Abstract Specifications for Weakly Consistent Data

Sebastian Burckhardt
Microsoft Research
Redmond, WA, USA

Jonathan Protzenko
Microsoft Research
Redmond, WA, USA

Abstract

Weak consistency can improve availability and performance when replicating data across slow or intermittent connections, but is difficult to describe abstractly and precisely. We shine a spotlight on recent results in this area, in particular the use of abstract executions *to specify the behavior of replicated objects or stores. Abstract executions are directed acyclic graphs of operations, ordered by visibility and arbitration. This specification approach has enabled several interesting theoretical results, including systematic investigations of the correctness and optimality of replicated data types, and has shed more light on the nature of the fundamental trade-off between availability and consistency.*

1 Introduction

Many organizations operate services that are used by a large number of clients connecting from a range of devices, possibly on multiple continents. Unfortunately, performance and availability of such services can suffer if the network connections (either between servers, or between the servers and the clients) are slow or intermittent. A natural solution is to replicate data. For example, clients may cache a replica on the device so it remains accessible even if communication with the server is temporarily unavailable. Or, data may be geo-replicated across data-centers to maintain low access times and preserve availability under network partitions when serving clients in multiple continents.

Unfortunately, this plan introduces some amount of inconsistency: if a device is operating offline, the device-local replica can temporarily diverge from the server replica; and if we want to avoid waiting for slow intercontinental communication whenever we read or write, then the replicas in multiple continents are not completely synchronized at all times.

The crux is that in any distributed system with slow or intermittent communication, the replication of data that is both read and written is inherently a double-edged sword. The positive effects (improved availability and latency) cannot effectively be separated from the negative effects (expose the application to inconsistent states). Where availability and latency are crucial, we must thus face the challenge of weak consistency. This well-known fact has been popularized as a fundamental problem by the CAP theorem [3, 11]. In the context of cloud storage, Amazon's Dynamo system [13] has garnered much attention and encouraged other storage solutions supporting weak consistency [2, 19] that are in common use today.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

1.1 The Specification Problem

Despite their popularity, weakly consistent storage systems are difficult to use. Particularly frustrating is the lack of useful specifications. Typical specifications are either too weak, or too detailed, and usually too vague, to serve their intended purpose: giving the programmer a useful, simple way to visualize the behavior of the system. Consider the notion of *eventual consistency*, which was pioneered by the Bayou system [37] in the context of replicas on mobile devices. It specifies that

> if clients stop issuing update requests, then the replicas will eventually reach a consistent state.

This specification is not actually strong enough to be useful. For one, it is not enough to reach an arbitrary consistent state: this state also must make sense in the context of what operations were performed (for example, one would expect a key-value store to only contain values that were actually written). Similarly, it matters what values are observed not just after, but also before convergence is achieved. Finally, a useful system must guarantee sensible behavior even if updates never stop, which is typical for online services.

In the absence of a clear, abstract specification, application programmers typically base their design on a vague description of some replication algorithm, including numerous low-level details about message protocols, quorum sizes, and background processes. This lack of abstraction makes it difficult to ensure that the application program functions correctly, and leads to non-portable, hard-to-read application code that heavily depends on the particulars of the chosen system.

1.2 Overview

The goal of this article is to put a spotlight on recent, theoretically-oriented work that addresses the “specification problem” for weakly consistent data. We explain the key concept of *abstract executions* that enable us to separate the “what” from the “how”. Abstract executions generalize operation sequences (which are suitable for specifying the semantics of single-copy, strongly consistent data) to graphs of operations ordered by visibility and arbitration relations (which are suitable for specifying the semantics of replicated, highly available data). We then mention a few results that demonstrate how the use of abstract executions for devising specifications has opened the door to a systematic study of implementations of replicated data types, including correctness and optimality proofs.

2 Abstract Executions

A specification of a data object or a data store must describe what data is being stored, and how it is being read and updated. We do not distinguish between objects and stores here, but treat them both simply as an abstract data type. First, we introduce two sets Val and Op as follows.

Val contains all data *values* we may ever want to use. For example, Val may include both simple data types such as strings, integers, or booleans, and structured data such as arrays, records, relational databases, or XML documents.

Op contains all *operations* of all data types we may ever want to use. Op may range from simple read and write operations to operations that read and/or write structured data, and may include application-defined operations such as stored procedures.

Operations can include input parameters, and all operations return a value (sometimes a simple ok to acknowledge completion). For an operation $o \in \mathsf{Op}$ and value $v \in \mathsf{Val}$ we write $o : v$ to represent the *invocation* of the operation together with the returned value. For example, when accessing a key-value store, we may observe the following sequence of invocations:

$$wr(A,1) : \mathsf{ok}, \quad wr(B,1) : \mathsf{ok}, \quad rd(A) : 1, \quad wr(B,2) : \mathsf{ok}, \quad rd(B) : 2$$

2.1 Sequential Data Types

Invocation sequences capture the possible sequential interactions with the object. We can use such sequences to define the observable behavior of the object as a black-box.

Definition 1: A sequential data type specification is a set of invocation sequences.

Example 1: Define the **register** data type to contain the sequences where (a) each invocation is of the form $wr(v):\texttt{ok}$ or $rd:v$ (where v ranges over some set of values), and (b) each read returns the value last written, or `null` if the location has not been written.

Example 2: Define the **counter** data type with an increment operation *inc* that returns the updated counter value as the set of all sequences of the form $inc:1, inc:2, \cdots inc:n$ (for $n \in \mathbb{N}$).

Example 3: Define the **key-value store** data type to contain the sequences where (a) each invocation is of the form $wr(l,v):\texttt{ok}$ or $rd(l):v$ (where l ranges over some set of keys and v ranges over some set of values), and (b) each read returns the value last written to the same location, or `null` if the location has not been written.

Sequential data types are also used to define strong consistency models. Sequential consistency [15] and Linearizability [12] both require that for all concurrent interactions with an object, there exists a sequential witness that "justifies" the observed values. Thus, any system guaranteeing strong consistency must somehow guarantee a globally consistent linear order of operations.

2.2 Replicated Data Types

What distinguishes a replicated data type [6] from a sequential or concurrent data type is that *all operations are always available* regardless of temporary network partitions. Concretely, it requires that, at any location and time, any operation can be issued and returns a value immediately. All communication is asynchronous: the propagation of updates is a background process that takes a variable amount of time to complete.

Abstract Executions. For a replicated data type, a simple invocation sequence is not sufficient to define its semantics. For example, if two different replicas simultaneously update the data, the version history forks and is no longer linear. Thus, the key to a specification of a replicated data type is to generalize from an invocation *sequence* to an invocation *graph*, which we call an *abstract execution*. We define abstract executions as directed acyclic graphs like the ones shown in Fig. 1: each vertex is an invocation, edges represent *visibility*, and the subscripts represent *arbitration* timestamps.[1] The generalization of Definition 1 is now straightforward.

Definition 2: A replicated data type specification is a set of abstract executions.

Instead of ordering all invocations as a sequence, an abstract execution graph contains two ordering relations on the vertices, visibility (which is partial, and shown as arrows) and arbitration (which is total, and shown as timestamps). Roughly, visibility captures the fact that update propagation is not instantaneous, and arbitration is used for resolving conflicts consistently.

Consider a counter as in Example 2 that provides a single increment operation *inc* that returns the updated value. Then, a possible abstract execution could look as in Fig. 1(a). The idea is that the current value of the counter at any point of the execution is the number of visible increments (and thus zero if no increments are visible).

Example 4: Define the **replicated counter** data type to contain all abstract executions such that the value returned by each increment matches the number of increments that are visible to it, plus one.

[1] For formal definitions of abstract executions see [4, 6, 1].

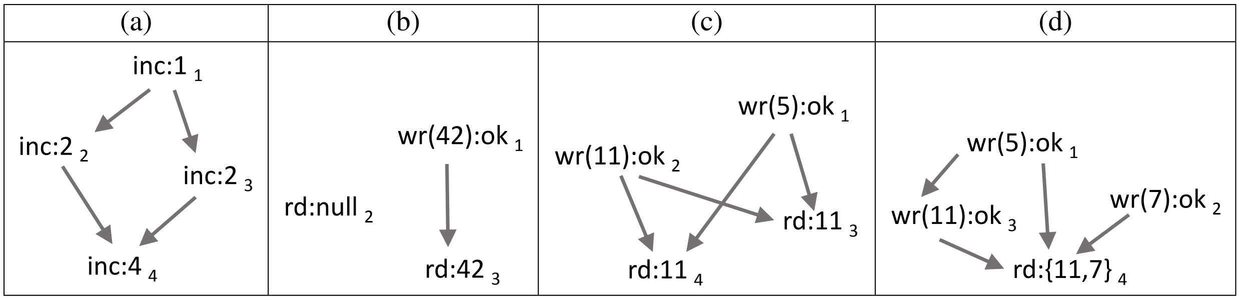

Figure 1: Three abstract executions; (a) an execution of the replicated counter, (b, c) executions of the last-writer-wins register, (d) an execution of the multi-value register

Note that the arbitration order does not appear in the definition. It is in fact irrelevant here: its function is to resolve conflicts, but for the counter, all operations are commutative, therefore we do not need to make use of the arbitration order at all. However, this is not true for all data types. Consider a register as in Example 1. Two possible abstract execution are shown in Fig. 1(b,c). Note that if more than one write is visible to a read then we must consistently pick a winner. This is what the arbitration can do for us: if there are multiple visible writes, the one with the highest timestamp wins.

Example 5: Define the **last-writer-wins register** data type to contain all abstract executions such that the value returned by each read matches the value written by the visible write with maximal arbitration, or `null` if no writes are visible.

We can think of visibility and arbitration as two aspects of the execution that were synonymous in sequential executions, but are no longer the same for replicated data types. Specifically, even if an operation *A* is ordered before operation *B* by its arbitration timestamp, this does not necessarily imply that *A* is also visible to *B*.

Both the replicated counter and the last-writer-wins register take a very simple approach to conflict resolution. For the former, conflicts are irrelevant, and for the latter, it is assumed that the latest write is the only one that matters. However, this may not be enough. Sometimes, conflicts may need to be detected and handled in an application-specific way. The multi-value register data type (introduced by Dynamo [13]) achieves this by modifying the semantics of a read operation: instead of returning just the latest write, it returns all conflicting writes. An example execution is shown in Fig. 1(d).

Example 6: Define the **multi-value register** data type to contain all abstract executions such that each read returns the set of values written by visible writes that are maximal, i.e. which are not visible to any later visible write.

A multi-value register can be used by the application programmer to detect conflicting writes; once a conflict is detected, it can be resolved by writing a new value. It is typically offered as part of a key-value store, each value being a multi-value register.

2.3 System Guarantees

Abstract executions allow us to clearly articulate the values an operation may or may not return in a given context. Beyond that, they also allow us to formalize system-wide safety and liveness guarantees. Thus, we can precisely describe the behavior of a large class of differing implementations and consistency protocols.

Eventual Visibility. Perhaps the most important liveness property we expect from a replicated data type is that *all operations become eventually visible*. Technically, we can specify this by requiring that in all infinite abstract executions, all operations are visible to all but finitely many operations. Note that this definition is effective even in executions where updates never stop.

Ordering Guarantees. To write correct applications, we may want to rely on some basic ordering guarantees. For example, we may expect that an operation performed at some replica is visible to all subsequent operations at the same replica. This is sometimes called the read-my-writes guarantee [17].

Causality. Causality means that if operation A is visible to operation B, and operation B is visible to operation C, then operation A should be visible to operation C as well. Thus, it corresponds simply to transitivity of the visibility relation. Typically, causal consistency is defined in a way that also implies the read-my-writes guarantee.

Multiple vs. single objects. Consistency guarantees may vary depending on whether we consider an individual object, or a store containing multiple objects. For example, some stores may guarantee causality between all accesses to one object, but not for accesses across multiple objects.

3 Protocols

Abstract executions represent the propagation of updates using an abstract visibility relation, without fixing a particular relationship to the underlying message protocols. This means that the same specifications can be implemented by a wide variety of protocol styles. The following categories are used to organize the replication protocol examples in [4].

Centralized. In a centralized protocol, replication is asymmetric, using a primary replica (perhaps on some highly reliable infrastructure) and secondary replicas (perhaps cheaper and less reliable). Eventual visibility in that case means that all updates are sent to the primary, and then back out to the secondaries.

Operation-Based. In an operation-based protocol, all replicas are equal. After performing an update operation, a replica broadcasts information about the operation to all other replicas. All operations are delivered to all replicas eventually, which guarantees eventual visibility. However, messages are not delivered in a consistent order.

Epidemic. In an epidemic protocol, all replicas are equal. However, replicas do not broadcast individual messages for each operation. Rather, they periodically send messages containing their entire state, which summarizes the effect of *all* operations they are aware of (thus, this is sometimes called a *state-based* protocol). Visibility of operations can thus spread via other nodes (indirectly like an infectious disease). Eventual visibility can be guaranteed even if many messages are lost, as long as the network remains fully connected.

4 Results

Abstract executions as a specification methodology have enabled systematic research on weakly consistent data, and led to several theoretical results (both positive and negative).

Correctness Proofs. One can prove that a protocol implementation satisfies a specification by showing that for each concrete execution, there exists a corresponding abstract execution that satisfies the data type specification, eventual visibility, and any other system guarantees we wish to validate. This approach is used in [4, 6] to prove correctness of several optimized protocols that implement various replicated data types (counters, registers, multi-value registers, and key-value stores) using various architectures (centralized, broadcast-based, and epidemic) and for differing system guarantees (causality, session guarantees).

Bounds on Metadata Complexity. An important design consideration is the space required by *metadata*. We define the *metadata overhead* of a replica to be the ratio of the size of all information stored, divided by the size of the current data content. In Burckhardt et al. [6], lower bounds for the worst-case metadata overhead are derived; specifically it is shown that epidemic counters, epidemic registers, and epidemic multi-value registers have a worst-case metadata overhead of at least $\Omega(n)$, $\Omega(\lg m)$, and $\Omega(n \lg m)$, respectively (where n is the number of replicas and m is the number of operations performed). Since these bounds match the metadata overhead of the best known implementations, we know that they are asymptotically optimal.

Strongest Always-Available Consistency. Another interesting question is how to maximally strengthen the consistency guarantees without forsaking availability. In [1], it is shown that a key-value store with multi-value-registers cannot satisfy a stronger consistency guarantee than observable causal consistency (OCC), a slightly stronger variant of causal consistency.

5 Living with Weak Guarantees

Many storage systems provide only the bare minimum: eventual visibility, without transaction support, and without inter-object causality. In such cases, it can be daunting to ensure applications preserve even basic data integrity. However, data abstraction can go a long way. For example, simple collection types such as sets and lists may be supported as replicated data types.

Even more powerful is the concept that application programmers can raise the abstraction level further, by defining their own custom data types. Taken to the extreme, we could consider the entire store to be a single database object, with operations defined by stored procedures, and with update propagation corresponding to queued transaction processing [2].

Operational Consistency Models. So far, we have defined replicated data types as a set of abstract executions, and have defined those sets by stating the properties its members must satisfy. This is known as the *axiomatic* approach. Another common approach is to use *operational* consistency models, i.e. define the set of possible executions by specifying an abstract machine that generates them. Operational models can augment our intuitions significantly, and make good mental reference models for visualizing executions. For example, the popular TSO memory model can be defined in both ways, axiomatically, or as a simple operational model [8]. The global sequence protocol (GSP) operational model [7] is an adaptation of the TSO model that defines a generic replicated data type, supporting both weak and strong consistency, using a simple centralized protocol.

6 Related Work

Protocols for replication have been around for a long time; recent interest in specialized protocols was spurred by the comprehensive collection by Shapiro et al. [16]. It describes counters, registers, multi-value registers, and a few other data types, and introduces the distinction between operation-based and state-based protocols. It does not however specify the behavior of these implementations abstractly.

The use of event graphs and relations for specifying consistency models can be traced back to the common practice of axiomatic memory consistency models, with a long history of publications that we shall elide here as they are only marginally relevant in this context. We refer the interested reader to [6] for some comparisons to the C++ memory model.

The use of a visibility relation is similar to the use of justification sets by Fekete et al. [9]. The use of both visibility and arbitration relations appears first in Burckhardt et al. [5], in the context of defining eventually consistent transactions. This basic idea was later significantly expanded and used to specify replicated data types, verify implementations, and prove optimality [6]. The latter paper also coins the term “abstract execution”. An equivalent formalization is used in [1], where arbitration is represented implicitly as a sequence. A slight generalization of abstract executions that can capture both weak and strong consistency models appears in [4], which also contains a variety of protocol examples and correctness proofs.

References

[1] H. Attiya, F. Ellen, and A. Morrison. Limitations of highly-available eventually-consistent data stores. In *Principles of Distributed Computing (PODC)*, pages 385–394, 2015.

[2] P. A. Bernstein and E. Newcomer. *Principles of Transaction Processing*. Morgan Kaufmann, 2nd ed. edition, 2009.

[3] E. A. Brewer. Towards robust distributed systems (abstract). In *Principles of Distributed Computing (PODC)*, 2000.

[4] S. Burckhardt. *Principles of Eventual Consistency*. Foundations and Trends in Programming Languages, 2014.

[5] S. Burckhardt, M. Fähndrich, D. Leijen, and M. Sagiv. Eventually consistent transactions. In *European Symposium on Programming (ESOP), (extended version available as Microsoft Tech Report MSR-TR-2011-117), LNCS*, volume 7211, pages 64–83, 2012.

[6] S. Burckhardt, A. Gotsman, H. Yang, and M. Zawirski. Replicated data types: Specification, verification, optimality. *SIGPLAN Not.*, 49(1):271–284, Jan. 2014.

[7] S. Burckhardt, D. Leijen, J. Protzenko, and M. Fähndrich. Global sequence protocol: A robust abstraction for replicated shared state. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 568–590, 2015.

[8] D. Dill, S. Park, and A. Nowatzyk. Formal specification of abstract memory models. In *Symposium on Research on Integrated Systems*, pages 38–52. MIT Press, 1993.

[9] A. D. Fekete and K. Ramamritham. Consistency models for replicated data. In *Replication*, volume 5959 of *LNCS*, pages 1–17. Springer, 2010.

[10] G. DeCandia et al. Dynamo: Amazon's highly available key-value store. In *Symposium on Operating Systems Principles (SOSP)*, 2007.

[11] S. Gilbert and N. A. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33:51–59, June 2002.

[12] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[13] R. Klophaus. Riak core: Building distributed applications without shared state. In *Commercial Users of Functional Programming (CUFP)*. ACM SIGPLAN, 2010.

[14] A. Lakshman and P. Malik. Cassandra - a decentralized structured storage system. In *Workshop on Large-Scale Distributed Systems and Middleware (LADIS)*, 2009.

[15] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comp.*, C-28(9):690–691, 1979.

[16] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Technical Report 7506, INRIA, 2011.

[17] D. B. Terry, A. J. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. W. Welch. Session guarantees for weakly consistent replicated data. In *Parallel and Distributed Information Systems (PDIS)*, 1994.

[18] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Symposium on Operating Systems Principles (SOSP)*, 1995.