. Adve and Mark D. Hill. 1993. A unified formalization of four shared-memory models. *IEEE Transactions on Parallel and Distributed Systems* 4, 6 (1993), 613–624. DOI:http://dx.doi.org/10.1109/71.242161

Atul Adya. 1999. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. Ph.D. Dissertation. MIT, Cambridge, MA. Also as Technical Report MIT/LCS/TR-786.

Mustaque Ahamad, Rida A. Bazzi, Ranjit John, Prince Kohli, and Gil Neiger. 1993. The power of processor consistency. In *ACM Symposium on Parallel Algorithms and Architectures (SPAA'93), 1993*. 251–260. DOI:http://dx.doi.org/10.1145/165231.165264

Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. 1995. Causal memory: Definitions, implementation, and programming. *Distributed Computing* 9, 1 (1995), 37–49.

Amitanand S. Aiyer, Lorenzo Alvisi, and Rida A. Bazzi. 2005. On the availability of non-strict quorum systems. In *Distributed Computing (DISC'05)*. 48–62. DOI:http://dx.doi.org/10.1007/11561927_6

Sérgio Almeida, João Leitão, and Luís Rodrigues. 2013. ChainReaction: A causal+ consistent datastore based on chain replication. In *European Conference on Computer Systems (EuroSys'13), 2013*. 85–98.

Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and David Maier. 2014. Blazes: Coordination analysis for distributed programs. In *IEEE Conference on Data Engineering (ICDE'14)*. 52–63. DOI:http://dx.doi.org/10.1109/ICDE.2014.6816639

Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and William R. Marczak. 2011. Consistency analysis in bloom: A CALM and collected approach. In *Conference on Innovative Data Systems Research (CIDR'11)*. 249–260. http://www.cidrdb.org/cidr2011/Papers/CIDR11_Paper35.pdf.

David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. 2009. FAWN: A fast array of wimpy nodes. In *ACM Symposium on Operating Systems Principles (SOSP'09)*. 1–14. DOI:http://dx.doi.org/10.1145/1629575.1629577

Eric Anderson, Xiaozhou Li, Mehul A. Shah, Joseph Tucek, and Jay J. Wylie. 2010. What consistency does your key-value store actually provide? In *Hot Topics in System Dependability (HotDep'10)*. USENIX Association, Berkeley, CA, 1–16. http://dl.acm.org/citation.cfm?id=1924908.1924919

Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. 1995. Sharing memory robustly in message-passing systems. *Journal of the ACM* 42, 1 (1995), 124–142. DOI:http://dx.doi.org/10.1145/200836.200869

Hagit Attiya, Faith Ellen, and Adam Morrison. 2015. Limitations of highly-available eventually-consistent data stores. In *ACM Symposium on Principles of Distributed Computing (PODC'15)*. ACM, 385–394. DOI:http://dx.doi.org/10.1145/2767386.2767419

Hagit Attiya and Roy Friedman. 1992. A correctness condition for high-performance multiprocessors (extended abstract). In *ACM Symposium on Theory of Computing, 1992*. 679–690. DOI:http://dx.doi.org/10.1145/129712.129778

Hagit Attiya and Jennifer Welch. 1994. Sequential consistency versus linearizability. *ACM Transactions on Computer Systems (TOCS)* 12, 2 (May 1994), 99–122.

Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2012. The potential dangers of causal consistency and an explicit solution. In *ACM Symposium on Cloud Computing (SOCC'12)*. 22. DOI:http://dx.doi.org/10.1145/2391229.2391251

Peter Bailis, Alan Fekete, Joseph M. Hellerstein, Ali Ghodsi, and Ion Stoica. 2014. Scalable atomic visibility with RAMP transactions. In *ACM International Conference on Management of Data (SIGMOD'14)*. 27–38. DOI:http://dx.doi.org/10.1145/2588555.2588562

Peter Bailis and Ali Ghodsi. 2013. Eventual consistency today: Limitations, extensions, and beyond. *Queue* 11, 3 (March 2013), 20:20–20:32. DOI:http://dx.doi.org/10.1145/2460276.2462076

Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2013. Bolt-on causal consistency. In *ACM SIGMOD International Conference on Management of Data (SIGMOD'13)*. 761–772.

Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. 2012. Probabilistically bounded staleness for practical partial quorums. *VLDB Journal* 5, 8 (2012), 776–787.

Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. 2014. Quantifying eventual consistency with PBS. *VLDB Journal* 23, 2 (2014), 279–302. DOI:http://dx.doi.org/10.1007/s00778-013-0330-1

Jason Baker, Chris Bond, James C. Corbett, J. J. Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. 2011. Megastore: Providing scalable, highly available storage for interactive services. In *Conference on Innovative Data Systems Research (CIDR'11)*. 223–234.

Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno M. Preguiça, Mahsa Najafzadeh, and Marc Shapiro. 2015. Putting consistency back into eventual consistency. In *European Conference on Computer Systems (EuroSys'15)*. 6. DOI:http://dx.doi.org/10.1145/2741948.2741972

Jordi Bataller and José M. Bernabéu-Aubán. 1997. Synchronized DSM models. In *Parallel Processing (Euro-Par'97)*. 468–475. DOI:http://dx.doi.org/10.1007/BFb0002771

Nalini Moti Belaramani, Michael Dahlin, Lei Gao, Amol Nayate, Arun Venkataramani, Praveen Yalagandula, and Jiandan Zheng. 2006. PRACTI replication. In *Symposium on Networked Systems Design and Implementation (NSDI'06)*. http://www.usenix.org/events/nsdi06/tech/belaramani.html.

David Bermbach and Jörn Kuhlenkamp. 2013. Consistency in distributed storage systems–An overview of models, metrics and measurement approaches. In *Networked Systems (NETYS'13)*. 175–189.

David Bermbach and Stefan Tai. 2011. Eventual consistency: How soon is eventual? An evaluation of amazon S3's consistency behavior. In *Workshop on Middleware for Service Oriented Computing (MW4SOC'11)*. ACM, New York, NY, 1:1–1:6. DOI:http://dx.doi.org/10.1145/2093185.2093186

Philip A. Bernstein and Sudipto Das. 2013. Rethinking eventual consistency. In *ACM SIGMOD International Conference on Management of Data (SIGMOD'13)*. 923–928. DOI:http://dx.doi.org/10.1145/2463676.2465339

Brian N. Bershad and Matthew J. Zekauskas. 1991. *Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors*. Technical Report.

Alysson Neves Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. 2013. DepSky: Dependable and secure storage in a cloud-of-clouds. *ACM Transactions on Storage (TOS)* 9, 4 (2013), 12. DOI:http://dx.doi.org/10.1145/2535929

Alysson Neves Bessani, Ricardo Mendes, Tiago Oliveira, Nuno Ferreira Neves, Miguel Correia, Marcelo Pasin, and Paulo Veríssimo. 2014. SCFS: A shared cloud-backed file system. In *USENIX Annual Technical Conference (ATC'14)*. 169–180. https://www.usenix.org/conference/atc14/technical-sessions/presentation/bessani.

Kenneth Birman, Andre Schiper, and Pat Stephenson. 1991. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems (TOCS)* 9, 3 (1991), 272–314.

Vita Bortnikov, Gregory Chockler, Alexey Roytman, and Mike Spreitzer. 2010. Bulletin board: A scalable and robust eventually consistent shared memory over a peer-to-peer overlay. *Operating Systems Review* 44, 2 (April 2010), 64–70. DOI:http://dx.doi.org/10.1145/1773912.1773929

Marcus Brandenburger, Christian Cachin, and Nikola Knezevic. 2015. Don't trust the cloud, verify: Integrity and consistency for cloud object stores. In *ACM International Systems and Storage Conference (SYSTOR'15)*. 16:1–16:11. http://doi.acm.org/10.1145/2757667.2757681

Eric A. Brewer. 2000. Towards robust distributed systems (abstract). In *ACM Symposium on Principles of Distributed Computing (PODC'00)*. 7. DOI:http://dx.doi.org/10.1145/343477.343502

Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry C. Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkateshwaran Venkataramani. 2013. TAO: Facebook's distributed data store for the social graph. In *USENIX Annual Technical Conference (ATC'13)*. 49–60. https://www.usenix.org/conference/atc13/technical-sessions/presentation/bronson.

Jerzy Brzezinski, Cezary Sobaniec, and Dariusz Wawrzyniak. 2003. Session guarantees to achieve PRAM consistency of replicated shared objects. In *Parallel Processing and Applied Mathematics (PPAM'03)*. 1–8.

Jerzy Brzezinski, Cezary Sobaniec, and Dariusz Wawrzyniak. 2004. From session causality to causal consistency. In *Workshop on Parallel, Distributed and Network-Based Processing (PDP'04)*. 152–158. DOI:http://dx.doi.org/10.1109/EMPDP.2004.1271440

Sebastian Burckhardt. 2014. *Principles of Eventual Consistency*. Foundations and Trends in Programming Languages, Vol. 1. now publishers. 1–150 pages. http://research.microsoft.com/apps/pubs/default.aspx?id=230852.

Sebastian Burckhardt, Manuel Fähndrich, Daan Leijen, and Benjamin P. Wood. 2012. Cloud types for eventual consistency. In *Object-Oriented Programming (ECOOP'12)*. 283–307. DOI:http://dx.doi.org/10.1007/978-3-642-31057-7_14

Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. 2014. Replicated data types: Specification, verification, optimality. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'14)*. 271–284. DOI:http://dx.doi.org/10.1145/2535838.2535848

Sebastian Burckhardt, Daan Leijen, Manuel Fähndrich, and Mooly Sagiv. 2012. Eventually consistent transactions. In *European Symposium on Programming (ESOP'12), Held as Part of ETAPS 2012*. 67–86. DOI:http://dx.doi.org/10.1007/978-3-642-28869-2_4

Michael Burrows. 2006. The chubby lock service for loosely-coupled distributed systems. In *Symposium on Operating Systems Design and Implementation (OSDI'06)*. 335–350. http://www.usenix.org/events/osdi06/tech/burrows.html.

Christian Cachin, Idit Keidar, and Alexander Shraer. 2009a. Fork sequential consistency is blocking. *Information Processing Letters* 109, 7 (2009), 360–364.

Christian Cachin, Idit Keidar, and Alexander Shraer. 2009b. Trusting the cloud. *SIGACT News* 40, 2 (2009), 81–86. DOI:http://dx.doi.org/10.1145/1556154.1556173

Christian Cachin, Idit Keidar, and Alexander Shraer. 2011. Fail-aware untrusted storage. *SIAM Journal on Computing (SICOMP)* 40, 2 (2011), 493–533. DOI:http://dx.doi.org/10.1137/090751062

Christian Cachin, Abhi Shelat, and Alexander Shraer. 2007. Efficient fork-linearizable access to untrusted shared memory. In *ACM Symposium on Principles of Distributed Computing (PODC'07)*. 129–138.

Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. 2011. Windows azure storage: A highly available cloud storage service with strong consistency. In *ACM Symposium on Operating Systems Principles (SOSP'11)*. 143–157.

Andrea Cerone, Giovanni Bernardi, and Alexey Gotsman. 2015. A framework for transactional consistency models with atomic visibility. In *Conference on Concurrency Theory (CONCUR'15)*. 58–71. DOI:http://dx.doi.org/10.4230/LIPIcs.CONCUR.2015.58

Houssem-Eddine Chihoub, Shadi Ibrahim, Gabriel Antoniu, and María S. Pérez-Hernández. 2012. Harmony: Towards automated self-adaptive consistency in cloud storage. In *IEEE International Conference on Cluster Computing (CLUSTER'12)*. 293–301.

Houssem-Eddine Chihoub, Shadi Ibrahim, Gabriel Antoniu, and María S. Pérez-Hernández. 2013. Consistency in the cloud: When money does matter! In *Symposium on Cluster, Cloud, and Grid Computing (CCGrid'13)*. 352–359.

Brian A. Coan, Brian M. Oki, and Elliot K. Kolodner. 1986. Limitations on database availability when networks partition. In *ACM Symposium on Principles of Distributed Computing, 1986*. 187–194. DOI:http://dx.doi.org/10.1145/10590.10606

Neil Conway, William R. Marczak, Peter Alvaro, Joseph M. Hellerstein, and David Maier. 2012. Logic and lattices for distributed programming. In *ACM Symposium on Cloud Computing (SOCC'12)*. 1. DOI:http://dx.doi.org/10.1145/2391229.2391230

Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. 2008. PNUTS: Yahoo!'s hosted data serving platform. *VLDB Journal* 1, 2 (2008), 1277–1288.

James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2013. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)* 31, 3 (2013), 8.

Susan B. Davidson, Hector Garcia-Molina, and Dale Skeen. 1985. Consistency in partitioned networks. *Computer Surveys* 17, 3 (1985), 341–370.

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's highly available key-value store. In *ACM Symposium on Operating Systems Principles (SOSP'07)*. 205–220.

Dan Dobre, Paolo Viotti, and Marko Vukolić. 2014. Hybris: Robust hybrid cloud storage. In *ACM Symposium on Cloud Computing (SOCC'14)*. 12:1–12:14. DOI:http://dx.doi.org/10.1145/2670979.2670991

Jiaqing Du, Calin Iorgulescu, Amitabha Roy, and Willy Zwaenepoel. 2014. GentleRain: Cheap and scalable causal consistency with physical clocks. In *ACM Symposium on Cloud Computing (SOCC'14)*. 4:1–4:13. DOI:http://dx.doi.org/10.1145/2670979.2670983

Michel Dubois, Christoph Scheurich, and Faye A. Briggs. 1986. Memory access buffering in multiprocessors. In *International Symposium on Computer Architecture (ISCA'86)*. 434–442.

Dmytro Dziuma, Panagiota Fatourou, and Eleni Kanellou. 2014. Consistency for transactional memory computing. *Bulletin of the EATCS* 113 (2014). http://eatcs.org/beatcs/index.php/beatcs/article/view/288.

Alan Fekete, David Gupta, Victor Luchangco, Nancy A. Lynch, and Alexander A. Shvartsman. 1996. Eventually-serializable data services. In *ACM Symposium on Principles of Distributed Computing, 1996*. 300–309. DOI:http://dx.doi.org/10.1145/248052.248113

Ariel J. Feldman, William P. Zeller, Michael J. Freedman, and Edward W. Felten. 2010. SPORC: Group collaboration using untrusted cloud resources. In *Symposium on Operating Systems Design and Implementation (OSDI'10)*. 337–350. http://www.usenix.org/events/osdi10/tech/full_papers/Feldman.pdf.

Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. 1985. Impossibility of distributed consensus with one faulty process. *Journal of the ACM* 32, 2 (1985), 374–382. DOI:http://dx.doi.org/10.1145/3149.214121

Roy Friedman, Roman Vitenberg, and Gregory Chockler. 2003. On the composability of consistency conditions. *Information Processing Letters* 86, 4 (2003), 169–176. DOI:http://dx.doi.org/10.1016/S0020-0190(02)00498-2

Guang R. Gao and Vivek Sarkar. 2000. Location consistency-A new memory model and cache consistency protocol. *IEEE Transactionas on Computers* 49, 8 (2000), 798–813. DOI:http://dx.doi.org/10.1109/12.868026

Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip B. Gibbons, Anoop Gupta, and John L. Hennessy. 1990. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *International Symposium on Computer Architecture*. 15–26.

Phillip B. Gibbons and Ephraim Korach. 1997. Testing shared memories. *SIAM Journal on Computing (SICOMP)* 26, 4 (1997), 1208–1244. DOI:http://dx.doi.org/10.1137/S0097539794279614

Seth Gilbert and Nancy A. Lynch. 2002. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News* 33, 2 (2002), 51–59.

Lisa Glendenning, Ivan Beschastnikh, Arvind Krishnamurthy, and Thomas E. Anderson. 2011. Scalable consistency in scatter. In *ACM Symposium on Operating Systems Principles (SOSP'11)*. 15–28.

Wojciech M. Golab, Xiaozhou Li, and Mehul A. Shah. 2011. Analyzing consistency properties for fun and profit. In *ACM Symposium on Principles of Distributed Computing (PODC'11)*. 197–206.

Wojciech M. Golab, Muntasir Raihan Rahman, Alvin AuYoung, Kimberly Keeton, and Indranil Gupta. 2014. Client-centric benchmarking of eventual consistency for cloud storage systems. In *IEEE Conference on Distributed Computing Systems (ICDCS'14)*. 493–502. DOI:http://dx.doi.org/10.1109/ICDCS.2014.57

James R. Goodman. 1989. *Cache Consistency and Sequential Consistency*. Technical Report no. 61. SCI Commitee.

Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 2016. 'Cause I'm strong enough: Reasoning about consistency choices in distributed systems. In *ACM SIGPLAN-SIGACT*

Symposium on Principles of Programming Languages (POPL'16). http://lip6.fr/Marc.Shapiro/papers/CISE-POPL-2016.pdf.

Rachid Guerraoui and Marko Vukolic. 2006. How fast can a very robust read be? In *ACM Symposium on Principles of Distributed Computing (PODC'06)*. 248–257. DOI:http://dx.doi.org/10.1145/1146381.1146419

Vassos Hadzilacos and Sam Toueg. 1994. *A Modular Approach to Fault-Tolerant Broadcasts and Related Problems*. Technical Report. Cornell University, Department of Computer Science.

Seungyeop Han, Haichen Shen, Taesoo Kim, Arvind Krishnamurthy, Thomas E. Anderson, and David Wetherall. 2015. MetaSync: File synchronization across multiple untrusted storage services. In *USENIX Annual Technical Conference (ATC'15)*. 83–95. https://www.usenix.org/conference/atc15/technical-session/presentation/han.

Tim Harris, James R. Larus, and Ravi Rajwar. 2010. *Transactional Memory*, 2nd ed. Morgan & Claypool Publishers. DOI:http://dx.doi.org/10.2200/S00272ED1V01Y201006CAC011

Pat Helland. 2007. Life beyond distributed transactions: An Apostate's opinion. In *Conference on Innovative Data Systems Research (CIDR'07)*. 132–141. http://www.cidrdb.org/cidr2007/papers/cidr07p15.pdf.

Maurice Herlihy and Nir Shavit. 2008. *The Art of Multiprocessor Programming*. Morgan Kaufmann. I–XX, 1–508 pages.

Maurice Herlihy and Jeannette M. Wing. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12, 3 (1990), 463–492.

Phillip W. Hutto and Mustaque Ahamad. 1990. Slow memory: Weakening consistency to enhance concurrency in distributed shared memories. In *International Conference on Distributed Computing Systems (ICDCS'90)*. 302–309.

Liviu Iftode, Cezary Dubnicki, Edward W. Felten, and Kai Li. 1996a. Improving release-consistent shared virtual memory using automatic update. In *IEEE International Symposium on High-Performance Computer Architecture (HPCA'96)*. 14–25. DOI:http://dx.doi.org/10.1109/HPCA.1996.501170

Liviu Iftode, Jaswinder Pal Singh, and Kai Li. 1996b. Scope consistency: A bridge between release consistency and entry consistency. In *ACM Symposium on Parallel Algorithms and Architectures (SPAA'96), 1996*. 277–287.

Paul R. Johnson and Robert H. Thomas. 1975. *Maintenance of Duplicate Databases*. RFC 677. RFC Editor. http://www.rfc-editor.org/rfc/rfc677.txthttp://www.rfc-editor.org/rfc/rfc677.txt.

Peter J. Keleher, Alan L. Cox, and Willy Zwaenepoel. 1992. Lazy release consistency for software distributed shared memory. In *International Symposium on Computer Architecture, 1992*. 13–21. DOI:http://dx.doi.org/10.1145/139669.139676

Tim Kraska, Martin Hentschel, Gustavo Alonso, and Donald Kossmann. 2009. Consistency rationing in the cloud: Pay only when it matters. *VLDB Journal* 2, 1 (2009), 253–264.

Sudha Krishnamurthy, William H. Sanders, and Michel Cukier. 2002. An adaptive framework for tunable consistency and timeliness using replication. In *Dependable Systems and Networks (DSN'02)*. 17–26.

Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. 1992. Providing high availability using lazy replication. *ACM Transactions on Computer Systems (TOCS)* 10, 4 (1992), 360–391. DOI:http://dx.doi.org/10.1145/138873.138877

Avinash Lakshman and Prashant Malik. 2010. Cassandra: A decentralized structured storage system. *Operating Systems Review* 44, 2 (2010), 35–40. DOI:http://dx.doi.org/10.1145/1773912.1773922

Subramanian Lakshmanan, Mustaque Ahamad, and H. Venkateswaran. 2001. A secure and highly available distributed store for meeting diverse data storage needs. In *Dependable Systems and Networks (DSN'01)*. 251–260. DOI:http://dx.doi.org/10.1109/DSN.2001.941410

Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM (CACM)* 21, 7 (1978), 558–565.

Leslie Lamport. 1979. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers* 28, 9 (1979), 690–691.

Leslie Lamport. 1983. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 5, 2 (1983), 190–222. DOI:http://dx.doi.org/10.1145/69624.357207

Leslie Lamport. 1986a. On interprocess communication. Part I: Basic formalism. *Distributed Computing* 1, 2 (1986), 77–85.

Leslie Lamport. 1986b. On interprocess communication. Part II: Algorithms. *Distributed Computing* 1, 2 (1986), 86–101.

Leslie Lamport. 2001. Paxos made simple. *SIGACT News* 32, 4 (2001), 51–58. DOI:http://dx.doi.org/10.1145/568425.568433

Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. 1982. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 4, 3 (1982), 382–401. DOI:http://dx.doi.org/10.1145/357172.357176

Collin Lee, Seo Jin Park, Ankita Kejriwal, Satoshi Matsushita, and John K. Ousterhout. 2015. Implementing linearizability at large scale and low latency. In *ACM Symposium on Operating Systems Principles (SOSP'15)*. 71–86. DOI:http://dx.doi.org/10.1145/2815400.2815416

Mohsen Lesani, Christian J. Bell, and Adam Chlipala. 2016. Chapar: Certified causally consistent distributed key-value stores. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'16)*. http://doi.acm.org/10.1145/2837614.2837622

Cheng Li, João Leitão, Allen Clement, Nuno M. Preguiça, Rodrigo Rodrigues, and Viktor Vafeiadis. 2014. Automating the choice of consistency levels in replicated systems. In *USENIX Annual Technical Conference (ATC'14)*. 281–292. https://www.usenix.org/conference/atc14/technical-sessions/presentation/li_cheng_2.

Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. 2012. Making geo-replicated systems fast as possible, consistent when necessary. In *Symposium on Operating Systems Design and Implementation (OSDI'12)*. USENIX Association, Berkeley, CA, 265–278. http://dl.acm.org/citation.cfm?id=2387880.2387906

Jinyuan Li, Maxwell N. Krohn, David Mazières, and Dennis Shasha. 2004. Secure untrusted data repository (SUNDR). In *Symposium on Operating System Design and Implementation (OSDI'04)*. 121–136.

Jinyuan Li and David Mazières. 2007. Beyond one-third faulty replicas in Byzantine fault tolerant systems. In *Symposium on Networked Systems Design and Implementation (NSDI'07)*.

Richard J. Lipton and Jonathan S. Sandberg. 1988. *PRAM: A Scalable Shared Memory*. Technical Report CS-TR-180-88. Princeton University.

Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2011. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *ACM Symposium on Operating Systems Principles (SOSP'11)*. 401–416.

Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2013. Stronger semantics for low-latency geo-replicated storage. In *Symposium on Networked Systems Design and Implementation (NSDI'13)*. USENIX Association, Berkeley, CA, 313–328. http://dl.acm.org/citation.cfm?id=2482626.2482657

Haonan Lu, Kaushik Veeraraghavan, Philippe Ajoux, Jim Hunt, Yee Jiun Song, Wendy Tobagus, Sanjeev Kumar, and Wyatt Lloyd. 2015. Existential consistency: Measuring and understanding consistency at facebook. In *ACM Symposium on Operating Systems Principles (SOSP'15)*. 295–310. DOI:http://dx.doi.org/10.1145/2815400.2815426

Nancy A. Lynch and Mark R. Tuttle. 1989. An introduction to input/output automata. *CWI Quarterly* 2 (1989), 219–246.

Prince Mahajan, Lorenzo Alvisi, and Mike Dahlin. 2011. *Consistency, Availability, and Convergence*. Technical Report TR-11-22. Computer Science Department, University of Texas at Austin.

Prince Mahajan, Srinath T. V. Setty, Sangmin Lee, Allen Clement, Lorenzo Alvisi, Michael Dahlin, and Michael Walfish. 2010. Depot: Cloud storage with minimal trust. In *Symposium on Operating Systems Design and Implementation (OSDI'10)*. 307–322.

Dahlia Malkhi and Michael K. Reiter. 1998a. Byzantine quorum systems. *Distributed Computing* 11, 4 (1998), 203–213. DOI:http://dx.doi.org/10.1007/s004460050050

Dahlia Malkhi and Michael K. Reiter. 1998b. Secure and scalable replication in phalanx. In *Symposium on Reliable Distributed Systems (SRDS'98)*. 51–58. DOI:http://dx.doi.org/10.1109/RELDIS.1998.740474

David Mazières and Dennis Shasha. 2002. Building secure file systems out of Byzantine storage. In *ACM Symposium on Principles of Distributed Computing (PODC'02)*. 108–117.

Jayadev Misra. 1986. Axioms for memory access in asynchronous hardware systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 8, 1 (1986), 142–153.

Iulian Moraru, David G. Andersen, and Michael Kaminsky. 2013. There is more consensus in Egalitarian parliaments. In *ACM Symposium on Operating Systems Principles (SOSP'13)*. 358–372. DOI:http://dx.doi.org/10.1145/2517349.2517350

David Mosberger. 1993. Memory consistency models. *Operating Systems Review* 27, 1 (1993), 18–26. DOI:http://dx.doi.org/10.1145/160551.160553

Alina Oprea and Michael K. Reiter. 2006. On consistency of encrypted files. In *Distributed Computing (DISC'06)*. 254–268.

Swapnil Patil, Milo Polte, Kai Ren, Wittawat Tantisiriroj, Lin Xiao, Julio López, Garth Gibson, Adam Fuchs, and Billie Rinaldi. 2011. YCSB++: Benchmarking and performance debugging advanced features in

scalable table stores. In *ACM Symposium on Cloud Computing (SOCC'11) in Conjunction with SOSP 2011*. 9. DOI:http://dx.doi.org/10.1145/2038916.2038925

Dorian Perkins, Nitin Agrawal, Akshat Aranya, Curtis Yu, Younghwan Go, Harsha V. Madhyastha, and Cristian Ungureanu. 2015. Simba: Tunable end-to-end data consistency for mobile apps. In *European Conference on Computer Systems (EuroSys'15)*. 7. DOI:http://dx.doi.org/10.1145/2741948.2741974

Karin Petersen, Mike Spreitzer, Douglas B. Terry, Marvin Theimer, and Alan J. Demers. 1997. Flexible update propagation for weakly consistent replication. In *ACM Symposium on Operating Systems Principles (SOSP'97)*. 288–301. DOI:http://dx.doi.org/10.1145/268998.266711

Muntasir Raihan Rahman, Wojciech M. Golab, Alvin AuYoung, Kimberly Keeton, and Jay J. Wylie. 2012. Toward a principled framework for benchmarking consistency. *Computing Research Repository* abs/1211.4290 (2012).

Jun Rao, Eugene J. Shekita, and Sandeep Tata. 2011. Using paxos to build a scalable, consistent, and highly available datastore. *VLDB Journal* 4, 4 (2011), 243–254.

Michel Raynal and André Schiper. 1997. A suite of definitions for consistency criteria in distributed shared memories. *Annales des Télécommunications* 52, 11–12 (1997), 652–661. DOI:http://dx.doi.org/10.1007/BF02997620

Peter L. Reiher, John S. Heidemann, David Ratner, Gregory Skinner, and Gerald J. Popek. 1994. Resolving file conflicts in the ficus file system. In *USENIX Summer 1994 Technical Conference, 1994*. 183–195. https://www.usenix.org/conference/usenix-summer-1994-technical-conference /resolving-file-conflicts-ficus-file-system.

Hyun-Gul Roh, Myeongjae Jeon, Jinsoo Kim, and Joonwon Lee. 2011. Replicated abstract data types: Building blocks for collaborative applications. *Journal of Parallel and Distributed Computing* 71, 3 (2011), 354–368. DOI:http://dx.doi.org/10.1016/j.jpdc.2010.12.006

Yasushi Saito and Marc Shapiro. 2005. Optimistic replication. *Computer Surveys* 37, 1 (2005), 42–81.

Nuno Santos, Luís Veiga, and Paulo Ferreira. 2007. Vector-field consistency for ad-hoc gaming. In *ACM/IFIP/USENIX Middleware Conference, 2007*. 80–100.

Marco Serafini, Dan Dobre, Matthias Majuntke, Péter Bokor, and Neeraj Suri. 2010. Eventually linearizable shared objects. In *ACM Symposium on Principles of Distributed Computing (PODC'10)*. 95–104.

Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. 2011a. Conflict-free replicated data types. In *Stabilization, Safety, and Security of Distributed Systems (SSS'11)*. 386–400. DOI:http://dx.doi.org/10.1007/978-3-642-24550-3_29

Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. 2011b. Convergent and commutative replicated data types. *Bulletin of the EATCS* 104 (2011), 67–88.

Alexander Shraer, Christian Cachin, Asaf Cidon, Idit Keidar, Yan Michalevsky, and Dani Shaket. 2010. Venus: Verification for untrusted cloud storage. In *ACM Cloud Computing Security Workshop (CCSW'10)*. 19–30.

Atul Singh, Pedro Fonseca, Petr Kuznetsov, Rodrigo Rodrigues, and Petros Maniatis. 2009. Zeno: Eventually consistent Byzantine-fault tolerance. In *Symposium on Networked Systems Design and Implementation (NSDI'09)*. 169–184. http://www.usenix.org/events/nsdi09/tech/full_papers/singh/singh.pdf.

Aman Singla, Umakishore Ramachandran, and Jessica K. Hodgins. 1997. Temporal notions of synchronization and consistency in beehive. In *ACM Symposium on Parallel Algorithms and Architectures (SPAA'97)*. 211–220. DOI:http://dx.doi.org/10.1145/258492.258513

Krishnamoorthy C. Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. 2015. Declarative programming over eventually consistent data stores. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, 2015*. 413–424. DOI:http://dx.doi.org/10.1145/2737924.2737981

Robert C. Steinke and Gary J. Nutt. 2004. A unified theory of shared memory consistency. *Journal of the ACM* 51, 5 (2004), 800–849. DOI:http://dx.doi.org/10.1145/1017460.1017464

Andrew S. Tanenbaum and Maarten van Steen. 2007. *Distributed Systems—Principles and Paradigms*, 2nd ed. Pearson Education. I–XVIII, 1–686 pages.

Doug Terry. 2013. Replicated data consistency explained through baseball. *Communications of the ACM (CACM)* 56, 12 (2013), 82–89.

Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, and Brent B. Welch. 1994. Session guarantees for weakly consistent replicated data. In *Parallel and Distributed Information Systems (PDIS'94)*. 140–149.

Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. 2013. Consistency-based service level agreements for cloud storage. In *ACM Symposium on Operating Systems Principles (SOSP'13)*. 309–324.

Douglas B. Terry, Marvin Theimer, Karin Petersen, Alan J. Demers, Mike Spreitzer, and Carl Hauser. 1995. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *ACM Symposium on Operating Systems Principles (SOSP'95)*. 172–183. DOI: http://dx.doi.org/10.1145/224056.224070

Francisco J. Torres-Rojas, Mustaque Ahamad, and Michel Raynal. 1999. Timed consistency for shared distributed objects. In *ACM Symposium on Principles of Distributed Computing (PODC'99)*. 163–172.

Francisco J. Torres-Rojas and Esteban Meneses. 2005. Convergence through a weak consistency model: Timed causal consistency. *CLEI Electronic Journal* 8, 2 (2005).

Werner Vogels. 2008. Eventually consistent. *Queue* 6, 6 (Oct. 2008), 14–19. DOI: http://dx.doi.org/10.1145/1466443.1466448

Marko Vukolić. 2010. The Byzantine empire in the intercloud. *SIGACT News* 41, 3 (Sept. 2010), 105–111. DOI: http://dx.doi.org/10.1145/1855118.1855137

Hiroshi Wada, Alan Fekete, Liang Zhao, Kevin Lee, and Anna Liu. 2011. Data consistency properties and the trade-offs in commercial cloud storage: The consumers' perspective. In *Conference on Innovative Data Systems Research (CIDR'11)*. 134–143.

Zhe Wu, Michael Butkiewicz, Dorian Perkins, Ethan Katz-Bassett, and Harsha V. Madhyastha. 2013. *SPANStore*: Cost-effective geo-replicated storage spanning multiple cloud services. In *ACM Symposium on Operating Systems Principles (SOSP'13)*. 292–308.

Haifeng Yu and Amin Vahdat. 2002. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Transactions on Computer Systems (TOCS)* 20, 3 (2002), 239–282.

Marek Zawirski, Nuno Preguiça, Sérgio Duarte, Annette Bieniusa, Valter Balegas, and Marc Shapiro. 2015. Write fast, read in the past: Causal consistency for client-side applications. In *ACM/IFIP/USENIX Middleware Conference, 2015*.

Received December 2015; revised March 2016; accepted April 2016