A Generic Specification Framework for Weakly Consistent Replicated Data Types

Xue Jiang, Hengfeng Wei, Yu Huang, *Member, IEEE,* Yuxing Chen, Anqun Pan

Abstract—Burckhardt et al. proposed a formal specification framework for eventually consistent replicated data types, denoted (vis, ar), based on the notions of visibility and arbitration relations. However, being specific to eventually consistent systems, this framework has two limitations. First, it does not cover non-convergent consistency models since arbitration ar is a total order over events. Second, it does not cover the consistency models in which each event is required to be aware of the return values of some events that are visible to it when justifying its return value. These limitations make the (vis, ar) framework not generic enough to specify and reason about important weak consistency models such as Causal Memory and PRAM. In this paper, we extend this framework to a more generic one called (vis, ar, V) for weakly consistent replicated data types. To specify non-convergent consistency models as well, we relax the arbitration relation ar to be a partial order. To overcome the second limitation, we allow to specify for each event e, a subset $V(e)$ of its visible set whose return values cannot be ignored when justifying the return value of e. To make it practically feasible, we provide candidates for the visibility and arbitration relations and the V function. By combining candidates for these three components, we are able to specify not only existing consistency models but also new ones that are reasonable and promising for practical usefulness. We then show how to specify consistency models in our framework, and provide three case studies.

Index Terms—Replicated Data Types, Consistency Models, Sequential Consistency, Causal Consistency, Pipelined Consistency

1 Introduction

Geographically distributed systems often replicate data at multiple sites to achieve high availability and low latency, even under network partitions [1], [2]. According to the CAP theorem [3], [4] and the PACELC tradeoff [5], these systems often sacrifice strong consistency and choose to implement *weakly consistent replicated data types*.

Eventual consistency is one of the most widely used weak consistency models in distributed systems [6], [7]. It guarantees that "if clients stop issuing updates, the replicas will eventually reach a consistent state." [7]. For example, to allow replicas to respond to user operations immediately, collaborative text editing systems [8], [9] usually implement an *eventually consistent* replicated list object modelling the shared document. It requires the final lists at all replicas to be identical after executing the same set of user operations [8]. In principle, eventual consistency has two aspects:

1) At the *strong* aspect, it requires eventual *convergence* among replicas and thus the clients[1] will eventually obtain the same view of replica states.
2) At the *weak* aspect, it imposes no restrictions on the intermediate states and thus the clients may obtain (temporarily) inconsistent views of replica states.

Burckhardt et al. [1], [2] proposed a formal specification framework for eventually consistent replicated data types. It is based on the *visibility* (denoted vis) and *arbitration* (denoted ar) relations over events of a history, and we call it the (vis, ar) framework.

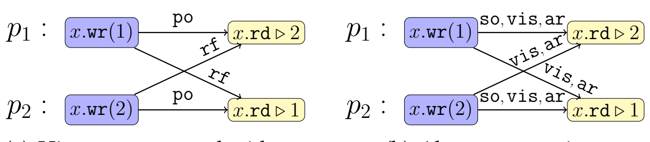
- The visibility relation is an acyclic relation that accounts for the relative timing of events. Intuitively, if an event e_1 is visible to another event e_2, it means that the effect of e_1 is visible to the client performing e_2 before e_2 is invoked. For example, e_2 may be a query returning the value written by update e_1 [10]. We call two events *concurrent* if they are invisible to each other.
- The arbitration relation is a *total order* over all events of a history. It indicates how the system resolves the conflicts due to concurrent events. For example, such a total order can be achieved using timestamps [11].

Being specific to eventually consistent systems, the (vis, ar) framework is not general enough to cover some important weak consistency models such as PRAM [12] and Causal Memory [13]. Specifically, we identify two limitations of the (vis, ar) framework as follows, corresponding to the two aspects of eventual consistency mentioned above:

1) *Convergence vs. Non-Convergence.* The total ordering requirement of ar over all events aims to ensure eventual *convergence* of the clients' views of replica states. Therefore, the (vis, ar) framework does not cover the consistency models that do not enforce convergence.
2) *Awareness vs. Unawareness.* In the (vis, ar) framework, the return value of an event e is justified by the set $vis^{-1}(e)$ of events visible to e (arranged in the order

- Xue Jiang was with the State Key Laboratory for Novel Software Technology, Nanjing University, China until June 2024, and is currently with School of Navigation Engineering, Zhejiang International Maritime College, China. The work was primarily completed before her graduation from Nanjing University.
E-mail: xuejiang1225@gmail.com
- Hengfeng Wei, and Yu Huang are with the State Key Laboratory for Novel Software Technology, Nanjing University, China.
E-mail: xuejiang1225@gmail.com,{hfwei, yuhuang}@nju.edu.cn
- Yuxing Chen and Anqun Pan are with Tencent Inc., China.
E-mail: axinggu@gmail.com, aaronpan@tencent.com

1. Following [2], we use clients, sessions, and processes interchangeably. We also use program order and session order interchangeably.



(a) History augmented with rf. (b) Abstract execution.

Fig. 1: Motivating history for convergence and non-convergence: It satisfies *CM*, but it is non-convergent. We denote the operation of writing value v to x by $x.\mathtt{wr}(v) \triangleright \perp$ (where $\perp$ indicates that it returns no value and is omitted in figures) and the operation of reading v from x by $x.\mathtt{rd} \triangleright v$. We write $x.\mathtt{rd} \triangleright \_$ to emphasize that the return value has been ignored.

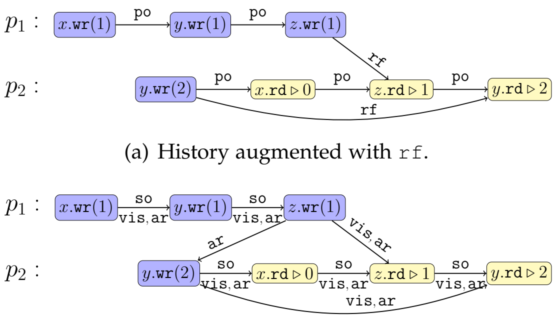
ar), while *ignoring* all their return values. Therefore, the (vis, ar) framework does not cover the consistency models in which an event is required to be *aware* of the return values of some events that are visible to it.

The feature of unawareness is useful for wait-free asynchronous message-passing distributed systems where operations should be completed without waiting for any other process. Unawareness allows such systems to execute operations in a speculative order with the possibility of rolling back and reordering previously executed operations if needed [14], [15], [16]. For example, Bayou [6] makes both local operations and remote operations received immediately visible, even if there are prior operations in the arbitration that may arrive in the future. When an operation o is received, all operations arbitrated after o but have already been applied are undone and then reapplied after applying o [16].

We illustrate both limitations of the (vis, ar) framework with Causal Memory (*CM*) [13], [14] with respect to read/write registers, and motivate our first contribution of this paper which is a generalization of the (vis, ar) framework. Intuitively, *CM* ensures that processes agree on the relative ordering of operations that are *causally related* [13]. The *causality order* over operations is defined as the transitive closure of the union of program order po and the read-from relation rf which informally associates each read with a unique write from which it reads the value. A history satisfies *CM* (w.r.t. read/write registers) if for each process, the set of all operations on this process and all write operations on other processes can be arranged into an operation sequence which preserves the causality order such that each read reads the value from the most recently preceding write in this sequence on the same register.

Example 1. Consider the history in Fig. 1(a) consisting of two processes p_1 and p_2 which read from and write to a shared register x. This history satisfies *CM*: the witness operation sequences for p_1 and p_2 are $\langle x.\mathtt{wr}(1)\ x.\mathtt{wr}(2)\ x.\mathtt{rd} \triangleright 2\rangle$ and $\langle x.\mathtt{wr}(2)\ x.\mathtt{wr}(1)\ x.\mathtt{rd} \triangleright 1\rangle$, respectively. However, it is non-convergent: processes p_1 and p_2 cannot agree with a total order ar over $x.\mathtt{wr}(1)$ and $x.\mathtt{wr}(2)$. Particularly, it does *not* satisfy the convergent variant of causal consistency called *WCCv* [2], [14] defined in the (vis, ar) framework; see Fig. 1(b) and Definition 17.

This example demonstrates that the (vis, ar) framework is insufficient for specifying histories that adhere to non-convergent consistency models such as *CM*. To specify non-convergent consistency models as well, we can relax the arbitration relation ar to be a partial order.



(a) History augmented with rf.

(b) Abstract execution.

Fig. 2: Motivating history for awareness and unawareness. It violates *CM*. The arrows for relations implied by transitivity are not drawn.

Example 2. Consider the history in Fig. 2(a). It does not satisfy *CM*. In any operation sequence for process p_2, $x.\mathtt{wr}(1)$ must be placed after $x.\mathtt{rd} \triangleright 0$ and $z.\mathtt{wr}(1)$ must be placed before $z.\mathtt{rd} \triangleright 1$. As a consequence, $y.\mathtt{wr}(1)$ must be placed between $y.\mathtt{wr}(2)$ and $y.\mathtt{rd} \triangleright 2$. However, such sequences cannot be valid; in particular, the return value of $y.\mathtt{rd} \triangleright 2$ is not justified.

However, if the *return values* of the operations that causally precede $y.\mathtt{rd} \triangleright 2$ can be ignored, $y.\mathtt{rd} \triangleright 2$ can be justified by the operation sequence $\langle x.\mathtt{wr}(1)\ y.\mathtt{wr}(1)\ z.\mathtt{wr}(1)\ y.\mathtt{wr}(2)\ x.\mathtt{rd} \triangleright \_\ z.\mathtt{rd} \triangleright \_\ y.\mathtt{rd} \triangleright 2\rangle$. Similarly, $z.\mathtt{rd} \triangleright 1$ and $x.\mathtt{rd} \triangleright 0$ (0 is the initial value of x) can be justified by $\langle x.\mathtt{wr}(1)\ y.\mathtt{wr}(1)\ z.\mathtt{wr}(1)\ y.\mathtt{wr}(2)\ x.\mathtt{rd} \triangleright \_\ z.\mathtt{rd} \triangleright 1\rangle$ and $\langle y.\mathtt{wr}(2)\ x.\mathtt{rd} \triangleright 0\rangle$, respectively. Actually, this history in Fig. 2(a) satisfies *WCCv* [2], [14] defined in the (vis, ar) framework in which return values are ignored; see Fig. 2(b) and Definition 17.

This example demonstrates that the (vis, ar) framework is insufficient for specifying histories that adhere to consistency models that demand awareness. To allow an event e to be aware of the return values of some or all events in $\mathtt{vis}^{-1}(e)$, we introduce a function V defined on events. Specifically, $V(e)$ for event e is a subset of $\mathtt{vis}^{-1}(e)$ whose return values cannot be ignored when justifying the return value of e.

Our First Contribution. In this paper, we extend the (vis, ar) specification framework for eventually consistent replicated data types into a generic one called (vis, ar, V) for weakly consistent replicated data types (Section 3). On the one hand, by relaxing ar to be a partial order, the (vis, ar, V) framework is able to cover non-convergent consistency models such as PRAM [12] and Causal Memory [13]. On the other hand, by introducing the V function for events, the (vis, ar, V) framework is able to cover the consistency models in which each event is required to be aware of the return values of some or all events that are visible to it. To make it practically feasible, we provide common candidates for the three components of this generic framework, including the vis and ar relations and the V function.

Our Second Contribution. By combining candidates for each component, we are able to specify various consistency models, including both existing ones and new ones, in the (vis, ar, V) framework, including variants of sequential consistency (Section 4), causal consistency (Section 5), and pipelined consistency (Section 7). In particular, we find that three variants of sequential consistency are equivalent. We also provide case studies to demonstrate the usefulness of our framework. Specifically, we show that the causal consistency protocol of MongoDB in the failure-free sharded cluster deployment satisfies Causal Memory Convergence, a new causal consistency variant discovered in our framework (Section 6). For pipelined consistency, we show that although the failure-free client-server implementation of GSP (Global Sequence Protocol) satisfies Weak Causal Convergence, it does not satisfy Pipelined Convergence, another pipelined consistency variant discovered in our framework (Section 8). We also show that RA-Linearizability, a specification recently proposed for conflict-free replicated data types, implies Weak Pipelined Convergence, but not Pipelined Convergence (Section 9).

This paper is an extended version of the SRDS conference paper [17], with four new Sections 4, 7, 8, and 9.

2 PRELIMINARIES

In this section, we review the formal specification framework called (vis, ar) for eventually consistent replicated data types proposed by Burckhardt et al. [1], [2].

2.1 Relations and Orderings

A binary relation R over a given set A is a subset of $A \times A$, i.e., $R \subseteq A \times A$. For $a, b \in A$, we use $(a, b) \in R$ and $a \xrightarrow{R} b$ interchangeably. The inverse relation of R is denoted by R^{-1}, i.e., $(a, b) \in R \iff (b, a) \in R^{-1}$. We use $R^{-1}(b)$ to denote the set $\{a \in A \mid (a, b) \in R\}$.

Given two binary relations R and S over set A, we define the composition of them as $R; S = \{(a, c) \mid \exists b \in A : a \xrightarrow{R} b \xrightarrow{S} c\}$. For $n \in \mathbb{N}^+$, R^n denotes the n-ary composition $R; R; \ldots; R$. The transitive closure of R is $R^+ \triangleq \bigcup_{n \geq 1} R^n$. For some subset $A' \subseteq A$, the restriction of R to A' is $R|_{A'} \triangleq R \cap (A' \times A)$. If $f : A \to B$ is a function (also a relation) from A to B, the restriction of f to $A' \subseteq A$ is $f|_{A'} \triangleq f \cap (A' \times B) = \{(a, f(a)) \mid a \in A'\}$.

A relation R is *natural* if $\forall x \in A : |R^{-1}(x)| < \infty$. A (strict) partial order is an irreflexive and transitive relation. A total order is a relation which is a partial order and total. For $A' \subseteq A$, $to(R, A')$ asserts that R is a total order over A'.

2.2 Abstract Data Types

We consider a replicated database storing *objects* of some abstract data types.

Definition 1. An *abstract data type* $\tau \in \mathit{Type}$ is a pair $\tau = (\mathit{Op}, \mathit{Val})$ such that

- Op is the set of *operations* supported by τ;
- Val is the set of values allowed by τ. We assume $\bot \in \mathit{Val}$ to indicate that some operations may return no value.

Definition 2. The *sequential semantics* of a type τ is a function $\mathtt{eval}_\tau : \mathit{Op}^* \times \mathit{Op} \to \mathit{Val}$ that, given a sequence of operations S and an operation o, determines the return value $\mathtt{eval}_\tau(S, o) \in \mathit{Val}$ for o when o is performed after S.

Examples 3, 4, and 5 contain formal definitions for the intuitive sequential semantics of read/write registers reg, key-value stores kvs, and queues fq, respectively.

Example 3. An integer read/write *register* reg supports two operations: $\mathtt{wr}(v)$ writes value $v \in \mathbb{Z}$ to the register and $\mathtt{rd}$ reads the value from it. A $\mathtt{rd}$ operation returns the value of the last preceding $\mathtt{wr}$, or the initial value 0 if there are no prior writes. Formally, for any operation sequence S,

$$\begin{aligned}&\mathtt{eval}_{\mathsf{reg}}(S, \mathtt{wr}(v)) = \bot,\\ &\quad \mathtt{eval}_{\mathsf{reg}}(S, \mathtt{rd}) = v, \text{ if } \mathtt{wr}(0)\ S = S_1\ \mathtt{wr}(v)\ S_2\\ &\qquad\qquad\qquad\qquad \text{and } S_2 \text{ contains no } \mathtt{wr} \text{ operations.}\end{aligned}$$

Example 4. A *key-value store* kvs supports two operations: $\mathrm{PUT}(k, v)$ writes value v to key k and $\mathrm{GET}(k)$ reads value (which may be the initial value 0) from key k. Formally, for any operation sequence S,

$$\begin{aligned}&\mathtt{eval}_{\mathsf{kvs}}(S, \mathrm{PUT}(k, v)) = \bot,\\ &\quad \mathtt{eval}_{\mathsf{kvs}}(S, \mathrm{GET}(k)) = v, \text{ if } \mathrm{PUT}(k, 0)\ S = S_1\ \mathrm{PUT}(k, v)\ S_2\\ &\qquad\qquad\qquad\qquad \text{and } S_2 \text{ contains no } \mathrm{PUT} \text{ operations on } k.\end{aligned}$$

Example 5. An integer *FIFO queue* fq supports two operations. $\mathtt{enq}(v)$ adds value $v \in \mathbb{N}$ to the tail of the queue. $\mathtt{deq}$ removes and returns the element v at the head of the queue; if the queue is empty, we let $v = \bot$. Formally, for any operation sequence S,

$$\mathtt{eval}_{\mathsf{fq}}(S, \mathtt{enq}(v)) = \bot,$$

$$\mathtt{eval}_{\mathsf{fq}}(S, \mathtt{deq}) = \begin{cases} \bot, & \text{if } S \text{ is empty} \\ v, & \text{if } S = S_1\ \mathtt{enq}(v)\ S_2 \\ & \quad \text{and } \mathtt{eval}_{\mathsf{fq}}(S_1, \mathtt{deq}) = \bot \\ & \quad \text{and } S_2 \text{ contains no } \mathtt{deq}. \end{cases}$$

2.3 Histories

Clients interact with the replicated database by performing operations on objects. We use a *history* to record such interactions in a computation.

Definition 3. A *history* is a tuple $H = (E, \mathtt{op}, \mathtt{rval}, \mathtt{so})$ such that

- E is the set of all *events* of operations invoked by clients in a single computation;
- $\mathtt{op} : E \to \mathit{Op}$ describes the operation of an event;
- $\mathtt{rval} : E \to \mathit{Val}$ describes the value returned by the operation $\mathtt{op}(e)$ of an event e;
- $\mathtt{so} \subseteq E \times E$ is a partial order over E, called the *session order*. It relates operations within a session in the order they were invoked by clients.

We lift $\mathtt{op}$ to sets of events by defining $\mathtt{op}(F) = \{\mathtt{op}(e) \mid e \in F\}$ for $F \subseteq E$. See Figs. 1(b) and 2(b) for examples of histories (for now ignore the relations $\mathtt{vis}$ and $\mathtt{ar}$).

2.4 Abstract Executions

To justify the return values of all events in a history, we need to know how these events are related to each other. Following [1], [2], this is captured declaratively by the *visibility* and *arbitration* relations.

Definition 4. An *abstract execution* is a triple $A = ((E, \mathtt{op}, \mathtt{rval}, \mathtt{so}), \mathtt{vis}, \mathtt{ar})$ such that

- $(E, \mathtt{op}, \mathtt{rval}, \mathtt{so})$ is a history;
- Visibility $\mathtt{vis} \subseteq E \times E$ is an acyclic and natural relation;
- Arbitration $\mathtt{ar} \subseteq E \times E$ is a total order.

Figs. 1(b) and 2(b) show examples of abstract executions. In both abstract executions, the visibility relations $\mathtt{vis}$ correspond to the read-from $\mathtt{rf}$ relations. The arbitration relation $z.\mathtt{wr}(1) \xrightarrow{\mathtt{ar}} y.\mathtt{wr}(2)$ in Fig. 2(b) enforces an order between these two concurrent operations.

2.5 Consistency Models

Definition 5. A *consistency model* is a set of consistency predicates on abstract executions.

A consistency predicate on an abstract execution A is a statement that is true or false depending on A [2]. We write $A \models P$ if the consistency predicate P is true on the abstract execution A.

Definition 6. An *abstract execution A satisfies* consistency model $\mathcal{C} = \{P_1, \ldots, P_n\}$, denoted $A \models \mathcal{C}$, if each consistency predicate in $\mathcal{C}$ is true on A.

We are concerned with histories that satisfy some consistency model.

Definition 7. A *history H satisfies* consistency model $\mathcal{C} = \{P_1, \ldots, P_n\}$, denoted $H \models \mathcal{C}$, if it can be extended to an abstract execution that satisfies $\mathcal{C}$. Formally, $H \models \mathcal{C} \iff \exists\, \mathtt{vis}, \mathtt{ar}.\ (H, \mathtt{vis}, \mathtt{ar}) \models \mathcal{C}$.

2.6 Return Value Consistency

A common consistency predicate is the consistency of return values defined for a given data type τ, denoted $\text{RVAL}(\tau)$. $\text{RVAL}(\tau)$ requires that the return value $\mathtt{rval}(e)$ of an event e in an abstract execution A should agree with the result computed by applying the sequential semantics of data type τ to the operation sequence based on $\mathtt{ctxt}_A(e)$, which is obtained by arranging the events in $\mathtt{vis}^{-1}(e)$ according to $\mathtt{ar}$, and the operation $\mathtt{op}(e)$ [2]. Here $\mathtt{ctxt}_A(e)$ is called the *operation context* of e in A. It is the restriction of A to the set $\mathtt{vis}^{-1}(e)$ of events visible to e.

Definition 8. The *operation context* of an event $e \in E$ in an abstract execution $A = ((E, \mathtt{op}, \mathtt{rval}, \mathtt{so}), \mathtt{vis}, \mathtt{ar})$ is defined as $\mathtt{ctxt}_A(e) \triangleq A|_{\mathtt{vis}^{-1}(e), \mathtt{op}, \mathtt{vis}, \mathtt{ar}}$.

Definition 9. For a data type τ, its *return value consistency* predicate on an abstract execution A is $\text{RVAL}(\tau) \triangleq \forall e \in E.\ \mathtt{rval}(e) = \mathtt{eval}_\tau\big(\mathtt{ctxt}_A(e), \mathtt{op}(e)\big)$.

Note that $\mathtt{ctxt}_A(e)$ ignores $\mathtt{rval}$ in the original history. Consequently, when justifying the return value of event e, it is *not* required for $\mathtt{eval}$ to be consistent with the return values of the events visible to e. As shown in Example 2, the abstract execution in Fig. 2(b) satisfies $\text{RVAL}(\mathsf{reg})$.

3 A Generic Specification Framework for Weakly Consistent Replicated Data Types

In this section, we present our generic (vis, ar, V) specification framework for weakly consistent replicated data types. It not only covers more existing consistency models than the (vis, ar) framework does, but also helps to discover new ones. We first define the (vis, ar, V) framework in a general way. Then, we provide candidates for the visibility and arbitration relations and the V function.

3.1 The (vis, ar, V) Specification Framework

The (vis, ar, V) framework generalizes (vis, ar) in two ways, overcoming the two limitations of (vis, ar):

1) *Convergence vs. Non-Convergence.* The arbitration relation $\mathtt{ar}$ in (vis, ar, V) is a partial order over events. Therefore, (vis, ar, V) is able to cover classic non-convergent consistency models such as PRAM [12] and Causal Memory [13].
2) *Awareness vs. Unawareness.* In (vis, ar, V), we allow to specify for each event e, a subset $V(e)$ of $\mathtt{vis}^{-1}(e)$ whose return values cannot be ignored when justifying the return value of e. Therefore, (vis, ar, V) is able to cover the consistency models in which each event is required to be aware of the return values of some or all events that are visible to it.

To cover non-convergent consistency models, we reformulate the definition of abstract executions by relaxing $\mathtt{ar}$ to be a partial order.

Definition 10. An *abstract execution* in the (vis, ar, V) framework is a triple $A = ((E, \mathtt{op}, \mathtt{rval}, \mathtt{so}), \mathtt{vis}, \mathtt{ar})$ such that

- $(E, \mathtt{op}, \mathtt{rval}, \mathtt{so})$ is a history;
- Visibility $\mathtt{vis} \subseteq E \times E$ is an acyclic and natural relation;
- Arbitration $\mathtt{ar} \subseteq E \times E$ is a partial order.

To be aware of return values is a bit more involved. We first reformulate operation context by selectively unhiding the return values of some or all visible events.

Definition 11. Let $A = ((E, \mathtt{op}, \mathtt{rval}, \mathtt{so}), \mathtt{vis}, \mathtt{ar})$ be an abstract execution in the (vis, ar, V) framework. The *operation context* of $e \in E$ in A is defined as

$$\mathtt{ctxt}_A(e, V) \triangleq A|_{\mathtt{vis}^{-1}(e), \mathtt{op}, \mathtt{rval}|_{V(e)}, \mathtt{vis}, \mathtt{ar}},$$

where $V : E \to 2^E$ specifies a subset of $\mathtt{vis}^{-1}(e)$. We let *Ctxt* be the set of all operation contexts, ranged over by $\mathbb{C}$. We use $V_{\mathbb{C}}$ to select the set $V(e)$ in $\mathbb{C}$.

Accordingly, the RVAL consistency predicate should be adapted to use $\mathtt{ctxt}_A(e, V)$.

Definition 12. For a data type τ, its *return value consistency* predicate on an abstract execution A is

$$\text{RVAL}(\tau, V) \triangleq \forall e \in E.\ \mathtt{rval}(e) \in \mathtt{eval}_\tau\big(\mathtt{ctxt}_A(e, V), \mathtt{op}(e)\big).$$

In the definition above, we need to redefine the sequential semantics of τ, i.e., the $\mathtt{eval}_\tau$ function. On the one hand, since $\mathtt{ar}$ is a partial order, there may be a set of operation sequences over $\mathtt{vis}^{-1}(e)$ to evaluate. Thus, we regard an operation context as a *set of serializations*, which are linear extensions of $\mathtt{ar}$ over $\mathtt{vis}^{-1}(e)$. This is why we use '$\in$'

TABLE 1: Recipes for $\mathtt{vis}$, $\mathtt{ar}$, and V in (vis, ar, V) Framework.

$\mathtt{vis}$	$\emptyset \subseteq \mathtt{vis}$	
	$\mathtt{so} \subseteq \mathtt{vis}$	(*Read Your Writes* [18])
	$\mathtt{vis};\mathtt{so} \subseteq \mathtt{vis}$	(*Monotonic Reads* [18])
	$\mathtt{so};\mathtt{vis} \subseteq \mathtt{vis}$	(*Monotonic Writes* [18])
	$\mathtt{so};\mathtt{vis};\mathtt{so} \subseteq \mathtt{vis}$	
	$\mathtt{vis};\mathtt{so};\mathtt{vis} \subseteq \mathtt{vis}$	(*Writes Follow Reads* [18])
	$(\mathtt{so} \cup \mathtt{vis})^+ \subseteq \mathtt{vis}$	
$\mathtt{ar}$	$\emptyset \subseteq \mathtt{ar}$	
	$\mathtt{so} \subseteq \mathtt{ar}$	
	$\mathtt{vis} \subseteq \mathtt{ar}$	
	$\mathtt{vis};\mathtt{so} \subseteq \mathtt{ar}$	
	$\mathit{to}(\mathtt{ar}, E)$	
$V(e)$	$V(e) = \emptyset$	
	$V(e) = \mathtt{so}^{-1}(e) \cap \mathtt{vis}^{-1}(e)$	
	$V(e) = \mathtt{vis}^{-1}(e)$	

instead of '$=$' in the definition of $\textsc{RVal}(\tau, V)$. On the other hand, besides justifying the return value $\mathtt{rval}(e)$ of event e, $\mathtt{eval}_\tau$ is required to preserve the unhidden return values specified by $\mathtt{rval}|_{V(e)}$ in a given serialization obtained from $\mathtt{ctxt}_A(e, V)$. Such serializations are considered *valid*.

Definition 13. The *sequential semantics* of a type $\tau \in \mathit{Type}$ is a function $\mathtt{eval}_\tau : \mathit{Ctxt} \times \mathit{Op} \to 2^{\mathit{Val}}$ that, given an operation context $\mathbb{C} \in \mathit{Ctxt}$ and an operation $o \in \mathit{Op}$, determines the possible return values $\mathtt{eval}_\tau(\mathbb{C}, o) \subseteq \mathit{Val}$ for o when o is performed in the context $\mathbb{C}$. Specifically, $\mathtt{eval}_\tau(\mathbb{C}, o)$ is computed as follows[2]:

$$\mathtt{eval}_\tau(\mathbb{C}, o) = \big\{v \in \mathit{Val} \mid \exists S \in \mathbb{C} : \\ \big(\mathtt{eval}_\tau(\mathtt{op}(S), o) = v \wedge \\ \forall e \in V_{\mathbb{C}} : \mathtt{rval}(e) = \mathtt{eval}_\tau\big(\mathtt{op}(S_{\prec e}), \mathtt{op}(e)\big)\big)\big\},$$

where $S_{\prec e}$ is the prefix of S before e. Given a serialization $S \in \mathbb{C}$, the first conjunction computes the return value for o when o is performed after operation sequence $\mathtt{op}(S)$, and the second one ensures that S is valid by checking the unhidden return values w.r.t. corresponding prefixes of S.

Example 6. We argue that the history in Fig. 1 satisfies *CM* in the (vis, ar, V) framework. (*CM* is formally defined in Section 5.) We choose an $\mathtt{ar}$ which does not arbitrate between $x.\mathtt{wr}(1)$ and $x.\mathtt{wr}(2)$. It is easy to check that the resulting abstract execution in Fig. 1(b) satisfies $\textsc{RVal}(\mathsf{reg})$.

Example 7. We argue that the history in Fig. 2 does not satisfy *CM* in the (vis, ar, V) framework. Consider the abstract execution in Fig. 2(b), obtained by augmenting the history with $\mathtt{vis}$ and $\mathtt{ar}$ as in Example 2. Besides $\mathtt{vis} \subseteq \mathtt{ar}$, *CM* requires $\mathtt{eval}_{\mathsf{reg}}$ to preserve the unhidden return values of the visible events in $V(e) = \mathtt{so}^{-1}(e)$. By similar argument to that in Example 2, there is no way of justifying $y.\mathtt{rd} \triangleright 2$ while preserving the return values of $x.\mathtt{rd} \triangleright 0$ and $z.\mathtt{rd} \triangleright 1$.

3.2 Recipes for the (vis, ar, V) Framework

The (vis, ar, V) framework is parametric w.r.t. three components, i.e., the vis and ar relations and the V function. By combining different consistency predicates on them, we can specify various consistency models in this framework. To make it practically feasible, we provide a number of candidates for each component, as summarized in Table 1.

2. Note that we overload $\mathtt{eval}_\tau$ with different type signatures.

3.2.1 Recipe for the Visibility Relation

We identify a number of common consistency predicates on $\mathtt{vis}$ in roughly the order they induce larger and larger visible sets consisting of the events visible to some event.

- The weakest one does not enforce an event to observe any particular set of events. Formally, $\emptyset \subseteq \mathtt{vis}$.
- The most basic ingredient for visibility is the session order $\mathtt{so}$. The predicate $\mathtt{so} \subseteq \mathtt{vis}$ requires each event to see all the previous events *in the same session*.
- To allow an event to see the events *in different sessions* as well, it is necessary to compose $\mathtt{so}$ with $\mathtt{vis}$ in some ways. There are four basic kinds of compositions, namely $\mathtt{vis};\mathtt{so} \subseteq \mathtt{vis}$, $\mathtt{so};\mathtt{vis} \subseteq \mathtt{vis}$, $\mathtt{so};\mathtt{vis};\mathtt{so} \subseteq \mathtt{vis}$, and $\mathtt{vis};\mathtt{so};\mathtt{vis} \subseteq \mathtt{vis}$.
- To further allow an event to see the events in different sessions through an arbitrarily long chain of compositions of $\mathtt{vis}$ and $\mathtt{so}$, we rely on the transitive closure over $\mathtt{vis}$ and $\mathtt{so}$. That is, we have $(\mathtt{so} \cup \mathtt{vis})^+ \subseteq \mathtt{vis}$, where $\mathtt{hb} \triangleq (\mathtt{so} \cup \mathtt{vis})^+$ is the well-known *happens-before order* [2], [1] first proposed by Lamport [11]. Note that $\mathtt{hb} \subseteq \mathtt{vis}$ implies that $\mathtt{vis}$ is transitive.

3.2.2 Recipe for the Arbitration Relation

When justifying the return value of event e with respect to the sequential semantics $\mathtt{eval}$, we rely on $\mathtt{ar}$ to resolve conflicts caused by concurrent events in the set $\mathtt{vis}^{-1}(e)$ of events visible to e. In the (vis, ar, V) framework, $\mathtt{ar}$ is a partial order. In the following, we identify a number of common consistency predicate on $\mathtt{ar}$ in roughly the order they are able to resolve more and more conflicts.

- The weakest one does not impose any constraints on how the conflicts should be resolved. Formally, $\emptyset \subseteq \mathtt{ar}$.
- A slightly stronger arbitration orders the events in $\mathtt{vis}^{-1}(e)$ (for some event e) according to the session order [1]. Formally, $\mathtt{so} \subseteq \mathtt{ar}$. Note that $\mathtt{so}$ may be a *proper* subset of $\mathtt{vis}$.
- To resolve all conflicts (using $\mathtt{ar}$) among visible events to an event, we require $\mathtt{vis} \subseteq \mathtt{ar}$.
- $(\mathtt{vis};\mathtt{so}) \subseteq \mathtt{ar}$ orders an event after other ones previously observed in the same session [1].
- Finally, convergent consistency models often require $\mathtt{ar}$ to be a total order over E, denoted $\mathit{to}(\mathtt{ar}, E)$, as in the (vis, ar) framework [1], [2].

3.2.3 Recipe for the V Function

By definition, $V(e)$ is a subset of $\mathtt{vis}^{-1}(e)$. We identify three common consistency predicates on $V(e)$:

- The strongest one requires e to be aware of the return values of *all* events visible to it. Formally, $V(e) = \mathtt{vis}^{-1}(e)$;
- Consistency models like Causal Memory [13] allow e to ignore the return values of its visible events *in different sessions* while being aware of those in its own session. Formally, $V(e) = \mathtt{so}^{-1}(e) \cap \mathtt{vis}^{-1}(e)$;
- The weakest one allows e to ignore the return values of any visible events, as in the (vis, ar) framework. Formally, $V(e) = \emptyset$.

TABLE 2: Consistency variants specified in the (vis, ar, V) specification framework. (New variants are marked with $[*]$.)

Consistency Models		Alternative Names	vis	ar	$V(e)$	Case Studies
Sequential Consistency Variants	*SC*	SequentialConsistency (Def. 21 [10])	$\mathtt{vis} = \mathtt{ar}$	$\mathit{to}(\mathtt{ar}, E)$	$V(e) = \emptyset$	—
					$V(e) = \mathtt{so}^{-1}(e)$	
					$V(e) = \mathtt{vis}^{-1}(e)$	
Causal Consistency Variants	*WCC* (Def. 8 [14])	CC (Def. 4.2 [15])	$\mathtt{hb} \subseteq \mathtt{vis}$	$\mathtt{vis} \subseteq \mathtt{ar}$	$V(e) = \emptyset$	—
	CM ([13], [15])	CC (Def. 9 [14])			$V(e) = \mathtt{so}^{-1}(e)$	—
	SCC	[*] (Section 5.4)			$V(e) = \mathtt{vis}^{-1}(e)$	—
	WCCv	CCv (Def. 4.5 [15], Def. 12 [14]) CausalConsistency (Section 5.2 [2]) Causality (Def. 26 [10])		$\mathtt{vis} \subseteq \mathtt{ar} \wedge \mathit{to}(\mathtt{ar}, E)$	$V(e) = \emptyset$	—
	CMv	[*] (Section 5.6)			$V(e) = \mathtt{so}^{-1}(e)$	MongoDB (Section 6) The causal consistency protocol of MongoDB satisfies *CMv*.
	SCCv	[*] (Section 5.7)			$V(e) = \mathtt{vis}^{-1}(e)$	—
Pipelined Consistency Variants	*WPC*	[*] (Section 7.2)	$\mathtt{so} \subseteq \mathtt{vis}$	$\mathtt{vis} \subseteq \mathtt{ar}$	$V(e) = \emptyset$	—
	PC	PRAM [12] (Def. 2.2 [19])			$V(e) = \mathtt{so}^{-1}(e)$	—
	SPC	[*] (Section 7.4)			$V(e) = \mathtt{vis}^{-1}(e)$	—
	WPCv	*SUC* (Def. 9 [20])		$\mathtt{vis} \subseteq \mathtt{ar} \wedge \mathit{to}(\mathtt{ar}, E)$	$V(e) = \emptyset$	RA-Linearizability (Section 9) RA-Linearizability implies *WPCv*, but not *PCv*.
	PCv	[*] (Section 7.6)			$V(e) = \mathtt{so}^{-1}(e)$	GSP (Section 8) The implementation of GSP satisfies *WCCv*, but not *PCv*.
	SPCv	[*] (Section 7.7)			$V(e) = \mathtt{vis}^{-1}(e)$	—

4 Sequential Consistency: Models

Sequential consistency [21] is considered a strong consistency model. It requires that all operations appear to have executed in some sequential order consistent with the session order. We consider three variants of sequential consistency, namely $\mathit{SC}_{\emptyset}$, $\mathit{SC}_{\mathtt{so}}$, and $\mathit{SC}_{\mathtt{vis}}$ (which is the original Sequential Consistency (*SC*) in [21], [2]). They require $\mathit{to}(\mathtt{ar}, E)$ to enforce a total order over all events and $\mathtt{vis} = \mathtt{ar}$ to capture that all events in $\mathtt{ar}^{-1}(e)$ are visible to event e. They differ in the function V. Specifically, $\mathit{SC}_{\emptyset}$, $\mathit{SC}_{\mathtt{so}}$, and *SC* require $V(e)$ to be $\emptyset$, $\mathtt{so}^{-1}(e)$, and $\mathtt{vis}^{-1}(e)$, respectively. Interestingly, we find that $\mathit{SC}_{\emptyset}$, $\mathit{SC}_{\mathtt{so}}$, and *SC* are equivalent.

Theorem 1. Given a history H, we have

$$H \models \mathit{SC}_{\emptyset} \iff H \models \mathit{SC}_{\mathtt{so}} \iff H \models \mathit{SC}.$$

Proof. The "$\Longleftarrow$" direction holds trivially. Now suppose that $H \models \mathit{SC}_{\emptyset}$. Therefore, there exists a visibility relation $\mathtt{vis}$ and an arbitration relation $\mathtt{ar}$ such that $\mathtt{vis} = \mathtt{ar}$ and the return value of each event e in H can be justified by the prefix of $\mathtt{ar}$, i.e., $\mathtt{ar}^{-1}(e) = \mathtt{vis}^{-1}(e)$, while ignoring their return values. We show that these $\mathtt{vis}$ and $\mathtt{ar}$ relations also satisfy the properties required by $\mathit{SC}_{\mathtt{so}}$ and *SC*. The proof proceeds by induction on the events in the total $\mathtt{ar}$ order.

- (Base Case) The first event, denoted e_1, in $\mathtt{ar}$ can be justified by $\mathtt{vis}$ and $\mathtt{ar}$ in $\mathit{SC}_{\mathtt{so}}$ and *SC*, since $\mathtt{so}^{-1}(e_1) = \mathtt{vis}^{-1}(e_1) = \emptyset$.
- (Induction Hypothesis) Assume that for each of the first $(k-1)$ events e in $\mathtt{ar}$, the return value of e can be justified by $\mathtt{ar}^{-1}(e)$, while respecting the return values of events in $\mathtt{so}^{-1}(e)$ (resp. $\mathtt{vis}^{-1}(e)$) as required by $\mathit{SC}_{\mathtt{so}}$ (resp. *SC*).
- (Inductive Step) Consider the k-th event in $\mathtt{ar}$, denoted e_k. On the one hand, since $H \models \mathit{SC}_{\emptyset}$, the return value of e_k can be justified by $\mathtt{ar}^{-1}(e_k)$. On the other hand, by induction hypothesis, the return values of the events in $\mathtt{so}^{-1}(e_k)$ (resp. $\mathtt{vis}^{-1}(e_k)$) can be justified using these $\mathtt{vis}$ and $\mathtt{ar}$ as required by $\mathit{SC}_{\mathtt{so}}$ (resp. *SC*).

□

5 Causal Consistency: Models

5.1 Overview

Causal consistency is one of the most widely used weak consistency models in distributed systems [13], [22]. The key notion is the *happens-before* order $\mathtt{hb}$ over events [11], [2]. Intuitively, causal consistency ensures that if an event e_1 happens before event e_2, then all sessions must observe e_1 before e_2. However, concurrent events may be observed in different orders by different sessions.

In the literature, there are several causal consistency variants with subtle differences [14], [2], [15]. In this section, we consider six variants that we call Weak Causal Consistency (*WCC*), Weak Causal Convergence (*WCCv*), Causal Memory (*CM*), Causal Memory Convergence (*CMv*), Strong Causal Consistency (*SCC*), and Strong Causal Convergence (*SCCv*), as defined in Table 2. Note that these variants may have different names in related work, as summarized in Table 2. Among these variants, *CMv*, *SCC*, and *SCCv* are new variants discovered in our framework.

In terms of visibility, all causal consistency variants require $\mathtt{hb} = (\mathtt{so} \cup \mathtt{vis})^{+} \subseteq \mathtt{vis}$ to capture the happens-before order over events. In terms of arbitration, they all require $\mathtt{vis} \subseteq \mathtt{ar}$ to enforce the happens-before order over events in $\mathtt{vis}^{-1}(e)$ when justifying the return value of the event e in its operation context. However, they may differ in two aspects: [3]

- How large is the function V for specifying the subset of visible events whose return values must be respected? Note that in causal consistency models, $\mathtt{so}^{-1}(e) \subseteq \mathtt{vis}^{-1}(e)$ for any event e (since $\mathtt{so} \subseteq \mathtt{vis}$). Thus, the candidate for the V function $V(e) = \mathtt{so}^{-1}(e) \cap \mathtt{vis}^{-1}(e)$ is equivalent to $V(e) = \mathtt{so}^{-1}(e)$.
- How strong is the arbitration relation $\mathtt{ar}$ for resolving conflicts? We distinguish between two cases according to whether $\mathtt{ar}$ is a total order or not, given $\mathtt{vis} \subseteq \mathtt{ar}$.

3. To exclude the trivial implementations in which replicas update objects locally without communicating with each other, we may additionally require the *eventual visibility* (*EV*) predicate [2], [14]. *EV* ensures that an event can be invisible to at most finitely many other events.

Fig. 3 shows the relative strength of these variants and Fig. 4 gives examples on objects of FIFO queues.

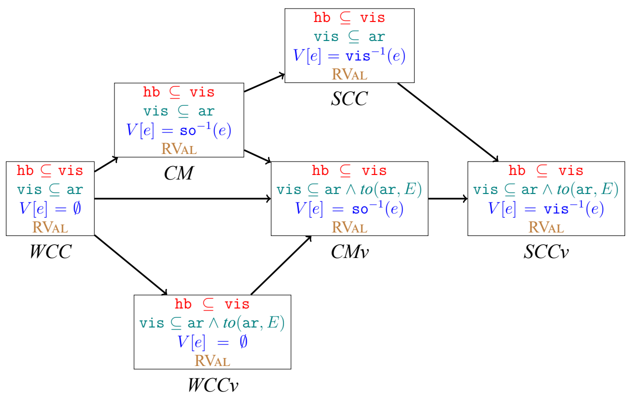


Fig. 3: Causal consistency variants and their relative strength. An arrow from a consistency model $\mathcal{C}_1$ to another one $\mathcal{C}_2$ denotes that $\mathcal{C}_2$ is stronger than $\mathcal{C}_1$.

5.2 Weak Causal Consistency

Weak causal consistency (*WCC*), recently proposed in [14], is the weakest variant of causal consistency we consider. Informally speaking, an abstract execution satisfies *WCC* as long as the return value $\mathtt{rval}(e)$ of each event e can be justified by some serialization of its visible set $\mathtt{vis}^{-1}(e)$, while ignoring their return values. More specifically, in terms of arbitration, *WCC* makes no extra restrictions on convergence and thus concurrent events may be observed in different orders by different sessions. In terms of $V(e)$, *WCC* allows each event e to ignore all the return values of its visible events.

Definition 14 (Weak Causal Consistency)**.**

$$\mathit{WCC} \triangleq (\mathtt{hb} \subseteq \mathtt{vis}) \land (\mathtt{vis} \subseteq \mathtt{ar}) \land (V(e) = \emptyset) \land \text{RVAL}.$$

Example 8. The history of Fig. 4(a) does not satisfy *WCC*. Intuitively, event $q.\mathtt{enq}(2)$ (resp. $p.\mathtt{enq}(2)$) must be visible to event $q.\mathtt{deq} \triangleright 2$ (resp. $p.\mathtt{deq} \triangleright 2$). By transitivity of $\mathtt{vis}$, event $p.\mathtt{enq}(1)$ is visible to event $p.\mathtt{enq}(2)$. Since $\mathtt{vis} \subseteq \mathtt{ar}$, it is impossible for the third session to dequeue 2 before 1 from the FIFO queue q.

The history of Fig. 4(b) satisfies *WCC*. For example, the return value of $b : \mathtt{deq}\triangleright 2$ can be justified by the serialization $\langle \mathtt{enq}(1)\ \mathtt{enq}(2)\ a : \mathtt{deq} \triangleright \_\ b : \mathtt{deq} \triangleright 2\rangle$.[4] Note that such a justification is not required by *WCC* to be consistent with the return value of $a : \mathtt{deq} \triangleright 2$. Similarly, the return value of $b : \mathtt{deq} \triangleright 1$ can be justified by the serialization $\langle \mathtt{enq}(2)\ \mathtt{enq}(1)\ a : \mathtt{deq} \triangleright \_\ b : \mathtt{deq} \triangleright 1\rangle$.

5.3 Causal Memory

Causal memory (*CM*) was originally defined by Ahamad et al. [13] on read/write registers. Recently, Perrin et al. [14] extended it to arbitrary replicated data types. *CM* is stronger than *WCC* in that when justifying the return value $\mathtt{rval}(e)$ of each event e, *CM* takes into account not only the *operation invocations* of the set $\mathtt{vis}^{-1}(e)$ of events visible to e as in *WCC* but also *the return values* of the set $\mathtt{so}^{-1}(e)$ of events that precede e in the same session. In other words, compared to *WCC*, *CM* requires that each session is consistent with respect to the previous return values provided [15].

4. For clarity, we include the event b to be justified in the serialization.

Definition 15 (Causal Memory)**.**

$$\mathit{CM} \triangleq (\mathtt{hb} \subseteq \mathtt{vis}) \land (\mathtt{vis} \subseteq \mathtt{ar}) \land (V(e) = \mathtt{so}^{-1}(e)) \land \text{RVAL}.$$

Example 9. Although the history of Fig. 4(b) satisfies *WCC*, it does not satisfy *CM*. Specifically, being aware of the return value of $a : \mathtt{deq} \triangleright 2$, $b : \mathtt{deq} \triangleright 2$ is unjustifiable. That is, it is impossible to construct a valid serialization consisting of $\mathtt{enq}(2)$, $a : \mathtt{deq} \triangleright 2$, $\mathtt{enq}(1)$, and $b : \mathtt{deq} \triangleright 2$ subject to $\mathtt{enq}(1) \xrightarrow{\mathtt{vis}} a : \mathtt{deq} \triangleright 2 \xrightarrow{\mathtt{vis}} b : \mathtt{deq} \triangleright 2$.

The history of Fig. 4(d) satisfies *CM*. For example, the return values of $a : p.\mathtt{deq} \triangleright 2$ and $b : p.\mathtt{deq} \triangleright 1$ can be justified by serializations $\langle q.\mathtt{enq}(2)\ p.\mathtt{enq}(1)\ a : p.\mathtt{deq} \triangleright 1\ p.\mathtt{enq}(2)\ a : p.\mathtt{deq} \triangleright 2\rangle$ and $\langle p.\mathtt{enq}(2)\ q.\mathtt{enq}(2)\ q.\mathtt{deq} \triangleright 2\ p.\mathtt{enq}(1)\ b : p.\mathtt{deq} \triangleright 2\ b : p.\mathtt{deq} \triangleright 1\rangle$, respectively.

5.4 Strong Causal Consistency

We strengthen *CM* to *SCC* (Strong Causal Consistency) by further requiring each session to be consistent with respect to the return values provided by other sessions. Formally, we have $V(e) = \mathtt{vis}^{-1}(e)$.

Definition 16 (Strong Causal Consistency)**.**

$$\mathit{SCC} \triangleq (\mathtt{hb} \subseteq \mathtt{vis}) \land (\mathtt{vis} \subseteq \mathtt{ar}) \land (V(e) = \mathtt{vis}^{-1}(e)) \land \text{RVAL}.$$

Example 10. The history of Fig. 4(f) satisfies *SCC*. The return values of $\mathtt{deq} \triangleright 1$, $b : \mathtt{deq} \triangleright 2$, $b : \mathtt{deq} \triangleright 3$, $a : \mathtt{deq} \triangleright 3$, and $a : \mathtt{deq} \triangleright 2$ can be justified by the serializations $\langle \mathtt{enq}(1)\ \mathtt{deq} \triangleright 1\rangle$, $\langle \mathtt{enq}(1)\ \mathtt{deq} \triangleright 1\ \mathtt{enq}(2)\ b : \mathtt{deq} \triangleright 2\rangle$, $\langle \mathtt{enq}(1)\ \mathtt{deq} \triangleright 1\ \mathtt{enq}(2)\ b : \mathtt{deq} \triangleright 2\ \mathtt{enq}(3)\ \mathtt{deq} \triangleright 3\rangle$, $\langle \mathtt{enq}(1)\ \mathtt{deq} \triangleright 1\ \mathtt{enq}(3)\ a : \mathtt{deq} \triangleright 3\rangle$ and $\langle \mathtt{enq}(1)\ \mathtt{deq} \triangleright 1\ \mathtt{enq}(3)\ \mathtt{enq}(2)\ a : \mathtt{deq} \triangleright 3\ a : \mathtt{deq} \triangleright 2\rangle$, respectively.

The history of Fig. 4(c) does not satisfy *SCC*. (For now, ignore $\mathtt{enq}(2) \xrightarrow{\mathtt{ar}} \mathtt{enq}(1)$ and $a \xrightarrow{\mathtt{ar}} b$ which are for *WCCv*.) Specifically, being aware of the return values of $a : \mathtt{deq} \triangleright 2$ and $b : \mathtt{deq} \triangleright 2$, $\mathtt{deq} \triangleright \bot$ is unjustifiable: it is impossible to construct a valid serialization consisting of all the events, since 2 is dequeued twice.

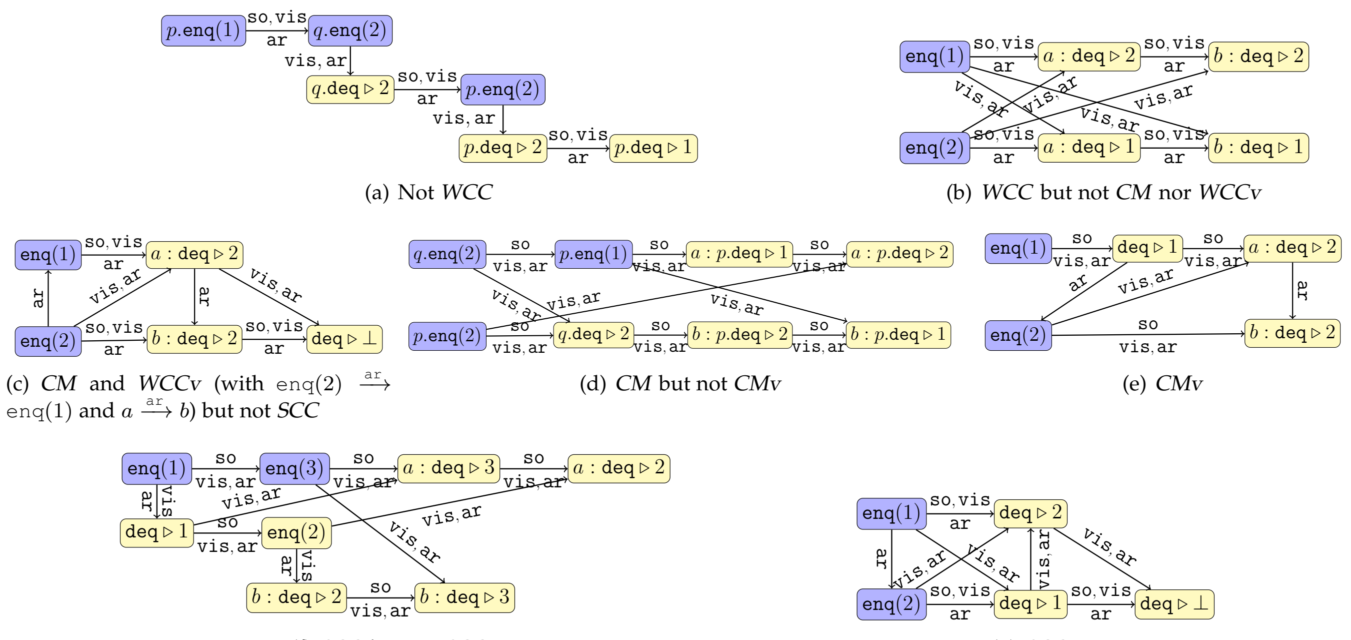
5.5 Weak Causal Convergence

WCCv (Weak Causal Convergence) [2], [14] is the convergent counterpart of *WCC*. It strengthens *WCC* by imposing a total order over all events in an execution, which provides all sessions with a uniform way of resolving conflicts caused by concurrent events. Consequently, the return value $\mathtt{rval}(e)$ of each event e is evaluated on the set $\mathtt{vis}^{-1}(e)$ of events visible to e, ordered by this common total order $\mathtt{ar}$, while ignoring all of their return values.

Definition 17 (Weak Causal Convergence)**.**

$$\begin{aligned}\mathit{WCCv} \triangleq\ & (\mathtt{hb} \subseteq \mathtt{vis}) \land (\mathtt{vis} \subseteq \mathtt{ar} \land \mathit{to}(\mathtt{ar}, E)) \\ & \land (V(e) = \emptyset) \land \text{RVAL}.\end{aligned}$$

Example 11. Although the history of Fig. 4(b) satisfies *WCC*, it does not satisfy *WCCv*. Specifically, the justification for the return value of $b : \mathtt{deq} \triangleright 2$ requires



(a) Not *WCC*

(b) *WCC* but not *CM* nor *WCCv*

(c) *CM* and *WCCv* (with $\mathtt{enq}(2) \xrightarrow{\mathtt{ar}} \mathtt{enq}(1)$ and $a \xrightarrow{\mathtt{ar}} b$) but not *SCC*

(d) *CM* but not *CMv*

(e) *CMv*

(f) *SCC* but not *SCCv*

(g) *SCCv*

Fig. 4: Examples for causal consistency variants on objects of FIFO queue fq. Both p and q are of type fq in 4(a) and 4(d). The queue q in other subfigures is implicitly assumed. In each history, events are grouped into sessions which are horizontally laid out. The arrows for relations implied by transitivity are not drawn. We use labels, such as a and b, to make events unique.

$\mathtt{enq}(1) \xrightarrow{\mathtt{ar}} \mathtt{enq}(2)$, while the one for $b : \mathtt{deq} \triangleright 1$ requires $\mathtt{enq}(2) \xrightarrow{\mathtt{ar}} \mathtt{enq}(1)$.

The history of Fig. 4(c) satisfies *WCCv*. The serialization $\langle \mathtt{enq}(2)\ \mathtt{enq}(1)\ a : \mathtt{deq} \triangleright 2\rangle$ for justifying $a : \mathtt{deq} \triangleright 2$, the one $\langle \mathtt{enq}(2)\ b : \mathtt{deq} \triangleright 2\rangle$ for $b : \mathtt{deq} \triangleright 2$, and the one $\langle \mathtt{enq}(2)\ \mathtt{enq}(1)\ a : \mathtt{deq} \triangleright \_\ b : \mathtt{deq} \triangleright \_\ \mathtt{deq} \triangleright \bot\rangle$ for $\mathtt{deq} \triangleright \bot$ agree with a common total order $\mathtt{ar}$, e.g., $\langle \mathtt{enq}(2)\ \mathtt{enq}(1)\ a : \mathtt{deq} \triangleright 2\ b : \mathtt{deq} \triangleright 2\ \mathtt{deq} \triangleright \bot\rangle$.

5.6 Causal Memory Convergence

CMv (Causal Memory Convergence) is the convergent counterpart of *CM*, which requires $\mathtt{ar}$ to be a total order.

Definition 18 (Causal Memory Convergence)**.**

$$\begin{aligned} \mathit{CMv} \triangleq\ & (\mathtt{hb} \subseteq \mathtt{vis}) \land (\mathtt{vis} \subseteq \mathtt{ar} \land \mathit{to}(\mathtt{ar}, E)) \\ & \land (V(e) = \mathtt{so}^{-1}(e)) \land \mathrm{RVAL}. \end{aligned}$$

Example 12. Although the history of Fig. 4(d) satisfies *CM*, it does not satisfy *CMv* which enforces a total order $\mathtt{ar}$ over all events. As shown in Example 9, the justification for the return value of $a : p.\mathtt{deq} \triangleright 2$ requires $p.\mathtt{enq}(1) \xrightarrow{\mathtt{ar}} p.\mathtt{enq}(2)$, while the justification for the return value of $b : p.\mathtt{deq} \triangleright 1$ requires $p.\mathtt{enq}(2) \xrightarrow{\mathtt{ar}} p.\mathtt{enq}(1)$.

The history of Fig. 4(e) satisfies *CMv*: the serialization $\langle \mathtt{enq}(1)\ \mathtt{deq} \triangleright 1\rangle$ for justifying the return value of $\mathtt{deq} \triangleright 1$, the one $\langle \mathtt{enq}(1)\ \mathtt{deq} \triangleright 1\ \mathtt{enq}(2)\ a : \mathtt{deq} \triangleright 2\rangle$ for $a : \mathtt{deq} \triangleright 2$, and the one $\langle \mathtt{enq}(2)\ b : \mathtt{deq} \triangleright 2\rangle$ for $b : \mathtt{deq} \triangleright 2$ agree with a common total order $\mathtt{ar}$, e.g., $\langle \mathtt{enq}(1)\ \mathtt{deq} \triangleright 1\ \mathtt{enq}(2)\ a : \mathtt{deq} \triangleright 2\ b : \mathtt{deq} \triangleright 2\rangle$.

5.7 Strong Causal Convergence

SCCv (Strong Causal Convergence) is the convergent counterpart of *SCC*, which further requires $\mathtt{ar}$ to be a total order.

Definition 19 (Strong Causal Convergence)**.**

$$\begin{aligned} \mathit{SCCv} \triangleq\ & (\mathtt{hb} \subseteq \mathtt{vis}) \land (\mathtt{vis} \subseteq \mathtt{ar} \land \mathit{to}(\mathtt{ar}, E)) \\ & \land (V(e) = \mathtt{vis}^{-1}(e)) \land \mathrm{RVAL}. \end{aligned}$$

Example 13. Although the history of Fig. 4(f) satisfies *SCC*, it does not satisfy *SCCv*. Specifically, the justification for the return value of $a : \mathtt{deq} \triangleright 2$ requires $\mathtt{enq}(3) \xrightarrow{\mathtt{ar}} \mathtt{enq}(2)$, while the one for $b : \mathtt{deq} \triangleright 3$ requires $\mathtt{enq}(2) \xrightarrow{\mathtt{ar}} \mathtt{enq}(3)$.

The history of Fig. 4(g) satisfies *SCCv*. For $\mathtt{deq} \triangleright \bot$ to return $\bot$, it must be aware of the event $\mathtt{deq} \triangleright 2$ and by transitivity all other events. It can be justified by the serialization $\langle \mathtt{enq}(1)\ \mathtt{enq}(2)\ \mathtt{deq} \triangleright 1\ \mathtt{deq} \triangleright 2\ \mathtt{deq} \triangleright \bot\rangle$. So, with $\mathtt{deq} \triangleright 1 \xrightarrow{\mathtt{vis}} \mathtt{deq} \triangleright 2$ and $\mathtt{enq}(1) \xrightarrow{\mathtt{ar}} \mathtt{enq}(2)$, $\mathtt{deq} \triangleright 1$ and $\mathtt{deq} \triangleright 2$ can be justified by the serializations $\langle \mathtt{enq}(1)\ \mathtt{enq}(2)\ \mathtt{deq} \triangleright 1\rangle$ and $\langle \mathtt{enq}(1)\ \mathtt{enq}(2)\ \mathtt{deq} \triangleright 1\ \mathtt{deq} \triangleright 2\rangle$, respectively.

6 Causal Consistency: Case Study on MongoDB

In this section, we prove that the causal consistency protocol of MongoDB [23] satisfies *CMv*. MongoDB is a distributed database supporting data replication and sharding [23]. It is *document*-oriented, where a document is an ordered set of key-value pairs. Thus, for simplicity, we model MongoDB as a typical key-value store, which provides $\mathrm{GET}(k)$ and $\mathrm{PUT}(k, v)$ operations to clients. We focus on the causal consistency protocol, called *MongoDB-CC*, of MongoDB in the *failure-free sharded cluster* deployment, where each shard

TABLE 3: Notations used in *MongoDB-CC*.

Notations	Description
ct_c	the greatest cluster time known by client c
ot_c	the last operation time at client c
ct_s	cluster time at server s
aot_s	the last applied operation time at server s
clock_s	current physical clock at server s
oplog_s	operation log at server s
$\textsc{Cluster}(s)$	the cluster that s belongs to
$\textsc{primary}(s)$	the primary node in $\textsc{Cluster}(s)$
$\textsc{IsPrimary}(s)$	whether s is the primary node in $\textsc{Cluster}(s)$
$\textsc{IsSecondary}(s)$	whether s is a secondary node in $\textsc{Cluster}(s)$
k, v	key, value
ct, ot, aot	cluster time, operation time, applied operation time
$oplog$	oplog

is replicated in a cluster consisting of a primary node and several secondary nodes. Only primary nodes can accept PUT operations from clients. Table 3 provides a summary of notations used in the protocol. We reference pseudocode lines using the format algorithm#:line#.

6.1 States

6.1.1 Logical Clocks and Cluster Time

MongoDB-CC uses hybrid logical clocks (HLC) [24]. An HLC is a pair $(sec, counter)$ of physical time (in seconds) and a counter to distinguish operations that occurred within the same second. HLCs are compared lexicographically.

The cluster time is the time value of a node's logical clock [23]. It ticks (Line 4:1) only when a PUT operation is applied in the primary node of a cluster (Line 2:13). Nodes maintain and distribute their greatest known cluster time via messages.

6.1.2 Client and Server States

Each client c keeps track of the greatest known cluster time ct_c. It also maintains the timestamp of its last operation, denoted ot_c, capturing the client's causal dependencies.

Each server s keeps track of the greatest known cluster time ct_s. It uses an append-only operation log oplog_s to record the operations, as well as their timestamps, applied at s. Additionally, it maintains in aot_s the timestamp of the last operation applied at s.

6.2 Protocol

Algorithms 1 and 2 show the core of *MongoDB-CC*, handling GET and PUT operations [5] at the client and server side, respectively. The pseudocode for replication and clock management are shown in Algorithms 3 and 4, respectively.

6.2.1 GET(k)

A client c sends a GET request, containing the key k, its greatest known cluster time ct_c, and its last operation time ot_c, to a server s which stores key k. The server s first updates its cluster time ct_s (Line 2:2). To guarantee causality, it then checks whether the causal dependencies specified by ot_c have been applied locally, by comparing its last applied

5. The GET and PUT operations are assumed to be executed atomically. The implementation issues about multi-thread concurrency and synchronization are left unspecified.

Algorithm 1 Client operations at client c.

```
1:  function GET($k$)
2:      $s \leftarrow$ a server storing key $k$
3:      $\langle v, ct, ot \rangle \leftarrow s.\text{GET-REQUEST}(k, \mathsf{ct}_c, \mathsf{ot}_c)$
4:      $\mathsf{ct}_c \leftarrow \max(\mathsf{ct}_c, ct)$
5:      $\mathsf{ot}_c \leftarrow ot$        ▷ $\mathtt{ts}(\text{GET}) \leftarrow (\mathsf{ot}_c, s)$, $\mathtt{dt}(\text{GET}) \leftarrow \mathsf{ot}_c$
6:      return $v$
7:  function PUT($k, v$)
8:      $s \leftarrow$ the primary node storing key $k$
9:      $\langle ct, ot \rangle \leftarrow s.\text{PUT-REQUEST}(k, v, \mathsf{ct}_c)$
10:     $\mathsf{ct}_c \leftarrow ct$
11:     $\mathsf{ot}_c \leftarrow ot$        ▷ $\mathtt{ts}(\text{PUT}) \leftarrow (\mathsf{ot}_c, s)$, $\mathtt{dt}(\text{PUT}) \leftarrow \mathsf{ot}_c$
12:     return ok
```

Algorithm 2 Server operations at server s.

```
    $\mathsf{store}_s[k] \leftarrow 0$ for each key $k$        ▷ Initialization
1:  function GET-REQUEST($k, ct, ot$)
2:      $\mathsf{ct}_s \leftarrow \max(\mathsf{ct}_s, ct)$
3:      if $\mathsf{aot}_s < ot$ then
4:          PRIMARY($s$).NOOP($\mathsf{ct}_s, ot$)        ▷ for liveness
5:          if ISSECONDARY($s$) then
6:              repeat
7:                  $s$.REPLICATE()
8:              until $\mathsf{aot}_s \geq ot$
9:      $v \leftarrow \mathsf{store}_s[k]$
10:     return $\langle v, \mathsf{ct}_s, \mathsf{aot}_s \rangle$
11: function PUT-REQUEST($k, v, ct$)
12:     $\mathsf{ct}_s \leftarrow \max(\mathsf{ct}_s, ct)$
13:     $\mathsf{ct}_s \leftarrow$ TICK()
14:     $\mathsf{aot}_s \leftarrow \mathsf{ct}_s$
15:     $\mathsf{store}_s[k] \leftarrow v$
16:     $\mathsf{oplog}_s \leftarrow \mathsf{oplog}_s \circ \langle k, v, \mathsf{aot}_s \rangle$
17:     return $\langle \mathsf{ct}_s, \mathsf{aot}_s \rangle$        ▷ $\mathsf{ct}_s = \mathsf{aot}_s$
```

operation time aot_s with $ot \leftarrow \mathsf{ot}_c$ (Line 2:3). If $\mathsf{aot}_s < ot$ and s is a secondary node (Line 2:5), the server keeps replicating oplog from its primary node until $\mathsf{aot}_s \geq ot$ (Line 2:8). To ensure liveness, we allow the primary node to catch up by keeping applying NO-OP (Line 4:6) in case $\mathsf{aot}_s < ot$ holds in the primary node (Line 2:4). (Note that if $\mathsf{aot}_s \geq ot$ and s is the primary node, Line 2:4 does nothing.) Once $\mathsf{aot}_s \geq ot$ holds, the server s retrieves the value v of key k in local store_s (Line 2:9). Finally, the value v, as well as ct_s and aot_s are returned to the client (Line 2:10). Upon receiving the reply, the client updates its ct_c and ot_c accordingly.

6.2.2 PUT(k, v)

A client sends a PUT request, containing the key k, the value v, and its greatest known cluster time ct_c, to the server s which is the primary node storing key k. The server s first updates its cluster time ct_s (Line 2:12). Then it ticks its ct_s (Line 2:13) and advances its last applied operation time aot_s to the new ct_s (Line 2:14). After being applied in local store_s (Line 2:15), the PUT operation, as well as its operation time aot_s, is appended to oplog_s (Line 2:16). Finally, both ct_s and aot_s are returned to the client (Line 2:17). Upon receiving the reply, the client updates its ct_c and ot_c accordingly.

6.2.3 *Replication*

In a cluster, each secondary node s periodically pulls the oplog entries with greater operation time than aot_s from the primary node (Line 3:3). The retrieved entries in $oplog$ are appended to local oplog_s (Line 3:8), and the operations in it are applied in local store_s in increasing order of their

Algorithm 3 Replication at server s.

```
1:  function REPLICATE()                                   ▷ Run periodically
2:      if ISSECONDARY($s$) then
3:          $\langle oplog, ct \rangle \leftarrow$ PRIMARY($s$).PULL-OPLOG($\mathsf{ct}_s, \mathsf{aot}_s$)
4:          $\mathsf{ct}_s \leftarrow \max(\mathsf{ct}_s, ct)$
5:          for $\langle k, v, ot \rangle \in oplog$ do                     ▷ in $ot$ order
6:              $\mathsf{store}_s[k] \leftarrow v$
7:              $\mathsf{aot}_s \leftarrow ot$
8:          $\mathsf{oplog}_s \leftarrow \mathsf{oplog}_s \circ oplog$
9:  function PULL-OPLOG($ct, aot$)
10:     $\mathsf{ct}_s \leftarrow \max(\mathsf{ct}_s, ct)$
11:     $oplog \leftarrow$ oplog entries after $aot$ in $\mathsf{oplog}_s$
12:     return $\langle oplog, \mathsf{ct}_s \rangle$
```

Algorithm 4 Clock management at server s.

```
1:  function TICK()
2:      if $\mathsf{ct}_s.sec \geq \mathsf{clock}_s$ then
3:          return $\langle \mathsf{ct}_s.sec, \mathsf{ct}_s.counter + 1 \rangle$
4:      else
5:          return $\langle \mathsf{clock}_s, 0 \rangle$
6:  function NOOP($ct, ot$)
7:      $\mathsf{ct}_s \leftarrow \max(\mathsf{ct}_s, ct)$
8:      while $\mathsf{aot}_s < ot$ do
9:          $\mathsf{ct}_s \leftarrow$ TICK()
10:         $\mathsf{aot}_s \leftarrow \mathsf{ct}_s$
11:         $\mathsf{oplog}_s \leftarrow \mathsf{oplog}_s \circ \langle$NO-OP$, \mathsf{aot}_s \rangle$
```

operation times (Line 3:5). In the end, aot_s refers to the operation time of the last entry in its current oplog_s (Line 3:7). Additionally, secondary nodes and the primary node also distribute and keep track of their greatest known cluster time during replication.

6.3 Correctness Proof

We prove that *MongoDB-CC* satisfies *CMv* (Definition 18) by showing that every history $H = (E, \mathtt{op}, \mathtt{rval}, \mathtt{so})$ of *MongoDB-CC* satisfies *CMv* with respect to kvs (Example 4). The key is to extract appropriate visibility and arbitration relations from H such that $(H, \mathtt{vis}, \mathtt{ar}) \models$ *CMv*.

In the following, we use G, P, P_k and G_k to denote the set of GET events, the set of PUT events, the set of PUT events on key k, and the set of GET events on key k, respectively. For a PUT(k, v) event e, we write $e \triangleright \langle k, v, ot \rangle$ to emphasize that it is associated with an operation time at Line 2:16. For a GET(k) event e, we write $e \triangleright \langle k, v, ot \rangle$ to denote that it retrieves the value v with operation time ot (Line 2:9).

6.3.1 Timestamps

We first define timestamp $\mathtt{ts}(e)$ for each event e as follows.

Definition 20 (Timestamps)**.** For an event e issued by client c, $\mathtt{ts}(e) = (\mathsf{ot}_c, s)$, where ot_c is the last operation time of client c at Line 1:5 for GET and Line 1:11 for PUT, and s is the identifier of the server processing e (Lines 1:2 and 1:8).

Timestamps are compared lexicographically (we assume a total order over the set of identifiers of servers). For notational convenience, we further define the dependency time $\mathtt{dt}(e) \triangleq \mathtt{ts}(e).\mathsf{ot}_c$ for each event e. Note that for a PUT event e, $\mathtt{dt}(e)$ is its operation time assigned at Line 2:16.

6.3.2 Visibility

The visibility relation $\mathtt{vis}$ is based on the following *read-from* relation $\mathtt{rf}$.

Definition 21 (Read-from Relation)**.** $(e, f) \in \mathtt{rf}$ if and only if $e = $ PUT$(k, v) \triangleright \langle k, v, ot \rangle$ and $f = $ GET$(k) \triangleright \langle k, v, ot \rangle$ for some key k.

Definition 22 (Visibility)**.** The visibility relation $\mathtt{vis}$ is the transitive closure of the union of session order $\mathtt{so}$ and read-from relation $\mathtt{rf}$. Formally, $\mathtt{vis} = (\mathtt{so} \cup \mathtt{rf})^+$.

By induction on the structure of $\mathtt{vis}$, we can show that $\mathtt{vis}$ is reflected in $\mathtt{dt}$. Formally,

Lemma 1. *Let e_1 and e_2 be two events of history H. We have*

$$e_1 \xrightarrow{\mathtt{vis}} e_2 \implies \mathtt{dt}(e_1) \leq \mathtt{dt}(e_2).$$

Furthermore,

$$e_1 \xrightarrow{\mathtt{vis}} e_2 \land e_2 \in P \implies \mathtt{dt}(e_1) < \mathtt{dt}(e_2).$$

Proof. By induction on the structure of $\mathtt{vis}$.

- CASE I: $e_1 \xrightarrow{\mathtt{so}_c} e_2$ for some client c.
  - ot_c is monotonically increasing. So $\mathtt{dt}(e_1) \leq \mathtt{dt}(e_2)$.
  - If e_2 is a PUT, ot_c is increased due to the ticking at Line 2:13. So $\mathtt{dt}(e_1) < \mathtt{dt}(e_2)$.
- CASE II: $e_1 \xrightarrow{\mathtt{rf}} e_2$.
  - According to the waiting condition at Line 2:3, $\mathsf{ot}_c \geq \mathtt{dt}(e_1)$ always holds when e_2 has been performed at Line 1:5. So $\mathtt{dt}(e_1) \leq \mathtt{dt}(e_2)$.
  - e_2 is a GET.
- CASE III: There is some event e' such that $e_1 \xrightarrow{\mathtt{vis}} e' \xrightarrow{\mathtt{vis}} e_2$.
  - By induction, we have $\mathtt{dt}(e_1) \leq \mathtt{dt}(e') \leq \mathtt{dt}(e_2)$.
  - By induction, if e_2 is a PUT, $\mathtt{dt}(e_1) \leq \mathtt{dt}(e') < \mathtt{dt}(e_2)$.

□

Theorem 2. *The visibility relation $\mathtt{vis}$ is a partial order.*

Proof. It suffices to prove that $\mathtt{vis}$ is acyclic (thus, irreflexive). Suppose for a contradiction that there is a cycle $C : e_1 \xrightarrow{\mathtt{vis}} e_2 \xrightarrow{\mathtt{vis}} \cdots \xrightarrow{\mathtt{vis}} o_i \xrightarrow{\mathtt{vis}} e_1$. By Lemma 1, all events in C are GET and they have the same $\mathtt{dt}$. Furthermore, all of them cannot occur at the same client, as this would imply a cycle in session order. Assume e and e' in C are on different clients. Since $e \xrightarrow{\mathtt{vis}} e'$ and both of them are GET, there must be some PUT e'' such that $e \xrightarrow{\mathtt{vis}} e'' \xrightarrow{\mathtt{vis}} e'$. By Lemma 1, $\mathtt{dt}(e) < \mathtt{dt}(e'') \leq \mathtt{dt}(e')$, implying $\mathtt{dt}(e) \neq \mathtt{dt}(e')$, a contradiction. □

Theorem 3. $\mathtt{hb} \subseteq \mathtt{vis}$*.*

Proof. By definitions of $\mathtt{hb}$ and $\mathtt{vis}$, we have

$$\mathtt{hb} \triangleq (\mathtt{so} \cup \mathtt{vis})^+ = \mathtt{vis}^+ = \mathtt{vis}.$$

Clearly, $\mathtt{hb} \subseteq \mathtt{vis}$. □

6.3.3 Arbitration

We construct the arbitration relation $\texttt{ar}$ which is a total order over all events as follows.

Definition 23 (Arbitration)**.** Order the PUT events in $\texttt{ar}$ by their timestamps (Definition 20). For each client c, we insert its GET events one by one in session order: each GET event g on client c is placed immediately after the later (in $\texttt{ar}$) of (1) the previous (in $\texttt{so}$) event of g on client c, if any; and (2) the PUT event p (in $\texttt{ar}$) such that $p \xrightarrow{\texttt{rf}} g$.

Theorem 4. $\texttt{vis} \subseteq \texttt{ar}$.

Proof. Let e_1 and e_2 be two events of history H. We need to show that if $e_1 \xrightarrow{\texttt{vis}} e_2$, then $e_1 \xrightarrow{\texttt{ar}} e_2$. By induction on the structure of $\texttt{vis}$.

- CASE I : $e_1 \xrightarrow{\texttt{so}} e_2$ on the same client. By Condition 1 of $\texttt{ar}$, $e_1 \xrightarrow{\texttt{ar}} e_2$.
- CASE II : $e_1 \xrightarrow{\texttt{rf}} e_2$. By Condition 2 of $\texttt{ar}$, $e_1 \xrightarrow{\texttt{ar}} e_2$.
- CASE III : There is some event e' such that $e_1 \xrightarrow{\texttt{vis}} e' \xrightarrow{\texttt{vis}} e_2$. By induction and the transitivity of $\texttt{vis}$ and $\texttt{ar}$, we have $e_1 \xrightarrow{\texttt{ar}} e' \xrightarrow{\texttt{ar}} e_2$.

□

6.3.4 Return Values

We need to prove that $(H, \texttt{vis}, \texttt{ar}) \models \text{RVAL}(\mathsf{kvs}, V)$, where $V(e) = \texttt{so}^{-1}(e)$ for each event e. The key is to show, for each GET event e, that

$$\forall e \in G : \texttt{rval}(e) \in \texttt{eval}_{\mathsf{kvs}}(\texttt{ctxt}_A(e, V), \texttt{op}(e)),$$

where $\texttt{ctxt}_A(e, V) = A|_{\texttt{vis}^{-1}(e),\texttt{op},\texttt{rval}|_{\texttt{so}^{-1}(e)},\texttt{vis},\texttt{ar}}$. Suppose e is issued by client c. Define $cP \triangleq \texttt{so}^{-1}(e) \cup (\texttt{vis}^{-1}(e) \cap P)$ and $S_e \triangleq \texttt{ar}|_{cP}$.

Theorem 5. S_e is a valid serialization of cP.

Proof. Let $S_{\preceq e} \triangleq S_e \circ \langle e \rangle$, where '$\circ$' denotes concatenation. We need to show that each GET(k) event in $S_{\preceq e}$ returns the value written by the most recently preceding PUT event on key k. Consider $g = \text{GET}(k)$ on client c in $S_{\preceq e}$. Suppose $p \xrightarrow{\texttt{rf}} g$ and p is on client i. We must show that no other PUT$(k, _)$ events place in between p and g in $S_{\preceq e}$. Suppose by contradiction that event $p' = \text{PUT}(k, \_)$ on client j does. We distinguish between $j = c$ and $j \neq c$.

- CASE I : $j = c$ (Fig. 5(a)). Since $p \xrightarrow{\texttt{ar}} p'$, $\texttt{ts}(p) < \texttt{ts}(p')$. Since p and p' are applied on the same primary node, $\texttt{dt}(p) < \texttt{dt}(p')$. Since $p' \xrightarrow{\texttt{ar}} g$ and $j = c$, $p' \xrightarrow{\texttt{so}} g$. So, when g is issued by client c, $\mathsf{ot}_c \geq \texttt{dt}(p') > \texttt{dt}(p)$ at Line 1:3. By Line 2:8, it is impossible for g to read from p at Line 2:9.
- CASE II : $j \neq c$. Since $p' \xrightarrow{\texttt{ar}} g$, there is some event o on client c such that $p' \xrightarrow{\texttt{ar}} o \xrightarrow{\texttt{ar}} g$. Let o be the first such event. We perform a case analysis according to whether o is a PUT or a GET.
  - CASE II-1 : $o \in P_{k'}$ (Fig. 5(b)). Since $p \xrightarrow{\texttt{ar}} p' \xrightarrow{\texttt{ar}} o$, $\texttt{ts}(p) < \texttt{ts}(p') < \texttt{ts}(o)$. Since both p and p' are applied on the same primary node, $\texttt{dt}(p) < \texttt{dt}(p') \leq \texttt{dt}(o)$. Since $o \xrightarrow{\texttt{so}} g$, when g is issued by client c, $\mathsf{ot}_c \geq \texttt{dt}(o) > \texttt{dt}(p)$ at Line 1:3. By Line 2:8, it is impossible for g to read from p at Line 2:9.
  - CASE II-2 : $o \in G_{k'}$. We consider two cases.
    * CASE II-2-A : There are no events between p' and o in $S_{\preceq e}$ (Fig. 5(c)). By construction of S_e, $k' = k$ and $p' \xrightarrow{\texttt{rf}} o$. Therefore, $\texttt{dt}(p) < \texttt{dt}(p') \leq \texttt{dt}(o)$. By a similar argument in CASE II-1, it is impossible for g to read from p.
    * CASE II-2-B : There is a set, denoted B, of events between p' and o in $S_{\preceq e}$ (Fig. 5(d)).By the choice of o, B contains no events on client c. Therefore, all events in B are PUT. Moreover, there exists some event $p'' \in B$ such that $p'' \xrightarrow{\texttt{rf}} o$; otherwise, o should be placed before p' in $\texttt{ar}$. Then, $\texttt{dt}(p) < \texttt{dt}(p') \leq \texttt{dt}(p'') \leq \texttt{dt}(o)$. By a similar argument in CASE II-1, it is impossible for g to read from p.

□

7 PIPELINED CONSISTENCY: MODELS

7.1 Overview

Pipelined consistency extends the PRAM [12] consistency for shared memory to support arbitrary replicated data types [20]. It requires the updates on a single session are observed by all sessions in the session order they were issued, whereas updates from different sessions may be observed in different orders by different sessions.

In analogous to causal consistency, Table 2 summarizes six variants of pipelined consistency, namely Weak Pipelined Consistency (*WPC*), Pipelined Consistency (*PC*), Strong Pipelined Consistency (*SPC*), Weak Pipelined Convergence (*WPCv*), Pipelined Convergence (*PCv*), and Strong Pipelined Convergence (*SPCv*). Fig. 6 shows the relative strength of these variants and also relates them to their counterparts of causal consistency. These variants of pipelined consistency are weaker than their counterparts of causal consistency, since in the visibility relation they enforce only the session order (i.e., $\texttt{so} \subseteq \texttt{vis}$), instead of the happens-before order among events (i.e., $\texttt{hb} \subseteq \texttt{vis}$). Fig. 7 gives examples of pipelined consistency on objects of FIFO queues. We explain them in the following subsections.

7.2 Weak Pipelined Consistency

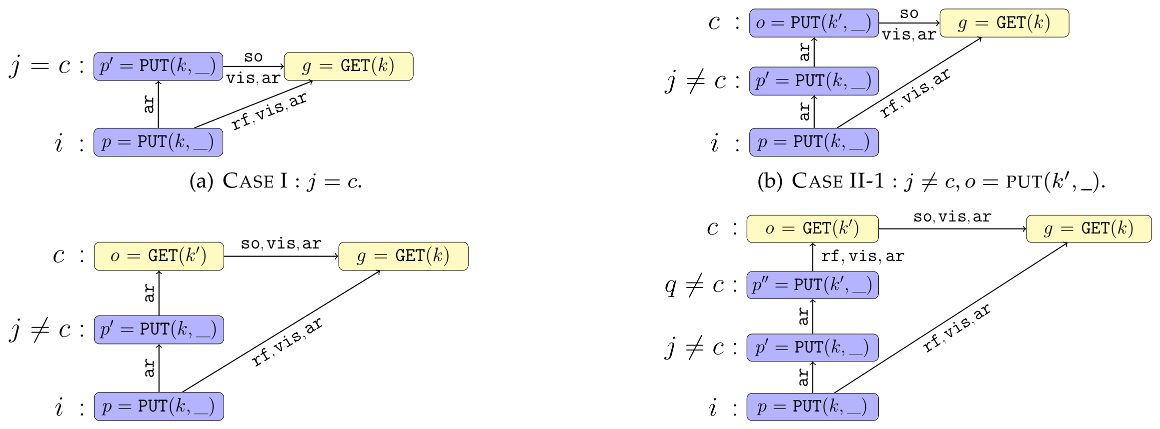
With $V(e) = \emptyset$, *WPC* allows each event e to ignore all the return values of its visible events.

Definition 24 (Weak Pipelined Consistency)**.**

$$\mathit{WPC} \triangleq (\texttt{so} \subseteq \texttt{vis}) \wedge (\texttt{vis} \subseteq \texttt{ar}) \wedge (V(e) = \emptyset) \wedge \text{RVAL}.$$

Example 14. The history of Fig. 7(a) does not satisfy *WPC*. Intuitively, events $\texttt{enq}(1)$, $\texttt{enq}(3)$, $a : \texttt{deq} \triangleright 1$, and $\texttt{deq} \triangleright 3$ must be visible to $b : \texttt{deq} \triangleright 1$. Since $\texttt{vis} \subseteq \texttt{ar}$, it is impossible for $b : \texttt{deq} \triangleright 1$ to dequeue 1 from the FIFO queue q.

The history of Fig. 7(b) satisfies *WPC*. The return values of $a : \texttt{deq} \triangleright 1$, $b : \texttt{deq} \triangleright 1$, and $\texttt{deq} \triangleright 3$ can be justified by the serializations $\langle \texttt{enq}(1)\ \texttt{enq}(3)\ a : \texttt{deq} \triangleright 1 \rangle$, $\langle \texttt{enq}(3)\ \texttt{enq}(1)\ a : \texttt{deq} \triangleright \_\ b : \texttt{deq} \triangleright 1 \rangle$, and $\langle \texttt{enq}(1)\ \texttt{enq}(2)\ \texttt{enq}(3)\ a : \texttt{deq} \triangleright \_\ b : \texttt{deq} \triangleright \_\ \texttt{deq} \triangleright 3 \rangle$, respectively.



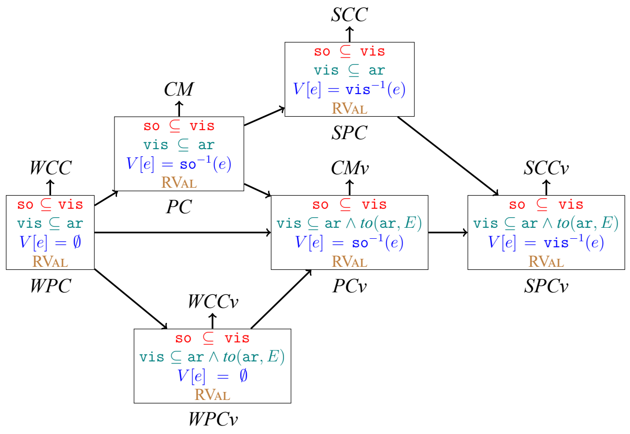
(a) CASE I : $j = c$.

(b) CASE II-1 : $j \neq c, o = \text{PUT}(k', \_)$.

(c) CASE II-2-A : $j \neq c, o = \text{GET}(k')$. There are no events between p' and o in $S_{\preceq e}$.

(d) CASE II-2-B : $j \neq c, o = \text{GET}(k')$. There is a set of events between p' and o in $S_{\preceq e}$.

Fig. 5: Illustration of the proof of Theorems 5.



SCC
SPC
CM
PC
WCC
WPC
CMv
PCv
SCCv
SPCv
WCCv
WPCv
$\mathtt{so} \subsetneq \mathtt{vis}$
$\mathtt{vis} \subsetneq \mathtt{ar}$
$V[e] = \mathtt{vis}^{-1}(e)$
$V[e] = \mathtt{so}^{-1}(e)$
$V[e] = \emptyset$
$\mathtt{vis} \subsetneq \mathtt{ar} \wedge to(\mathtt{ar}, E)$
RVAL

Fig. 6: Pipelined consistency variants and their relative strength.

7.3 Pipelined Consistency

With $V(e) = \mathtt{so}^{-1}(e)$, *PC* requires that each session is consistent with respect to the previous return values provided.

Definition 25 (Pipelined Consistency)**.**

$$PC \triangleq (\mathtt{so} \subseteq \mathtt{vis}) \wedge (\mathtt{vis} \subseteq \mathtt{ar}) \wedge (V(e) = \mathtt{so}^{-1}(e)) \wedge \text{RVAL}.$$

Example 15. Although the history of Fig. 7(b) satisfies *WPC*, it does not satisfy *PC*. Specifically, being aware of the return value of $a : \mathtt{deq} \triangleright 1, b : \mathtt{deq} \triangleright 1$ is unjustifiable: it is impossible to construct a valid serialization consisting of events $\mathtt{enq}(1)$, $\mathtt{enq}(3)$, $a : \mathtt{deq} \triangleright 1$, and $b : \mathtt{deq} \triangleright 1$, since 1 is dequeued twice.

The history of Fig. 7(d) satisfies *PC*. For example, the return values of $a : \mathtt{deq} \triangleright 2$ and $b : \mathtt{deq} \triangleright 1$ can be justified by the serializations $\langle \mathtt{enq}(1)\ \mathtt{enq}(2)\ a : \mathtt{deq} \triangleright 1\ a : \mathtt{deq} \triangleright 2 \rangle$ and $\langle \mathtt{enq}(2)\ \mathtt{enq}(1)\ b : \mathtt{deq} \triangleright 2\ b : \mathtt{deq} \triangleright 1 \rangle$, respectively.

7.4 Strong Pipelined Consistency

With $V(e) = \mathtt{vis}^{-1}(e)$, *SPC* further requires each session to be also consistent with respect to the return values provided by other sessions.

Definition 26 (Strong Pipelined Consistency)**.**

$$\begin{aligned} SPC \triangleq\ & (\mathtt{so} \subseteq \mathtt{vis}) \wedge (\mathtt{vis} \subseteq \mathtt{ar}) \\ & \wedge (V(e) = \mathtt{vis}^{-1}(e)) \wedge \text{RVAL}. \end{aligned}$$

Example 16. The history of Fig. 7(c) does not satisfy *SPC*. (For now, ignore $\mathtt{enq}(1) \xrightarrow{\mathtt{ar}} \mathtt{enq}(2)$ and $b : \mathtt{deq} \triangleright 1 \xrightarrow{\mathtt{ar}} a : \mathtt{deq} \triangleright 1$ which are for *WPCv*.) Specifically, being aware of the return values of $a : \mathtt{deq} \triangleright 1$ and $b : \mathtt{deq} \triangleright 1$, $\mathtt{deq} \triangleright \perp$ is unjustifiable: it is impossible to construct a valid serialization consisting of all the events, since 1 is dequeued twice.

The history of Fig. 7(f) satisfies *SPC*. The return values of $p.\mathtt{deq} \triangleright \perp$, $q.\mathtt{deq} \triangleright 2$, $q.\mathtt{deq} \triangleright 1$, and $p.\mathtt{deq} \triangleright 3$ can be justified by the serializations $\langle q.\mathtt{enq}(1)\ \ p.\mathtt{deq} \triangleright \perp \rangle$, $\langle p.\mathtt{enq}(1)\ q.\mathtt{enq}(2)\ q.\mathtt{deq} \triangleright 2 \rangle$, $\langle p.\mathtt{enq}(1)\ q.\mathtt{enq}(2)\ q.\mathtt{deq} \triangleright 2\ \ q.\mathtt{enq}(1)\ \ q.\mathtt{deq} \triangleright 1 \rangle$, and $\langle q.\mathtt{enq}(1)\ \ p.\mathtt{deq} \triangleright \perp\ p.\mathtt{enq}(3)\ p.\mathtt{enq}(1)\ q.\mathtt{enq}(2)\ p.\mathtt{deq} \triangleright 3 \rangle$, respectively.

7.5 Weak Pipelined Convergence

With $\mathtt{ar}$ being a total order, *WPCv* is the convergent counterpart of *WPC*.

Definition 27 (Weak Pipelined Convergence)**.**

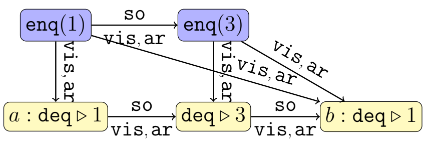
$$\begin{aligned} WPCv \triangleq\ & (\mathtt{so} \subseteq \mathtt{vis}) \wedge (\mathtt{vis} \subseteq \mathtt{ar} \wedge to(\mathtt{ar}, E)) \\ & \wedge (V(e) = \emptyset) \wedge \text{RVAL}. \end{aligned}$$

Example 17. Although the history of Fig. 7(b) satisfy *WPC*, it does not satisfy *WPCv*. Specifically, the justification for the return value of $a : \mathtt{deq} \triangleright 1$ requires $\mathtt{enq}(1) \xrightarrow{\mathtt{ar}} \mathtt{enq}(3)$, while the one of $b : \mathtt{deq} \triangleright 1$ requires $\mathtt{enq}(3) \xrightarrow{\mathtt{ar}} \mathtt{enq}(1)$.

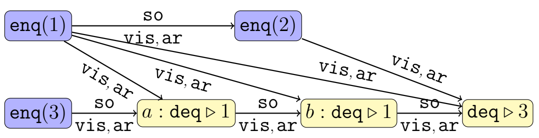
The history of Fig. 7(c) satisfies *WPCv*. The serialization $\langle \mathtt{enq}(1)\ \mathtt{enq}(2)\ b : \mathtt{deq} \triangleright 1 \rangle$ for justifying $b : \mathtt{deq} \triangleright 1$, the one $\langle \mathtt{enq}(1)\ a : \mathtt{deq} \triangleright 1 \rangle$ for justifying $a : \mathtt{deq} \triangleright 1$ and the one $\langle \mathtt{enq}(1)\ \mathtt{enq}(2)\ b : \mathtt{deq} \triangleright \_\ a : \mathtt{deq} \triangleright \_\ \mathtt{deq} \triangleright \perp \rangle$ for justifying $\mathtt{deq} \triangleright \perp$ agree with a common total order $\mathtt{ar}$, e.g., $\langle \mathtt{enq}(1)\ \mathtt{enq}(2)\ b : \mathtt{deq} \triangleright 1\ a : \mathtt{deq} \triangleright 1\ \mathtt{deq} \triangleright \perp \rangle$.

7.6 Pipelined Convergence

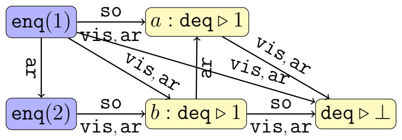
With $\mathtt{ar}$ being a total order, *PCv* is the convergent counterpart of *PC*.



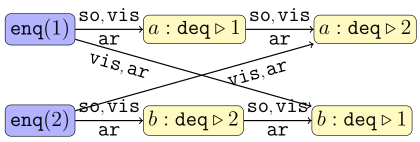
(a) not *WPC*



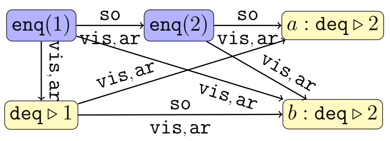
(b) *WPC* but not *PC* nor *WPCv*



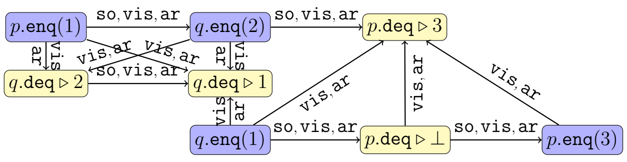
(c) *PC* and *WPCv* (with $\mathtt{enq}(1) \xrightarrow{\mathtt{ar}} \mathtt{enq}(2)$ and $b \xrightarrow{\mathtt{ar}} a$) but not *SPC*



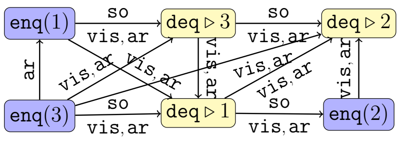
(d) *PC* but not *PCv*



(e) *PCv*



(f) *SPC* but not *SPCv*



(g) *SPCv*

Fig. 7: Examples for pipelined consistency variants on objects of FIFO queue fq. Both p and q are of type fq in 7(f). The queue q in other subfigures is implicitly assumed.

Definition 28 (Pipelined Convergence)**.**

$$\begin{aligned} PCv \triangleq\ & (\mathtt{so} \subseteq \mathtt{vis}) \land (\mathtt{vis} \subseteq \mathtt{ar} \land \mathit{to}(\mathtt{ar}, E)) \\ & \land (V(e) = \mathtt{so}^{-1}(e)) \land \text{RVAL}. \end{aligned}$$

Example 18. Although the history of Fig. 7(d) satisfies *PC*, it does not satisfy *PCv* which enforces a total order ar over all events. As shown in Example 15, the justification for the return value of $a : \mathtt{deq} \triangleright 2$ requires $\mathtt{enq}(1) \xrightarrow{\mathtt{ar}} \mathtt{enq}(2)$, while the justification for the return value of $b : \mathtt{deq} \triangleright 1$ requires $\mathtt{enq}(2) \xrightarrow{\mathtt{ar}} \mathtt{enq}(1)$.

The history of Fig. 7(e) satisfies *PCv*. The serialization $\langle \mathtt{enq}(1)\ \mathtt{deq} \triangleright 1\rangle$ for justifying $\mathtt{deq} \triangleright 1$, the one $\langle \mathtt{enq}(1)\ \mathtt{enq}(2)\ \mathtt{deq} \triangleright \_\ \ a : \mathtt{deq} \triangleright 2\rangle$ for justifying $a : \mathtt{deq} \triangleright 2$, and the one $\langle \mathtt{enq}(1)\ \mathtt{enq}(2)\ \mathtt{deq} \triangleright 1\ b : \mathtt{deq} \triangleright 2\rangle$ for justifying $b : \mathtt{deq} \triangleright 2$ agree with a total order, e.g., $\langle \mathtt{enq}(1)\ \mathtt{enq}(2)\ \mathtt{deq} \triangleright 1\ a : \mathtt{deq} \triangleright 2\ b : \mathtt{deq} \triangleright 2\rangle$.

7.7 Strong Pipelined Convergence

With ar being a total order, *SPCv* is the convergent counterpart of *SPC*.

Definition 29 (Strong Pipelined Convergence)**.**

$$\begin{aligned} SPCv \triangleq\ & (\mathtt{so} \subseteq \mathtt{vis}) \land (\mathtt{vis} \subseteq \mathtt{ar} \land \mathit{to}(\mathtt{ar}, E)) \\ & \land (V(e) = \mathtt{vis}^{-1}(e)) \land \text{RVAL}. \end{aligned}$$

Example 19. Although the history of Fig. 7(f) satisfies *SPC*, it does not satisfy *SPCv*. Specifically, the justification for the return value of $q.\mathtt{deq} \triangleright 1$ requires $q.\mathtt{enq}(2) \xrightarrow{\mathtt{ar}} q.\mathtt{enq}(1)$, while the one for $p.\mathtt{deq} \triangleright 3$ requires $q.\mathtt{enq}(1) \xrightarrow{\mathtt{ar}} q.\mathtt{enq}(2)$.

The history of Fig. 7(g) satisfies *SPCv*. For $\mathtt{deq} \triangleright 2$ to return 2, it must be aware of the event $\mathtt{deq} \triangleright 1$. It can be justified by the serialization $\langle \mathtt{enq}(3)\ \mathtt{enq}(1)\ \mathtt{deq} \triangleright 3\ \mathtt{deq} \triangleright 1\ \mathtt{enq}(2)\ \mathtt{deq} \triangleright 2\rangle$. So, with $\mathtt{enq}(3) \xrightarrow{\mathtt{ar}} \mathtt{enq}(1)$ and $\mathtt{deq} \triangleright 3 \xrightarrow{\mathtt{vis}} \mathtt{deq} \triangleright 1$, $\mathtt{deq} \triangleright 3$ and $\mathtt{deq} \triangleright 1$ can be justified by serializations $\langle \mathtt{enq}(3)\ \mathtt{enq}(1)\ \mathtt{deq} \triangleright 3\rangle$ and $\langle \mathtt{enq}(3)\ \mathtt{enq}(1)\ \mathtt{deq} \triangleright 3\ \mathtt{deq} \triangleright 1\rangle$, respectively.

TABLE 4: Notations used in GSP.

Notations	Description
known_c	prefix of total update sequence known by client c
$\mathsf{pending}_c$	local pending updates at client c
round_c	round number at client c
sequence	operation sequence at the server
updates	operations that have not been replicated
k, v, r	key, value, round
c, cl	client
log	operation log

8 Pipelined Consistency: Case Study on GSP

In this section, we show that although the *failure-free client-server* implementation of GSP (Global Sequence Protocol) [25] satisfies *WCCv* [2, Section 10.4.2], it does not satisfy *PCv*. GSP requires clients eventually agree on a global sequence of all updates, while seeing a subsequence at any time. To this end, the server uses *Reliable Total Order Broadcast* (RTOB) [26] to reliably deliver messages to all clients in the same total order. We model the system as a key-value store, which supports GET(k) and PUT(k, v) operations. Table 4 provides a summary of notations used in the protocol.

8.1 States

The server s maintains a sequence sequence of updates received from all clients and a set updates of updates that wait to be acknowledged and broadcast.

Each client c maintains two operation sequences, called known_c and $\mathsf{pending}_c$, that stores the currently known prefix of the global update sequence sequence and the local pending updates that have not been acknowledged by the server, respectively.

Algorithm 5 Client operations at client c.

```
1: function GET($k$)
2:     $log \leftarrow \mathsf{known}_c \circ \mathsf{pending}_c$
3:     $\langle k, v, r, cl\rangle \leftarrow$ UPDATES($k, log$)
4:     return $v$
5: function PUT($k, v$)
6:     $\mathsf{round}_c \leftarrow \mathsf{round}_c + 1$
7:     $\mathsf{pending}_c \leftarrow \mathsf{pending}_c \circ \langle k, v, \mathsf{round}_c, c\rangle$
8:     send PUT-REQUEST($k, v, \mathsf{round}_c, c$) to server $s$
9: function REPLICATE($k, v, r, cl$)
10:    $\mathsf{known}_c \leftarrow \mathsf{known}_c \circ \langle k, v, r, cl\rangle$
11:    if $cl = c$ then
12:        $\mathsf{pending}_c \leftarrow \mathsf{pending}_c[1..]$
```

Algorithm 6 Server operations at server s.

```
1: function PUT-REQUEST($k, v, r, c$)
2:     $\mathsf{sequence} \leftarrow \mathsf{sequence} \circ \langle k, v, r, c\rangle$
3:     $\mathsf{updates} \leftarrow \mathsf{updates} \cup \langle k, v, r, c\rangle$
4: function PROPAGATE-UPDATES()                ▷ Run periodically
5:     for $\langle k, v, r, c\rangle \in \mathsf{updates}$ do
6:         send REPLICATE($k, v, r, c$) to all clients by RTOB
7:     $\mathsf{updates} \leftarrow \emptyset$
```

8.2 Protocol

To issue a GET operation, a client c combines the updates in known_c and $\mathsf{pending}_c$ and determines the return value using an auxiliary function UPDATES. To issue a PUT operation, client c increments its local round number round_c, appends the update to $\mathsf{pending}_c$, and sends it to the server.

When the server s receives a PUT-REQUEST from client c, it appends the PUT operation to sequence, and adds it into updates. The server periodically replicates the updates in updates to all clients. When a client c receives an update, it appends it to known_c. If this update is originated from the client c, it is removed from $\mathsf{pending}_c$ since it has been acknowledged by the sever.

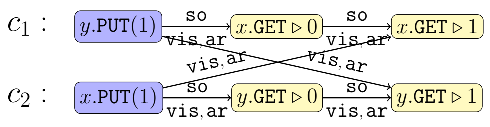


Fig. 8: A history for a kvs that can be produced by the implementation of GSP but does not satisfy *PCv*.

8.3 The Implementation of GSP does not Satisfy *PCv*

Consider the history of Figure 8. It consists of two clients c_1 and c_2 which issue GET and PUT operations on keys x and y. The implementation of GSP may produce this history as follows. The sequence at the server is $\langle y.\text{PUT}(1)\ x.\text{PUT}(1)\rangle$. When $x.\text{GET} \rhd 0$ and $x.\text{GET} \rhd 1$ are issued on client c_1, the values of known_{c_1} are $\langle y.\text{PUT}(1)\rangle$ and $\langle y.\text{PUT}(1)\ x.\text{PUT}(1)\rangle$, respectively, and both values of $\mathsf{pending}_{c_1}$ are empty. When $y.\text{GET} \rhd 0$ and $y.\text{GET} \rhd 1$ are issued on client c_2, the values of known_{c_2} are empty and $\langle y.\text{PUT}(1)\rangle$, respectively, and both values of $\mathsf{pending}_{c_2}$ are $\langle x.\text{PUT}(1)\rangle$.

The history does not satisfies *PCv*. Specifically, being aware of the return value of $x.\text{GET} \rhd 0$, the justification for the return value of $x.\text{GET} \rhd 1$ requires $y.\text{PUT}(1) \xrightarrow{\text{so,ar}} x.\text{GET} \rhd 0 \xrightarrow{\text{ar}} x.\text{PUT}(1)$. However, being aware of the return value of $y.\text{GET} \rhd 0$, the justification for the return value of $y.\text{GET} \rhd 1$ requires $x.\text{PUT}(1) \xrightarrow{\text{so,ar}} y.\text{GET} \rhd 0 \xrightarrow{\text{ar}} y.\text{PUT}(1)$.

9 Pipelined Consistency: Case Study on RA-Linearizability

Replication-Aware linearizability (RA-Linearizability), recently proposed by Wang et al. [27], is a consistency model tailored to Conflict-Free Replicated Data Types (CRDTs [28], [29]). RA-Linearizability relaxes linearizability [30] in two ways: (1) the linearization is required to be consistent with visibility relation instead of the real-time *returns-before* relation; (2) query events are allowed to see a subsequence, rather than a prefix, of the linearization. In this section, we show that RA-Linearizability implies *WPCv*, but not *PCv*.

Definition 30 (RA-Linearizability)**.** Let $h = (L, vis)$ be a history of a CRDT[6], where L is a set of `Queries` and `Updates` events[7] and vis is a partial order over L. The history h satisfies RA-Linearizability w.r.t. a sequential specification `Spec`, if there exists a sequence `seq` such that

- `seq` is a total order over L;
- $\mathtt{so} \subseteq vis$ (we use sessions to denote the replicas);
- $vis \cup \mathtt{seq}$ is acyclic; and
- for each query l_q in L, the subsequence of updates visible to l_q together with l_q, i.e., $\mathtt{seq}|_{vis^{-1}(l_q)\,\cap\,\mathtt{Updates}}$, is admitted by `Spec`.

9.1 RA-Linearizability Implies *WPCv*

The definition of RA-Linearizability is "isomorphic" to that of *WPCv*, in terms of the following three definitions.

Definition 31 (Visibility)**.** $\mathtt{vis} \triangleq vis$.

Definition 32 (Arbitration)**.** $\mathtt{ar} \triangleq \mathtt{seq}$.

Definition 33. $\mathtt{eval}_{\text{CRDT}} \triangleq \mathtt{Spec}$.

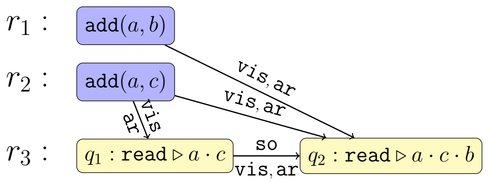


Fig. 9: A history for a list object l (implicitly assumed) that satisfies RA-Linearizability but not *PCv*.

9.2 RA-Linearizability does not Imply *PCv*

Consider the history of Figure 9. It consists of three replicas r_1, r_2, and r_3 which perform operations on a list l of characters. The list supports two operations[8]: (1) add(a, b) which inserts the character b immediately after the character a in the list, and (2) read which returns the list content. Assume that the list l initially contains only the character a.

6. RA-Linearizability is compositional under some assumptions; see [27, Section 5].
7. We do not consider query-update events and query-update rewriting; see [27, Definition 3.6].
8. We do not consider the *remove* operation in this example.

The history satisfies RA-Linearizability. The total sequence $\texttt{seq}$ can be $\langle \texttt{add}(a,b)\ \texttt{add}(a,c)\ q_1 : \texttt{read} \triangleright a \cdot c\ q_2 : \texttt{read} \triangleright a \cdot c \cdot b \rangle$. With $\texttt{add}(a,c) \xrightarrow{\texttt{vis}} q_1$, $\texttt{add}(a,b) \xrightarrow{\texttt{vis}} q_2$, and $\texttt{add}(a,c) \xrightarrow{\texttt{vis}} q_2$, the return values of q_1 and q_2 can be justified by the subsequences $\langle \texttt{add}(a,c) \rangle$ and $\langle \texttt{add}(a,b)\ \texttt{add}(a,c) \rangle$, respectively.

However, the history does *not* satisfy *PCv*. Being aware of the return value of q_1, q_2 is unjustifiable: q_1 requires $\texttt{add}(a,b)$ be ordered after both $\texttt{add}(a,c)$ and q_1, but q_2 requires $\texttt{add}(a,b)$ be ordered before $\texttt{add}(a,c)$.

10 Related Work

The (vis, ar) specification framework for arbitrary eventually consistent replicated data types is proposed by Burckhardt et al. [1], [2]. It introduces the visibility and arbitrary relations, which have been widely adopted in the literature. Using this framework, Viotti et al. [10] provide a comprehensive overview of more than 50 different consistency models in distributed systems. Emmi et al. [31] develop a fine-grained consistency specification methodology for software API via visibility relaxation. Recently, Almeida et al. [16] proposed a framework for consistency models in distributed systems using the traditional per-process serializations augmented with visibility. In this paper, we extend the (vis, ar) framework into a more generic one called (vis, ar, V) for weakly consistent replicated data types. Our framework is able to cover not only existing consistency models but also new ones that are reasonable and promising for practical usefulness.

Several works develop uniform frameworks for consistency models in *shared-memory* multiprocessor systems. Steinke et. al [19] present a unified theory of shared-memory consistency models based on four consistency properties. Enumerating all combinations of these four properties produces a lattice of consistency models. Alglave [32] provides a generic framework for weak consistency models in modern multiprocessor architectures. It uses the global time model and addresses the store atomicity relaxation.

Sequential consistency [21] is considered a strong consistency model. It is used in coordination services, such as ZooKeeper [33]. Interestingly, in our (vis, ar, V) framework, three variants of sequential consistency with different V functions turn out to be equivalent.

Causal consistency has been widely used in distributed systems [13], [22], [34]. There are three known causal consistency variants in the literature, namely *WCC* [14], [15], *CM* [13], [14], [15], and *WCCv* [2], [14], [15]. By following the recipes for our framework, we discover three new causal consistency variants, namely *SCC*, *CMv*, and *SCCv*. We show that the causal consistency protocol of MongoDB [23] satisfies *CMv*. As far as we know, this is the first correctness proof for MongoDB protocols against formal specifications.

Pipelined consistency [20] extends the PRAM consistency [12] in shared-memory multiprocessor systems to support arbitrary replicated data types in distributed systems. *WPCv* is a convergent variant of pipelined consistency [20]. We discover four new pipelined consistency variants, namely *WPC*, *SPC*, *PCv*, and *SPCv*. We show that the implementation of GSP [25] does not satisfy *PCv*, and that the RA-Linearizability [27] specification implies *WPCv* but not *PCv*.

11 Conclusion and Future Work

We extend the (vis, ar) specification framework for eventually consistent replicated data types into a more generic one called (vis, ar, V) for weakly consistent replicated data types. It covers both non-convergent consistency models and the consistency models in which each event is required to be aware of the return values of some or all events that are visible to it. We also provide case studies on MongoDB, the GSP protocol, and the RA-Linearizability specification to demonstrate the usefulness of our framework. We are implementing our specification framework in Alloy [35].

12 Acknowledgments

The authors would like to thank the anonymous reviewers for their insightful comments and suggestions. This work was partially supported by the Natural Science Foundation of Jiangsu Province (BK20242014) and the CCF-Tencent Open Fund (Tencent RAGR20200201). Hengfeng Wei is the corresponding author.

References

[1] S. Burckhardt, A. Gotsman, H. Yang, and M. Zawirski, “Replicated data types: Specification, verification, optimality,” in *Proceedings of the 41st ACM Symposium on Principles of Programming Languages (POPL’2014)*, pp. 271–284.

[2] S. Burckhardt, “Principles of eventual consistency,” *Found. Trends Program. Lang.*, vol. 1, no. 1-2, pp. 1–150, Oct. 2014.

[3] S. Gilbert and N. Lynch, “Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services,” *SIGACT News*, vol. 33, no. 2, pp. 51–59, Jun. 2002.

[4] E. A. Brewer, “Towards robust distributed systems (abstract),” in *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing (PODC’2000)*, pp. 7–7.

[5] D. Abadi, “Consistency tradeoffs in modern distributed database system design: CAP is only part of the story,” *IEEE Computer*, vol. 45, no. 2, pp. 37–42, Feb. 2012.

[6] 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 *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP’1995)*, pp. 172–182.

[7] W. Vogels, “Eventually consistent,” *Commun. ACM*, vol. 52, no. 1, pp. 40–44, Jan. 2009.

[8] C. A. Ellis and S. J. Gibbs, “Concurrency control in groupware systems,” in *Proceedings of the 1989 International Conference on Management of Data (SIGMOD’1989)*, pp. 399–407.

[9] H. Attiya, S. Burckhardt, A. Gotsman, A. Morrison, H. Yang, and M. Zawirski, “Specification and complexity of collaborative text editing,” in *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing (PODC’2016)*, pp. 259–268.

[10] P. Viotti and M. Vukolić, “Consistency in non-transactional distributed storage systems,” *ACM Comput. Surv.*, vol. 49, no. 1, pp. 19:1–19:34, Jun. 2016.

[11] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” *Commun. ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.

[12] R. J. Lipton and J. S. Sandberg, *PRAM: A scalable shared memory*. Princeton University, Department of Computer Science, 1988.

[13] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto, “Causal memory: Definitions, implementation, and programming,” *Distributed Computing*, vol. 9, no. 1, pp. 37–49, 1995.

[14] M. Perrin, A. Mostefaoui, and C. Jard, “Causal consistency: Beyond memory,” in *Proceedings of the 21st ACM Symposium on Principles and Practice of Parallel Programming (PPoPP’2016)*.

[15] A. Bouajjani, C. Enea, R. Guerraoui, and J. Hamza, "On verifying causal consistency," in *Proceedings of the 44th ACM Symposium on Principles of Programming Languages (POPL'2017)*, pp. 626–638.
[16] P. Sérgio Almeida, "A framework for consistency models in distributed systems," *arXiv e-prints*, pp. arXiv–2411, 2024.
[17] X. Jiang, H. Wei, and Y. Huang, "A generic specification framework for weakly consistent replicated data types," in *Proceedings of the 2020 International Symposium on Reliable Distributed Systems (SRDS'2020)*, 2020, pp. 143–154.
[18] K. Sivaramakrishnan, G. Kaki, and S. Jagannathan, "Declarative programming over eventually consistent data stores," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'2015)*. ACM, 2015, p. 413–424.
[19] R. C. Steinke and G. J. Nutt, "A unified theory of shared memory consistency," *J. ACM*, vol. 51, no. 5, pp. 800–849, Sep. 2004.
[20] M. Perrin, A. Mostefaoui, and C. Jard, "Update consistency for wait-free concurrent objects," in *2015 IEEE International Parallel and Distributed Processing Symposium (IPDPS'2015)*, pp. 219–228.
[21] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Transactions on Computers c-28*, vol. 9, pp. 690–691, 1979.
[22] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Don't settle for eventual: Scalable causal consistency for wide-area storage with cops," in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'2011)*, pp. 401–416.
[23] M. Tyulenev, A. Schwerin, A. Kamsky, R. Tan, A. Cabral, and J. Mulrow, "Implementation of cluster-wide logical clock and causal consistency in MongoDB," in *Proceedings of the 2019 International Conference on Management of Data*, pp. 636–650.
[24] S. S. Kulkarni, M. Demirbas, D. Madappa, B. Avva, and M. Leone, "Logical physical clocks," in *International Conference on Principles of Distributed Systems*, 2014, pp. 17–32.
[25] S. Burckhardt, D. Leijen, J. Protzenko, and M. Fähndrich, "Global sequence protocol: A robust abstraction for replicated shared state," in *Proceedings of the 29th European Conference on Object-Oriented Programming (ECOOP'2015)*, pp. 568–590.
[26] X. Défago, A. Schiper, and P. Urbán, "Total order broadcast and multicast algorithms: Taxonomy and survey," *ACM Computing Surveys (CSUR)*, vol. 36, no. 4, pp. 372–421, 2004.
[27] C. Wang, C. Enea, S. O. Mutluergil, and G. Petri, "Replication-aware linearizability," in *Proceedings of the 40th ACM Conference on Programming Language Design and Implementation (PLDI'2019)*, pp. 980–993.
[28] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in *Symposium on Self-Stabilizing Systems*, 2011, pp. 386–400.
[29] P. S. Almeida, "Approaches to conflict-free replicated data types," *ACM Comput. Surv.*, vol. 57, no. 2, pp. 51:1–51:36, Nov. 2024.
[30] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, Jul 1990.
[31] M. Emmi and C. Enea, "Weak-consistency specification via visibility relaxation," *Proc. ACM Program. Lang.*, vol. 3, Jan. 2019.
[32] J. Alglave, "A shared memory poetics," Ph.D. dissertation, L'université Paris Denis Diderot, 2010.
[33] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems," in *Proceedings of the 2010 USENIX Annual Technical Conference*, 2010.
[34] D. D. Akkoorath, A. Z. Tomsic, M. Bravo, Z. Li, T. Crain, A. Bieniusa, N. Preguiça, and M. Shapiro, "Cure: Strong semantics meets high availability and low latency," in *Proceedings of the 36th International Conference on Distributed Computing Systems (ICDCS'2016)*, pp. 405–414.
[35] D. Jackson, "Alloy: A lightweight object modelling notation," *ACM Trans. Softw. Eng. Methodol.*, vol. 11, no. 2, pp. 256–290, Apr 2002.

Xue Jiang received the Ph.D. degree in computer science from Nanjing University in 2024. She is currently a researcher with School of Navigation Engineering at Zhejiang International Maritime College. Her research interests include distributed computing and formal methods.

Hengfeng Wei received the Ph.D. degree in computer science from Nanjing University in 2016. He is currently a research assistant with Software Institute at Nanjing University. His research interests include distributed computing (especially the distributed data consistency problems) and formal methods.

Yu Huang received the Ph.D. degree in computer science from the University of Science and Technology of China in 2007. He is currently a professor with the Department of Computer Science and Technology at Nanjing University, China. His research interests include distributed algorithms and formal methods.

Yuxing Chen received the Ph.D. degree in computer science from the University of Helsinki, Finland. He is now a senior research engineer in database R&D department at Tencent, China. His research interests include database performance and evaluation, HTAP database design, and distributed system design.

Anqun Pan is the technical director of database R&D department at Tencent, China. He has more than 15 years' experience in the research and development of distributed computing and storage systems. He is currently responsible for the research and development of distributed database system TDSQL.