Comparing Causal Convergence Consistency Models

Sidi Mohamed Beillahi[1(✉)], Ahmed Bouajjani[2], and Constantin Enea[3]

[1] University of Toronto, Toronto, Canada
sm.beillahi@utoronto.ca
[2] Université Paris Cité, IRIF, CNRS, Paris, France
abou@irif.fr
[3] LIX, Ecole Polytechnique, CNRS and Institut Polytechnique de Paris, Paris, France
cenea@lix.polytechnique.fr

Abstract. In distributed databases, the CAP theorem establishes that a distributed storage system can only ensure two out of three properties: strong data consistency (i.e., reads returning the most recent writes), availability, or partition tolerance. Modern distributed storage systems prioritize performance and availability over strong consistency and thus offer weaker consistency models such as causal consistency.

This paper explores several variations of causal consistency (CC) that guarantee state convergence among replicas, meaning that all distributed replicas converge towards the same consistent state. We investigate a log-based CC model, a commutativity-based CC model, and a global sequence protocol-based CC model. To facilitate our study of the relationships between these models, we use a common formalism to define them. We then show that the log-based CC model is the weakest, while the commutativity-based CC and the global sequence protocol-based CC models are incomparable. We also provide sufficient conditions for a given application program to be robust against one CC model versus another, meaning that the program has the same behavior when executed over databases implementing the two CC models.

1 Introduction

In distributed databases, the CAP theorem establishes that a distributed storage system can only ensure two out of three properties: strong data consistency (i.e., reads returning the most recent writes), availability, or partition tolerance [14]. Modern distributed storage systems prioritize performance and availability over strong consistency and thus offer weaker consistency models such as causal consistency [20]. Causal consistency is a fundamental weak consistency model implemented in several production databases, e.g., CockroachDB and MongoDB, and extensively studied in the literature [6,8,9,18,24]. It guarantees the causal dependency relationship between transactions. Causal consistency requires that if a

This work is supported in part by the Agence National de Recherche project AdeCoDS (ANR-19-CE25-0007).

transaction t_1 affects another transaction t_2 (e.g., t_1 executes before t_2 in the same process or t_2 reads a value written by t_1) then the two transactions are observed by any other transaction in this order. Concurrent transactions, which are not causally related to each other, can be observed in different orders.

Modern distributed applications often use state replication to reduce latency when serving users from different parts of the world. Therefore, an essential property of these applications is ensuring that the different replicas (processes) eventually converge to the same state, ensuring consistent data across all processes. Recent works have proposed several weak consistency models that guarantee state convergence [11,15–17,21,25].

In this paper, we focus on consistency models that guarantee both causal dependency and state convergence. In particular, we study three variations of causal consistency (CC) that guarantee state convergence across processes. We investigate a log-based CC model [17], a commutativity-based CC model [15], and a global sequence protocol-based CC model [11].

In the log-based CC model (LbCC), processes apply received updates immediately and each process keeps a log of all updates applied to the state. When a process receives a delayed update and more recent concurrent updates have already been applied to the state of the process, the process undoes the more recent updates. Then, the process applies the delayed update and uses the log to redo the more recent updates. Thus, concurrent updates are eventually applied by all processes in the same order (according to some total order mechanism like timestamps), ensuring that the processes converge towards the same state. In the commutativity-based CC model (CbCC), processes coordinate to apply conflicting updates that are concurrent in the same order. In particular, updates that are not commutative, i.e., the states reached when t_1 and t_2 are executed in this order and when t_2 and t_1 are executed in this order are different, are delivered to all process in the same order. Thus, the only updates that might be applied in different orders by different processes are commutative updates. This ensures that all processes converge toward the same state. In the global sequence protocol-based CC model (GSP), a process serving as the server coordinates all updates and propagates them as a single sequence to all processes. Specifically, processes transmit their updates to the server process, which adds them to the update sequence and then propagates the sequence to all processes. This ensures that all processes converge to the same state by applying the updates in the same order.

Indeed, all three models mentioned above ensure causal dependency and state convergence properties. However, an important question remains open regarding which model is the weakest or strongest. In particular, since a weaker consistency model provides better performance, it is important to identify the weakest level of consistency required by an application program to meet its specifications. In this paper, we aim to study the causal consistency variations mentioned above to characterize the relationships between them. To facilitate the study of the different causal consistency models, we use a common formalism to define them. Adopting a common formalism for different consistency models helps us to better understand these models and the differences between them. We then demonstrate that the LbCC model is the weakest, while the CbCC and the GSP models are incomparable.

We also present sufficient conditions for a given application program $\mathcal{P}$ designed under a CC model S to have different behaviors when run under a weaker CC model W. This property of the application program is generally known as *non-robustness* against substituting S with W. It implies that the specification of P may not be preserved when weakening the consistency model (from S to W).

The rest of the paper is organized as follows. We present a preliminary for program semantics that we use to formalize the paper's contribution in Sect. 2. We then provide the formal definitions for the three CC models we study in Sect. 3. In Sect. 4, we illustrate through simple programs the differences between the three CC models. We then present the characterizations of non robustness against substituting LbCC with CbCC and LbCC with GSP. Finally, we discuss related work in Sect. 6 and conclude the paper in Sect. 7.

2 Programs

To simplify the technical exposition, we assume a program $\mathcal{P}$ is a parallel composition of a bounded number of *processes* distinguished using a set of identifiers $\mathbb{P}$ and each process is a sequence of a bounded number of transactions. We use transactions to encapsulate accesses to the program state that is kept in a storage system. We assume a transaction is a function that takes a program state and returns a tuple of another program state and a return value.

Definition 1. *A transaction t is a function from a state (pre-state) s to a state (post-state) s' that returns a value $ret \in V \cup \perp$, i.e., $t(\mathsf{s}) = (\mathsf{s}', ret)$.*

A transaction t is an update (write) if there exist a state s s.t. $t(\mathsf{s}) = (\mathsf{s}', \perp)$ where $\mathsf{s} \neq \mathsf{s}'$. A transaction t is a query (read) if for every state s, $t(\mathsf{s}) = (\mathsf{s}, ret)$ and $ret \neq \perp$.

Definition 2. *An execution ρ of a program $\mathcal{P}$ is the execution of all its transactions by all processes. We use $\mathsf{s}_{\mathcal{P}}$ to denote the program state and use $\mathsf{s}_{\mathcal{P}}^{p}$ to denote the state of a process p, i.e., $\rho : \mathsf{s}_{\mathcal{P}} = (\mathsf{s}_{\mathcal{P}}^{p0}, \mathsf{s}_{\mathcal{P}}^{p1}, \cdots, \mathsf{s}_{\mathcal{P}}^{pn}) \rightarrow \mathsf{s'}_{\mathcal{P}} = (\mathsf{s'}_{\mathcal{P}}^{p0}, \mathsf{s'}_{\mathcal{P}}^{p1}, \cdots, \mathsf{s'}_{\mathcal{P}}^{pn})$.*

Next we define commutativity conditions between two update transactions and between an update and query transactions.

Definition 3. *Two update transactions t_1 and t_2 are commutative if for every state s, we have $t_1(s) = (s_1, \perp)$ and $t_2(s_1) = (s', \perp)$, and $t_2(s) = (s_2, \perp)$ and $t_1(s_2) = (s', \perp)$.*

Definition 4. *An update transaction t_1 and a query transaction t_2 are commutative if for every state s, we have $t_1(s) = (s', \perp)$, $t_2(s') = (s', val)$, and $t_2(s) = (s, val)$.*

The update transactions t_1 and t_2 in Fig. 1 are not commutative since executing t_1 after t_2 leads to $x = 1$ while executing t_2 after t_1 leads to $x = 2$. On the other hand, the two update transactions t_1 and t_3 are commutative since executing t_1 after t_3 leads to $x = 3$ which is the same state outcome when executing t_3 after t_1. The update transaction t_1 and the query transaction t_4 are not commutative since executing t_4 after t_1 results in $r = 1$. Note that also the update transaction t_2 and the query transaction t_4 are not commutative since executing t_4 followed by t_2 in the state $s : \{x = 1\}$ (after executing the update t_1) results in $r = 2$.

t_1 $[x := x + 1]$ t_2 $[x := 2 * x]$ t_3 $[x := x + 2]$ t_4 $[r := x]$ t_5 $[if\ x + 1 < 2$
$x := x + 1]$

Fig. 1. A program with five transactions. Initially $x = 0$.

To decide whether transactions reordering is observable we now define distinguishable criterion between update transactions. In particular, a query transaction can distinguish between the ordering of two update transactions t_2 and t_3, if its return value changes when the order of applying the updates changes.

Definition 5. *Given two update transactions t_2 and t_3, a query transaction t_1 distinguishes between t_2 and t_3 versus t_3 and t_2 iff for every state* s *we have $t_2(\mathsf{s}) = (\mathsf{s}_1, \perp)$, $t_3(\mathsf{s}_1) = (\mathsf{s}_2, \perp)$, $t_1(\mathsf{s}_2) = (\mathsf{s}_2, val)$, and $t_3(\mathsf{s}) = (\mathsf{s}'_1, \perp)$, $t_2(\mathsf{s}'_1) = (\mathsf{s}'_2, \perp)$, and $t_1(\mathsf{s}'_2) = (\mathsf{s}'_2, val')$ where $val \neq val'$.*

Next we extend the distinguishable criteria to a query transaction t_1 that returns different values if it occurred after an update transaction t_2 alone versus if it occurred after the update transactions t_2 followed by t_3 in this order.

Definition 6. *Given two update transactions t_2 and t_3, a query transaction t_1 distinguishes between t_2 versus t_2 and t_3 iff for every state* s *we have $t_2(\mathsf{s}) = (\mathsf{s}_1, \perp)$, $t_3(\mathsf{s}_1) = (\mathsf{s}_2, \perp)$, $t_1(\mathsf{s}_1) = (\mathsf{s}_1, val)$, and $t_1(\mathsf{s}_2) = (\mathsf{s}_2, val')$ where $val \neq val'$.*

In Fig. 1, the query transaction t_4 distinguishes between t_1 versus t_1 followed by t_3. On the other hand, it does not distinguish between t_1 versus t_1 followed by t_5.

2.1 Execution

We use $s^{\mathcal{P}} = (s^{p0}, s^{p1}, \cdots, s^{pn})$ to denote of the state of the program $\mathcal{P}$ constituted of n processes. We use $t^{\mathsf{s}^p} : (s^{p0}, \cdots, \mathsf{s}^p, \cdots, s^{pn}) \rightarrow (s^{p0}, \cdots, t(\mathsf{s}^p), \cdots, s^{pn})$ to denote the transition of applying the transaction t to the state s^p of the process p. We use $\mathbb{T}$ to denote the set of transaction transition identifiers.

We say that an update transaction is initiated by a process, if the transaction was first executed by this process and the process propagated the transaction to other processes. A query transaction is executed by a single process that initiated the query. We use $\mathsf{pr}(t)$ to denote the process initiating the transaction t.

Definition 7. *An execution $\rho = t^{\mathsf{s}^p} \cdots t'^{\mathsf{s}^{p'}} \subset \mathbb{T}$ of a program $\mathcal{P}$ is a sequence of transitions the execution of transactions in the program $\mathcal{P}$, i.e., $\rho : s^{\mathcal{P}} = (s^{p0}, s^{p1}, \cdots, s^{pn}) \rightarrow s'^{\mathcal{P}} = (s'^{p0}, s'^{p1}, \cdots, s'^{pn})$.*

A well-formed execution is an execution where the first occurrence of a transition t^{s^p} of a transaction t corresponds to the transition of applying t to the state s^p of the process p that initiated the transaction t, i.e., $\mathsf{pr}(t) = p$. For the rest of the paper, we assume that every execution is a well-formed execution.

We define a program order as a relation between transactions that are initiated by the process. In particular, given an execution ρ, such that the transition $t_1(s^p)$ occurs in ρ before the transition $t_2(s'^p)$ where $p = \mathsf{pr}(t_1) = \mathsf{pr}(t_2)$ then $(t_1, t_2) \in \mathsf{PO}$. Next, we define the causality relationship between transactions in an execution.

Definition 8. *Given an execution ρ, we say a transaction t_1 is causally related to another transaction t_2 if the transition $t_2(s^p)$ occurs in ρ before the transition $t_1(s'^p)$ where p is the process initiating the transaction t_1 and s^p and s'^p are states of p, denoted by $(t_2^{\mathsf{s}^p}, t_1^{\mathsf{s}'^p}) \in \mathsf{CO}$.*

We say an execution ρ satisfies causal delivery if for every transaction t_1 that is causally related to another transaction t_2 then $\forall\ t_2^{\mathsf{s}^{p'}}\ t_1^{\mathsf{s}'^{p'}} \in \rho.\ (t_2^{\mathsf{s}^{p'}}, t_1^{\mathsf{s}'^{p'}}) \in \mathsf{CO}$. For the rest of the paper, we assume all program executions satisfy causal delivery. Thus, for simplicity if t_1 is causally related to another transaction t_2, we use $(t_2, t_1) \in \mathsf{CO}$. Note that $\mathsf{PO} \subset \mathsf{CO}$.

Using the causal relation we define the conflict relation between transactions that are not commutative and are not related by the causal order.

Definition 9. *Given an execution ρ, we say a query transaction t_1 is conflicting with an update transaction t_2, if the two are not commutative and are not related by the causal order, i.e., $(t_2, t_1) \notin \mathsf{CO}$. We use $(t_1, t_2) \in \mathsf{CF}$ to denote the conflict relation.*

Using the commutative information between transactions in a program we introduce the commutativity relation between them.

Definition 10. *Given an execution ρ, we say two update transactions t_1 and t_2 are related by commutativity relation, denoted by $(t_2, t_1) \in \mathsf{CM}$, if the two transactions do not commute and there exist a process p where the transition $t_2(s^p)$ occurs in ρ before the transition $t_1(s'^p)$ where s^p and s'^p are states of p.*

Next, we define the store order relation. Generally, the store order relation orders update transactions that write to a common variable. However, since we use an abstract model for transactions, thus it is not possible to statically know the variables that those transactions access. This is in contrast with traditional modeling of transactions as a sequence of simple register read and write operations. Therefore, to define the store order between update transactions we check whether they lead to observably distinguishable states. This allows to only order transactions that modify the state in an observable way. For instance, two transactions that write the same value will not be ordered by the store order.

Definition 11. *Given an execution ρ, we say two update transactions t_1 and t_2 are related by store order, denoted by $(t_2, t_1) \in \mathsf{ST}$, if there exist a process p where the transition $t_2(s^p)$ occurs in ρ before the transition $t_1(s'^p)$ where s^p and s'^p are states of p and one of the followings hold:*

- *$(t_2, t_1) \in \mathsf{CM}$; or*
- *there exist a query transaction t_3 in ρ s.t. one of the followings hold:*
  - *t_3 distinguishes between t_1 versus t_1 and t_2; or*
  - *t_3 distinguishes between t_2 versus t_2 and t_1.*

3 Consistency Models

In this section we present the three models of causal consistency using a set of constraints on the relations between transactions in a program execution.

3.1 Log-Based Causal Consistency (LbCC)

Kleppmann et al. [17] present a consistency model that allows concurrent move operations for replicated tree data structures, while guaranteeing that all replicas converge without compromising availability. The main idea of the model consists of keeping a log of operations by all replicas. Thus, replicas apply operations immediately when they are received. If a replica receives operations out-of-order, it can then undo and redo the operations to follow the correct order by fetching previously applied operations from its log to redo them.

Indeed, this operational model of using transactions logging is compatible with different data structures and protocols for message propagation between processes. In this paper, we focus on an instantiation of this operational model with the causal delivery protocol. This model guarantees both convergence and causal consistency. Thus, the store order relation between update transactions is acyclic, i.e., ST is acyclic. The definition of causal dependency and conflict relation imposes that their composition is acyclic, i.e., $(\mathsf{CO}; \mathsf{CF})$ is acyclic (; denotes the sequential composition of two relations). Also, note that the causal delivery ensures that ST are CO compatible, i.e., $\nexists\, t_1,\ t_2.\ (t_1, t_2) \in \mathsf{CO}$ *and* $(t_2, t_1) \in \mathsf{ST}$.

Definition 12. *We say an execution is allowed under the log-based causal consistency model (LbCC) iff* $(\mathsf{CO}; \mathsf{CF})$ *and* ST *are acyclics and* $\nexists\, t_1\ t_2.\ (t_1, t_2) \in \mathsf{CO}$ *and* $(t_2, t_1) \in \mathsf{ST}$.

3.2 Commutativity-Based Causal Consistency (CbCC)

Houshmand and Lesani [15] propose a consistency model that synchronizes conflicting and dependent operations to ensure the integrity and convergence of the state in a replicated setting. They statically analyze state operations to infer conflicting and dependent operations and present distributed protocols that ensure coordination between them.

In this paper, we focus on an instantiation of this operational model with a causal delivery protocol. This consistency model guarantees causal consistency, ensuring that dependent operations are always delivered in order. Furthermore, for simplicity, we assume that conflicting operations correspond to update operations that are not commutative. Thus, the consistency model guarantees that update operations that are not commutative are received in the same order by all processes, thanks to the distributed protocol. Thus, CM is acyclic and implies causality relation, i.e., $\mathsf{CM} \subset \mathsf{CO}$.

Definition 13. *We say an execution is allowed under the commutativity-based causal consistency model (CbCC) iff* $(\mathsf{CO}; \mathsf{CF})$ *and* CM *are acyclic,* $\mathsf{CM} \subset \mathsf{CO}$*, and* $\nexists\ t_1\ t_2.\ (t_1, t_2) \in \mathsf{CO}$ *and* $(t_2, t_1) \in \mathsf{ST}$*.*

3.3 Global Sequence Protocol (GSP)

Burckhardt et al. [11] present a consistency model that utilizes a server (a process that plays the role of a server for other processes[1]) to receive and arrange all exchanged operations, i.e., update transactions, into a totally ordered sequence and forward the sequence to other processes. Processes can delay transmitting their update transactions to the server and buffer those updates to be able to access them locally when they execute query transactions.

The GSP consistency model ensures that all processes agree on a global sequence of update transactions. Therefore, when a process receives an update operation, it must have received all update transactions that were ordered before it in the sequence, i.e., $\mathsf{ST}; (\mathsf{CO} \setminus \mathsf{PO}) \subset \mathsf{CO}$. The global sequencing of operations ensures that the store order relation between update transactions is acyclic, i.e., ST is acyclic.

Definition 14. *We say an execution is allowed under the global sequence protocol model (GSP) iff* $\mathsf{ST}; (\mathsf{CO} \setminus \mathsf{PO}) \subset \mathsf{CO}$*,* $(\mathsf{CO}; \mathsf{CF})$ *and* ST *are acyclic, and* $\nexists\ t_1,\ t_2.\ (t_1, t_2) \in \mathsf{CO}$ *and* $(t_2, t_1) \in \mathsf{ST}$*.*

Note that from GSP and LbCC definitions, that GSP semantics is stronger than LbCC semantics.

Lemma 1. *GSP semantics model is stronger than LbCC semantics model.*

4 Relations Between Consistency Semantics

We will illustrate the differences between the various consistency models using the programs shown in Fig. 2. In all programs, transactions initiated by the same process are aligned vertically and ordered from top to bottom using the program order relation. Each query transaction is commented with the value it reads in the execution. Additionally, in all programs, the state consists of a single variable, x. We assume that initially $x = 2$.

[1] Note that the reliance on one server can create a single point of failure, however, processes can adopt a Byzantine fault tolerance consensus approach to elect new process to play the role of server.

Compared to CbCC and GSP semantics, LbCC allows processes to apply update transactions immediately when received. The program in Fig. 2a contains six transactions that are issued by four distinct processes. All transactions manipulate and query the state of the variable x. We emphasize an execution where t_3 reads 3, t_5 reads 4, and both t_4 and t_6 read 6. This execution is allowed under LbCC since the two update transactions t_1 and t_2 are not causally dependent and are received in different orders by the processes. Thus, one process first receives t_1 and applies it to its state, then performs the query t_3. It then receives t_2 and updates its state[2] and performs the second query t_3. The other process first receives t_2 and applies it to its state, then performs the query t_5. It then receives t_1, which results in undoing t_2 and applying t_1, then t_2 to the process's state. Finally, the last query t_6 is performed. However, this execution is not feasible under CbCC. In particular, the two transactions t_1 and t_2 are not commutative. Thus, they must be received in order by all processes. If we assume $(t_1, t_2) \in \mathsf{CO}$, then it is not possible that the query t_5 reads only the update of t_2. Similarly, the above execution is not possible under GSP as well. In particular, the two transactions t_1 and t_2 will be arranged by the server before propagating them to other processes in a sequence. Thus, assuming that $(t_1, t_2) \in \mathsf{ST}$ (note that $(t_1, t_2) \in \mathsf{CM}$), then $(t_2, t_5) \in (\mathsf{CO} \setminus \mathsf{PO})$ implies that $(t_1, t_5) \in \mathsf{CO}$, which contradicts the fact that the query t_5 reads only the update of t_2.

$$\begin{tikzcd}
t_1\ [x := x + 1] & t_2\ [x := 2 * x] & t_3\ [r1 := x]\ //3 \arrow[d, "\mathsf{PO}"'] & t_5\ [r3 := x]\ //4 \arrow[d, "\mathsf{PO}"'] \\
 & & t_4\ [r2 := x]\ //6 & t_6\ [r4 := x]\ //6
\end{tikzcd}$$

(a) A LbCC execution but not GSP or CbCC.

$$\begin{tikzcd}
t_1\ [x := x + 1] \arrow[d, "\mathsf{PO}"'] & t_2\ [x := 2 * x] \arrow[d, "\mathsf{PO}"'] \\
t_3\ [r1 := x]\ //3 \arrow[d, "\mathsf{PO}"'] & t_5\ [r3 := x]\ //4 \arrow[d, "\mathsf{PO}"'] \\
t_4\ [r2 := x]\ //6 & t_6\ [r4 := x]\ //6
\end{tikzcd}$$

(b) A LbCC and GSP execution but not CbCC.

$$\begin{tikzcd}
t_1\ [x := x + 1] & t_2\ [x := x + 2] & t_3\ [r1 := x]\ //3 \arrow[d, "\mathsf{PO}"'] & t_5\ [r3 := x]\ //4 \arrow[d, "\mathsf{PO}"'] \\
 & & t_4\ [r2 := x]\ //5 & t_6\ [r4 := x]\ //5
\end{tikzcd}$$

(c) A LbCC and CbCC execution but not GSP.

Fig. 2. Litmus programs. Initially $x = 2$.

In Fig. 2b, we show a program with six transactions initiated by two distinct processes. We highlight an execution where t_3 reads 3, t_5 reads 4, and both t_4 and

[2] We assume that t_1 is ordered before t_2 based on some mechanism like timestamps.

t_6 read 6 from x. This execution is allowed under GSP (and also under LbCC). In particular, the second process locally buffers transaction t_2 and delays the propagation of the update. It then performs the query t_5, obtaining its value by inspecting the updates stored in the buffer and its state. When t_1 is received, the process applies it to its state and then pops t_2 from the buffer and applies it to its state. Finally, it performs the last query t_6. On the other hand, the first process first applies the update t_1 to its state (without buffering) and then performs its first state query, t_3. Then it receives the second update t_1 and applies it to its state, performing the second state query, t_4. However, this execution is not feasible under CbCC. In particular, the two transactions t_1 and t_2 are not commutative. Thus, the two processes must coordinate and apply them to their states in the same order. In particular, if we assume $(t_1, t_2) \in \mathsf{CO}$, then it is not possible for query t_5 to read only the update of t_2.

In Fig. 2c, we show a program with six transactions that are initiated by four distinct processes. We highlight an execution where t_3 reads 3, t_5 reads 4 and both t_4 and t_6 read 5 from x. This execution is allowed under CbCC (and also under LbCC). In particular, the two transactions t_1 and t_2 are commutative. Thus, in CbCC processes will not coordinate and t_1 and t_2 can be applied in any order. However, this execution is not feasible under GSP. In particular, the two transactions t_1 and t_2 will be arranged by the server before propagating them to other processes in a sequence. Thus, if we assume $(t_1, t_2) \in \mathsf{ST}$ (note that t_3 distinguishes between t_1 versus t_1 and t_2) then $(t_2, t_5) \in (\mathsf{CO} \setminus \mathsf{PO})$ implies that $(t_1, t_5) \in \mathsf{CO}$ which contradicts the fact the query t_5 reads only the update of t_2.

The programs in Figs. 2b and 2c show that the two semantics CbCC and GSP are incomparable.

Lemma 2. *CbCC and GSP semantics models are incomparable.*

5 Robustness

In this section, we present sufficient criteria to characterize programs that have distinct behaviors under the different semantics models. In particular, we characterize programs that have behaviors which are possible under LbCC but are not possible under CbCC and GSP. We formulate those comparisons as robustness of the programs against LbCC relative CbCC and GSP.

We adopt a *return-value-based* robustness criterion such that a program is robust iff the set of all its return values under the weak semantics is the same as its set of return values under the strong semantics. More precisely, we say that a program $\mathcal{P}$ is not robust against a semantics X relative to another stronger semantics Y iff for every execution ρ of $\mathcal{P}$ under X we have a replica execution ρ' of $\mathcal{P}$ under Y where both executions contain the same executed transactions and the values returned by query transactions are the same.

Definition 15. *Given two consistency models X and Y, we say that a program $\mathcal{P}$ is robust against X relative to Y iff for every execution ρ of $\mathcal{P}$ under the semantics model X there must exist an execution ρ' of $\mathcal{P}$ under the semantics model Y s.t.*

- *for every state transition by a process p in ρ, i.e., $t(s_{\rho}^{p}) = (s_{\rho}^{\prime p}, val) \in \rho$, there exists a state transition by p in ρ', i.e., $t(s_{\rho'}^{p}) = (s_{\rho'}^{\prime p}, val) \in \rho'$, where the two transitions return the same value;*
- *for every update only state transition by a process p of a transaction t in ρ, i.e., $t(s_{\rho}^{p}) = (s_{\rho}^{\prime p}, \perp) \in \rho$, there exists a state transition by p of the transaction t in ρ', i.e., $t(s_{\rho'}^{p}) = (s_{\rho'}^{\prime p}, \perp) \in \rho'$.*

Note that in our definition of robustness, there are no programs that are not robust against CbCC relative to LbCC. Even though under CbCC update transactions that are commutative can be applied in different order by different processes which results in a cycle in the store order relation ST which is not allowed under LbCC semantics, this will not result in a query transaction returning a value under CbCC that is not possible to observe under LbCC. This is because the update transactions are commutative.

Lemma 3. *All programs are robust against CbCC relative to LbCC.*

In the following we give sufficient conditions based on a set of constraints of the transactions in a given program to characterize whether this program is robust or not against LbCC relative to CbCC or GSP.

5.1 LbCC Versus CbCC

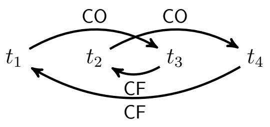
The following robustness characterization theorem presents a sufficient conditions to characterize programs that are not robust against LbCC relative to CbCC as illustrated in Fig. 3. The theorem follows from the fact that CbCC enforces coordination between t_1 and t_2 since they are not commutative. Thus, in an execution where t_4 observes the updates of t_1 and t_2 in this order (or observes the update of t_1 without the update of t_2), this implies that any other process must observe these two updates in this order (or cannot observe the update of t_2 without observing the update of t_1). Therefore, it is not possible for t_3 to observe the update of t_2 without observing the update of t_1 under CbCC.

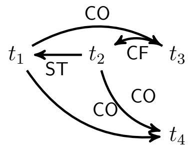
However, under LbCC, no coordination is enforced between t_1 and t_2, and t_3 can observe the update of t_2 without observing the update of t_1, while t_4 observes the updates of t_1 and t_2 in this order (or observes the update of t_1 without the update of t_2). When t_1 is received, the process will undo the update of t_2, apply the update of t_1, and then redo the update of t_2.

Theorem 1. *A program $\mathcal{P}$ is not robust against LbCC relative to CbCC if there exist four transactions t_1, t_2, t_3, and t_4 in $\mathcal{P}$ that satisfy the following:*

- *t_1 and t_2 are updates that are not commutative and are initiated by different processes.*
- *t_3 distinguishes between t_2 versus t_1 and t_2.*
- *One of the following holds:*
  - *t_4 distinguishes between t_1 and t_2 versus t_2 and t_1; or*
  - *t_4 distinguishes between t_1 versus t_2 and t_1.*

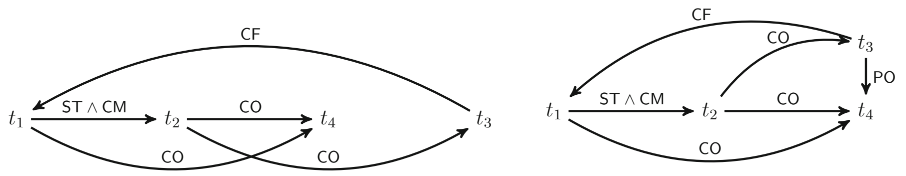
Proof. Let $\mathcal{P}$ with four transactions t_1, t_2, t_3, and t_4 that satisfy the conditions in the theorem statement. The following are possible executions of $\mathcal{P}$ under LbCC:





Notice that in both executions the two queries receive the updates in different orders. In the first execution, we have $(t_1, t_3) \in \mathsf{CO}$ and $(t_3, t_2) \in \mathsf{CF}$ which implies that $(t_2, t_1) \notin \mathsf{ST}$ under CbCC, otherwise, we get contradiction if $(t_2, t_1) \in \mathsf{ST}$ resulting in $(t_2, t_1) \in \mathsf{CO}$ under CbCC since the two transactions are not commutative. Thus, $(t_1, t_2) \in \mathsf{ST}$ which results in a contradiction as well since $(t_2, t_4) \in \mathsf{CO}$ and $(t_4, t_1) \in \mathsf{CF}$. In the second execution, we have $(t_2, t_1) \in \mathsf{ST}$ which implies that under CbCC $(t_2, t_1) \in \mathsf{CO}$ which contradicts the fact that we have $(t_1, t_3) \in \mathsf{CO}$ and $(t_3, t_2) \in \mathsf{CF}$ in the execution.

In Fig. 2a, we have four query transactions in the program which is not robust against LbCC relative to CbCC. However, note that the two query transactions t_3 and t_5 (representing t_4 and t_3 in Theorem 1, respectively) are sufficient with the two update transactions t_1 and t_2 to show that this program is not robust against LbCC relative to CbCC. In Fig. 2b, we show another program that is not robust against LbCC relative to CbCC. Note that in this program t_1 and t_2 are initiated by the same processes initiating t_4 and t_3, respectively.



(a) T3 and T4 are initiated by different processes.

(b) T3 and T4 are initiated by the same process.

Fig. 3. Characterization of programs that are not robust against LbCC relative to CbCC.

5.2 LbCC Versus GSP

Next we present sufficient conditions to characterize programs that are not robust against LbCC relative to GSP as shown in Fig. 4. The theorem follows from the fact that GSP enforces a global sequencing between t_1 and t_2. If, in an execution, t_3 observes the update of t_2 but not the update of t_1, this implies that the server did not order t_1 before t_2 in the global sequencing of the operations.

Thus, it is not possible for t_4 to observe the update of t_1 without observing the update of t_2 under GSP semantics.

On the other hand, under LbCC semantics, since no global sequence is enforced between t_1 and t_2, t_4 can observe the update of t_1 without observing the update of t_2. When t_2 is received, the process will undo the update of t_1, apply the update of t_2, and then redo the update of t_1, assuming that t_2 is timestamped before t_1.

Theorem 2. *A program $\mathcal{P}$ is not robust against LbCC relative to GSP if there exist four transactions t_1, t_2, t_3, and t_4 in $\mathcal{P}$ that satisfy the following:*

- *t_1 and t_2 are updates related by ST and initiated by distinct processes.*
- *t_3 and t_4 are queries initiated by processes different from the processes of t_1 and t_2.*
- *t_3 distinguishes between t_2 versus t_1 and t_2.*
- *One of the following holds:*
  - *t_4 distinguishes between t_1 and t_2 versus t_2 and t_1; or*
  - *t_4 distinguishes between t_1 versus t_2 and t_1.*

Proof. Similar to before let $\mathcal{P}$ with four transactions t_1, t_2, t_3, and t_4 that satisfy the conditions in the theorem statement. The following are possible executions of $\mathcal{P}$ under LbCC:

$$\begin{tikzcd}
t_1 \arrow[rr, bend left, "\mathsf{CO}"] & t_2 \arrow[rr, bend left, "\mathsf{CO}"] & t_3 \arrow[l, bend left, "\mathsf{CF}"] & t_4 \arrow[lll, bend left, "\mathsf{CF}"]
\end{tikzcd}
\qquad
\begin{tikzcd}
t_1 \arrow[rr, bend left, "\mathsf{CO}"] \arrow[rrd, bend right, "\mathsf{CO}"'] & t_2 \arrow[l, "\mathsf{ST}"] \arrow[rd, "\mathsf{CO}"] & t_3 \arrow[l, bend right, "\mathsf{CF}"'] \\
 & & t_4
\end{tikzcd}$$

In both executions, the two queries receive the updates in different orders. In the first execution, we have $(t_1, t_3) \in \mathsf{CO}$ and $(t_3, t_2) \in \mathsf{CF}$ which implies that $(t_2, t_1) \notin \mathsf{ST}$ under GSP, otherwise, we get contradiction if $(t_2, t_1) \in \mathsf{ST}$ resulting in $(t_2, t_1) \in \mathsf{CO}$ under GSP since the two transactions are not originating from the same processes as t_1. Thus, $(t_1, t_2) \in \mathsf{ST}$ which results in a contradiction as well since $(t_2, t_4) \in \mathsf{CO}$ and $(t_4, t_1) \in \mathsf{CF}$. Similarly, in the second execution, we have $(t_2, t_1) \in \mathsf{ST}$ which implies that under GSP $(t_2, t_1) \in \mathsf{CO}$ (store order relation between transactions originating from different processes implies visibility relation) which contradicts the fact that we have $(t_1, t_3) \in \mathsf{CO}$ and $(t_3, t_2) \in \mathsf{CF}$ in the execution.

Similar to CbCC in Fig. 2a, the two query transactions t_4 and t_5 (representing t_4 and t_3 in Theorem 2, respectively) are sufficient, along with the two update transactions t_1 and t_2, to show that this program is not robust against LbCC relative to GSP. In Fig. 2c, we present another program that is not robust against LbCC relative to GSP. It should be noted that in this program, t_1 and t_2 are commutative.

$$\begin{tikzcd}
t_1 \arrow[r, "\mathsf{ST}"] \arrow[rr, bend right, "\mathsf{CO}"'] & t_2 \arrow[r, "\mathsf{CO}"] \arrow[rr, bend right, "\mathsf{CO}"'] & t_4 & t_3 \arrow[lll, bend right, "\mathsf{CF}"']
\end{tikzcd}$$

Fig. 4. Characterization of programs that are not robust against LbCC relative to GSP.

6 Related Work

There have been several studies investigating the problem of data consistency for applications that allow for concurrent modification of distributed replicas [9–11,15–17,25,26]. These works commonly examine consistency models that guarantee causal dependency and state convergence properties, as they offer a good balance between strong consistency and availability, and are suitable for various types of replicated applications. Different variations of causal consistency have been explored in the literature [6,9,10,24]. Additionally, several consistency models that ensure the state convergence property have been introduced recently [11,15–17,25].

In this paper we aimed to understand the relationships between three consistency protocols (LbCC, CbCC, and GSP) that ensure causal dependency and state convergence properties, using different approaches. To the best of our knowledge, this is the first work to classify and characterize the differences between these protocols. Our work is similar in spirit to the work of Shapiro et al. on consistency models classification [26].

In our characterization of the differences between the consistency models, we adopted a *return-value-based* robustness that is distinct from the standard *trace-based* and *state-based* robustness criterion. In state-based robustness, a program is robust iff the set of all its reachable states under the weak semantics is the same as its set of reachable states under the strong semantics. State-based robustness requires computing the set of reachable states under the weak semantics, which is in general a hard problem. Abdulla et al. [1] and Lahav and Boker [18] have investigated the decidability and the complexity computing the set of reachable states under release-acquire and causal consistency semantics, respectively, showing that it is either decidable but highly complex (non-primitive recursive), or undecidable. On the other hand, in trace-based robustness, a program is robust iff the set of all its execution traces under the weak semantics is the same as its set of execution traces under the strong semantics. Trace-based robustness has been investigated for several weak consistency models including causal consistency [2–5,7,8,13,19,22].

7 Conclusion

Towards comparing the three causal convergence consistency models, we have defined the three consistency models using a common formalism scheme allowing us to classify them. We showed that log-based causal consistency (LbCC) is the weakest while commutativity-based causal consistency (CbCC) and global sequence protocol (GSP) are incomparable. We then developed a robustness criterion based on return values to characterize programs that admit different behaviors under LbCC relative to CbCC and GSP.

In the future we plan to extend our work along a few dimensions to build on the work presented in this paper. First, we plan to investigate the relationship between the three causal consistency models and the stronger models such as prefix consistency [12] and serializability [23]. Second, we plan to develop a benchmark set of programs to compare the three causal consistency models using this benchmark set. Finally, we plan to investigate synchronization primitives to use to repair non-robust programs to become robust.

References

1. Abdulla, P.A., Arora, J., Atig, M.F., Krishna, S.N.: Verification of programs under the release-acquire semantics. In: McKinley, K.S., Fisher, K. (eds.) Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22–26, 2019, pp. 1117–1132. ACM (2019). https://doi.org/10.1145/3314221.3314649. https://doi.org/10.1145/3314221.3314649
2. Beillahi, S.M., Bouajjani, A., Enea, C.: Checking robustness against snapshot isolation. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11562, pp. 286–304. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25543-5_17
3. Beillahi, S.M., Bouajjani, A., Enea, C.: Robustness against transactional causal consistency. In: Fokkink, W.J., van Glabbeek, R. (eds.) 30th International Conference on Concurrency Theory, CONCUR 2019, August 27–30, 2019, Amsterdam, the Netherlands. LIPIcs, vol. 140, pp. 30:1–30:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019). https://doi.org/10.4230/LIPIcs.CONCUR.2019.30
4. Beillahi, S.M., Bouajjani, A., Enea, C.: Checking robustness between weak transactional consistency models. In: ESOP 2021. LNCS, vol. 12648, pp. 87–117. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-72019-3_4
5. Bernardi, G., Gotsman, A.: Robustness against consistency models with atomic visibility. In: Desharnais, J., Jagadeesan, R. (eds.) 27th International Conference on Concurrency Theory, CONCUR 2016, August 23–26, 2016, Québec City, Canada. LIPIcs, vol. 59, pp. 7:1–7:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2016). https://doi.org/10.4230/LIPIcs.CONCUR.2016.7
6. Bouajjani, A., Enea, C., Guerraoui, R., Hamza, J.: On verifying causal consistency. In: Castagna, G., Gordon, A.D. (eds.) Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, 18–20 January 2017, pp. 626–638. ACM (2017). https://doi.org/10.1145/3009837.3009888
7. Brutschy, L., Dimitrov, D.K., Müller, P., Vechev, M.T.: Serializability for eventual consistency: criterion, analysis, and applications. In: Castagna, G., Gordon, A.D. (eds.) Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, 18–20 January 2017, pp. 458–472. ACM (2017). https://doi.org/10.1145/3009837.3009895

8. Brutschy, L., Dimitrov, D.K., Müller, P., Vechev, M.T.: Static serializability analysis for causal consistency. In: Foster, J.S., Grossman, D. (eds.) Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, 18–22 June 2018, pp. 90–104. ACM (2018). https://doi.org/10.1145/3192366.3192415
9. Burckhardt, S.: Principles of eventual consistency. Found. Trends Program. Lang. **1**(1-2), 1–150 (2014). https://doi.org/10.1561/2500000011
10. Burckhardt, S., Gotsman, A., Yang, H., Zawirski, M.: Replicated data types: specification, verification, optimality. In: Jagannathan, S., Sewell, P. (eds.) The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2014, San Diego, CA, USA, 20–21 January 2014, pp. 271–284. ACM (2014). https://doi.org/10.1145/2535838.2535848
11. Burckhardt, S., Leijen, D., Protzenko, J., Fähndrich, M.: Global sequence protocol: a robust abstraction for replicated shared state. In: Boyland, J.T. (ed.) 29th European Conference on Object-Oriented Programming, ECOOP 2015, 5–10 July 2015, Prague, Czech Republic. LIPIcs, vol. 37, pp. 568–590. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2015). https://doi.org/10.4230/LIPIcs.ECOOP.2015.568
12. Cerone, A., Bernardi, G., Gotsman, A.: A framework for transactional consistency models with atomic visibility. In: Aceto, L., de Frutos-Escrig, D. (eds.) 26th International Conference on Concurrency Theory, CONCUR 2015, Madrid, Spain, September 1.4, 2015. LIPIcs, vol. 42, pp. 58–71. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2015). https://doi.org/10.4230/LIPIcs.CONCUR.2015.58
13. Cerone, A., Gotsman, A.: Analysing snapshot isolation. J. ACM **65**(2), 11:1–11:41 (2018). https://doi.org/10.1145/3152396
14. Gilbert, S., Lynch, N.A.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. SIGACT News **33**(2), 51–59 (2002). https://doi.org/10.1145/564585.564601
15. Houshmand, F., Lesani, M.: Hamsaz: replication coordination analysis and synthesis. Proc. ACM Program. Lang. **3**(POPL), 74:1–74:32 (2019). https://doi.org/10.1145/3290387
16. Kaki, G., Priya, S., Sivaramakrishnan, K.C., Jagannathan, S.: Mergeable replicated data types. Proc. ACM Program. Lang. 3(OOPSLA) **154**, 1–154:29 (2019). https://doi.org/10.1145/3360580
17. Kleppmann, M., Mulligan, D.P., Gomes, V.B.F., Beresford, A.R.: A highly-available move operation for replicated trees. IEEE Trans. Parallel Distributed Syst. **33**(7), 1711–1724 (2022). https://doi.org/10.1109/TPDS.2021.3118603
18. Lahav, O., Boker, U.: Decidable verification under a causally consistent shared memory. In: Donaldson, A.F., Torlak, E. (eds.) Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, 15–20 June 2020, pp. 211–226. ACM (2020). https://doi.org/10.1145/3385412.3385966
19. Lahav, O., Margalit, R.: Robustness against release/acquire semantics. In: McKinley, K.S., Fisher, K. (eds.) Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, 22–26 June 2019, pp. 126–141. ACM (2019). https://doi.org/10.1145/3314221.3314604
20. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7), 558–565 (1978). https://doi.org/10.1145/359545.359563

21. Li, C., Porto, D., Clement, A., Gehrke, J., Preguiça, N.M., Rodrigues, R.: Making geo-replicated systems fast as possible, consistent when necessary. In: Thekkath, C., Vahdat, A. (eds.) 10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, 8–10 October 2012, pp. 265–278. USENIX Association (2012), https://www.usenix.org/conference/osdi12/technical-sessions/presentation/li
22. Nagar, K., Jagannathan, S.: Automated detection of serializability violations under weak consistency. In: Schewe, S., Zhang, L. (eds.) 29th International Conference on Concurrency Theory, CONCUR 2018, 4–7 September, 2018, Beijing, China. LIPIcs, vol. 118, pp. 41:1–41:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2018). https://doi.org/10.4230/LIPIcs.CONCUR.2018.41
23. Papadimitriou, C.H.: The serializability of concurrent database updates. J. ACM **26**(4), 631–653 (1979). https://doi.org/10.1145/322154.322158
24. Perrin, M., Mostéfaoui, A., Jard, C.: Causal consistency: beyond memory. In: Asenjo, R., Harris, T. (eds.) Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2016, Barcelona, Spain, 12–16 March 2016, pp. 26:1–26:12. ACM (2016). https://doi.org/10.1145/2851141.2851170, https://doi.org/10.1145/2851141.2851170
25. Preguiça, N.M., Baquero, C., Shapiro, M.: Conflict-free replicated data types (crdts). CoRR abs/1805.06358 (2018). http://arxiv.org/abs/1805.06358
26. Shapiro, M., Ardekani, M.S., Petri, G.: Consistency in 3d. In: Desharnais, J., Jagadeesan, R. (eds.) 27th International Conference on Concurrency Theory, CONCUR 2016, August 23–26, 2016, Québec City, Canada. LIPIcs, vol. 59, pp. 3:1–3:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2016). https://doi.org/10.4230/LIPIcs.CONCUR.2016.3