Consistency-Aware Scheduling for Weakly Consistent Programs

Maryam Dabaghchian, Zvonimir Rakamarić,
Burcu Kulahcioglu Ozkan[a]*, Erdal Mutlu*[b]*,*
Serdar Tasiran[b]*.*

UUCS-17-002

[a]Max Planck Institute for Software Systems Kaiserslautern, Germany
[b]Computer Science Department, Koç University, Istanbul, Turkey

School of Computing
University of Utah
Salt Lake City, UT 84112 USA

November 17, 2017

Abstract

Modern geo-replicated data stores provide high availability by relaxing the underlying consistency requirements. Programs layered over such data stores are called weakly consistent programs. Due to the reduced consistency requirements, they exhibit highly nondeterministic behaviors, some of which might violate program invariants. In general, some behaviors of such programs might violate program invariants, while others might not. Therefore, implementing correct weakly consistent programs and reasoning about them is challenging. In this paper, we present a systematic scheduling approach that is aware of the underlying consistency model. Our approach dynamically explores all possible program behaviors allowed by the used data store consistency model, and it evaluates program invariants during the exploration. We implement the approach in a prototype model checker for Antidote, which is a causally consistent key-value data store with convergent conflict handling. We evaluate our tool on several benchmarks. The results show that our approach is effective in detecting buggy behaviors in weakly consistent programs.

1 Introduction

Modern Internet-scale programs often rely on high-performance geo-replicated data stores. In such data stores, replicas are located in geographically separate locations to avoid latency in the wide area network and tolerate network partitioning. According to the Consistency, Availability, and Partition tolerance (CAP) theorem [26], partitioning is unavoidable, and data stores have to sacrifice either strong consistency or availability. Modern data stores provide high availability through weaker consistency models called *eventual consistency* [40], which curbs costly synchronization requirements. We refer to an atomic step that updates some data in such data stores as an *event*. In general, eventual consistency guarantees that events occurred at each replica will eventually be propagated and become visible on all remote replicas.

Programs using such geo-replicated data stores maintain a copy of their data on different replicas, which can be concurrently updated by connected clients. However, due to the often limited synchronization guarantees, it is possible to have conflicting concurrent events on different replicas. In order to provide eventual consistency, many replicated data types are equipped with conflict resolution mechanisms [11, 12, 36, 17]. Such store types are called *conflict-free replicated data types* (CRDTs) [37, 8], and they guarantee that every replica that receives the same set of events reaches the same state. Hence, all replicas eventually converge to the same state if the system becomes quiescent. In order to provide high availability, such systems never reach the quiescent state, as they continuously receive events from clients. However, we consider only a finite set of events when checking such programs, which ensures the system eventually becomes quiescent.

Due to the relaxed consistency guarantees of the systems using CRDTs, a wider set of program behaviors is possible when compared to a strongly consistent system, some of which are unintuitive. This makes it harder for developers to reason about expected executions of their programs and specify the intended program behavior correctly. For example, on a replica, receiving an event either from a client or from a remote replica are all nondeterministic. In addition, an event might not observe the effect of all previous events from other replicas, and hence can easily inadvertently access stale data. Such subtle *schedules* (i.e., execution orders) can violate the intended invariants of programs written with CRDTs. Hence, developers have to reason about complex schedules of concurrent events that can happen in a CRDT to be able to implement their programs correctly. Alternatively, they can manually specify the consistency rules they deem are needed for the correct execution of their program, which are in turn enforced by the underlying CRDT. This is again a complex and error-prone task requiring deep understanding of the provided consistency guarantees, allowed schedules, and their interplay with the program.

In order to assist the developers in overcoming the challenges of writing correct CRDT programs, we introduce a systematic scheduling approach that is aware of the underlying consistency model. Our approach enables the developers to automatically and systematically explore different scheduling scenarios for their program allowed by the specified consistency model, thereby helping them to detect subtle consistency bugs. Our approach is parameterized both in terms of the used schedule exploration strategy and instantiated consistency model. Moreover, it works in the local development environment, and it does not require the program to be run in a distributed system.

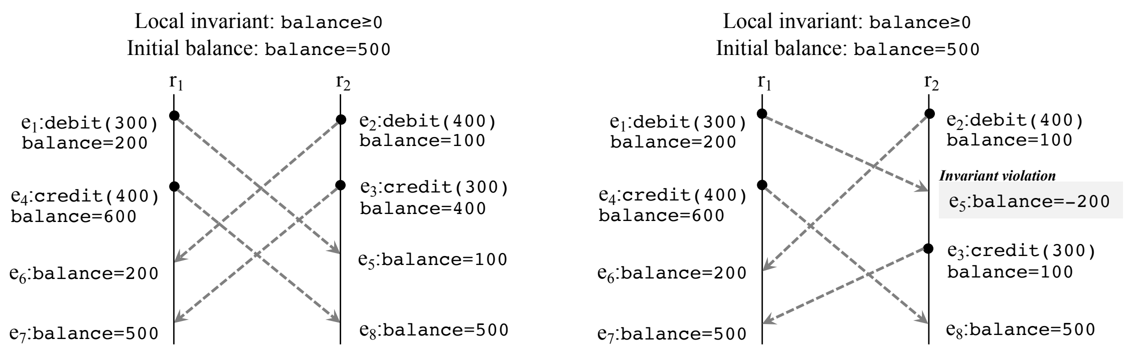
Our approach parameterizes the exploration space by the consistency guarantee of the system, i.e., it is consistency-aware. Different levels of weak consistency enforce different synchronization constraints on the system. Since consistency-aware scheduling takes the consistency guarantee into consideration while generating new schedules, it is precise in the sense that the generated schedules satisfy the consistency requirements. Hence, it neither misses bugs due to exploring only strongly consistent schedules nor reports false bugs by exploring overly relaxed weakly consistent schedules.

Within our approach, we propose two schedule exploration strategies (random and extended delay-bounded [19]) to detect violations of the supplied program invariants. We implement our approach in a tool for the Antidote platform [4, 3], which is a highly available geo-replicated CRDT key-value data store. Our tool helps the developer to properly specify the consistency level needed for their program by providing counterexamples that break the invariants if the chosen consistency is too weak. Finally, we apply our tool on several use cases from the SyncFree project [39], and we successfully detect bug-inducing schedules. Our contributions are summarized as follows:

- We introduce and formalize a consistency-aware schedule exploration approach for weakly consistent systems that is parameterized by the scheduler and consistency model.
- We implement our approach in a prototype tool within the Antidote CRDT platform and include two schedule exploration strategies.
- We evaluate our tool on several benchmarks and and show that it can efficiently find real bugs.

2 Motivating Example

We provide a virtual wallet example to explicate how an interleaving of a weakly consistent program, introduced by time nondeterminism, can result in an invariant violation by



(a) Bug-free scenario (b) Buggy scenario

Figure 1: Two scheduling scenarios in the virtual wallet example.

acting on stale data. Our virtual wallet has a balance data field, a CRDT counter, with an accompanying invariant of having a non-negative value at each replica. The balance can be updated using `credit` and `debit` events, where `debit` decrements the balance value by the specified amount only if the current balance is sufficient. The virtual wallet data is replicated over multiple replicas where clients can connect and apply events concurrently. We implemented the program using a causally consistent [2, 33] geo-replicated data store that guarantees the eventual delivery of each event and convergence of the state in all replicas. Given the initial balance of 500 at every replica, Figure 1 gives two possible scheduling scenarios: one that satisfies and another that violates our invariant.

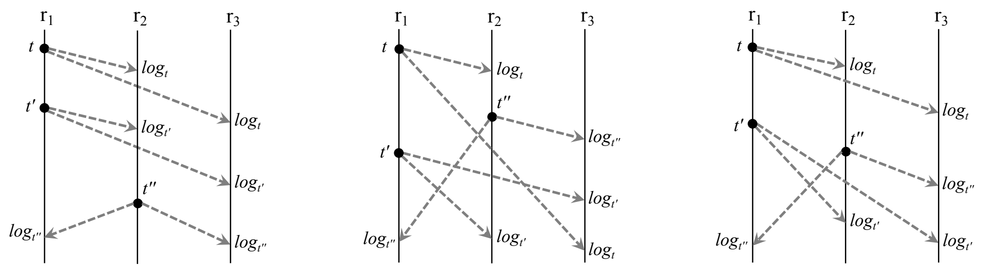
Figure 1a shows a bug-free scheduling scenario. Suppose two clients C_1 and C_2 are connected to two different replicas r_1 and r_2, respectively; the clients are issuing events to the same virtual wallet concurrently. First, C_1 debits 300 from r_1, thereby making the balance 200 (e_1). Then, C_2 debits 400 from r_2 (e_2) and credits 300, thereby making the balance 400 (e_3). Afterwards, C_1 credits 400 on r_1, and the balance becomes 600 (e_4). Now, r_1 propagates the C_1's first event to r_2, making the balance 100 (e_5); r_2 propagates both events issued by C_2 to r_1, which makes the balance 500 (e_6, e_7). Finally, the second event issued by C_1 is propagated to r_2, and the ending balance is 500 (e_8). In this scheduling scenario, the value of balance is always non-negative, and the state of both replicas converged in the end. Hence, a developer might think that the invariant always holds, while that is in fact not the case, as our next scheduling scenario shows.

Figure 1b shows a buggy scheduling scenario, which starts the same as the bug-free one. First, C_1 debits 300 from r_1, making the balance 200 (e_1), and C_2 debits 400 from r_2, making the balance 100 (e_2). Then, C_1 credits 400, making the balance 600 on r_1 (e_4). Differently than in the bug-free scenario, but still allowed by weak consistency, r_1 now propagates C_1's first event to r_2, thereby making the balance value -200 (e_5). This violates our `balance>=0` invariant. Note that the two debit events e_1 and e_2 are concurrent. Due to the nondeterminism in weakly consistent systems, event e_5 can be received either before or after e_3; in fact, it can be received even before e_2! As shown in this schedule, if e_5 is scheduled right after e_2 and right before e_3, the program invariant is violated, although the schedule still guarantees causal consistency. Note that a scheduler guaranteeing a stronger consistency model (e.g., serializability) would fail to detect this bug. To catch such invariant violations, a developer has to take into consideration and be able to explore different orderings that are allowed under the given consistency model of the system; this is hard to accomplish by relying on typical ad-hoc testing techniques. We address this need by providing a consistency-aware schedule exploration approach and a prototype implementation that helps developers discover scheduling scenarios leading to such deep-seated bugs. In this example, the invariant would be preserved if the balance is defined as a CRDT bounded-counter, which enforces strong consistency [35] on decrement operations.

3 Weakly Consistent Programs

We formalize our approach based on the transactional consistency framework proposed by Cerone et al. [13]. Let $Rs = \{r_1, r_2, ..., r_n\}$ be the set of all replicas in the system and $n = |Rs|$ the total number of replicas. We define $Txns$ as the set of messages (transactions) initiated by clients on replicas. We define $Logs$ as the set of messages (transaction logs) transmitting between replicas in the system. Then, $Msgs = (Txns \cup Logs) \times Rs$ is the set of all messages transmitting between clients and replicas or between different replicas. For a message $msg = \langle t, r \rangle$, r denotes the originating replica of the transaction t. We formally define events (i.e., atomic steps in a program) as a set of tuples $Events = Msgs \times Rs \times \mathbb{Z}^n_{\geq 0}$. Each event consists of a message, a replica to which the message is being delivered, and a vector clock [22] denoting a snapshot of the system that captures message dependencies.

Let history $\mathcal{H} \subseteq \wp(Events)$ be the set of events $\{\langle msg, r, vc \rangle \mid vc \prec now()^n\}$ that occurred in the system so far, where now^n denotes the current snapshot replica r has. So, the history at the initial state, denoted by $\mathcal{H}_0$, is an empty set. We define a function $ct : Events \rightarrow \mathbb{Z}^n_{\geq 0}$, such that for every event $e = \langle \langle t, r' \rangle, r, vc \rangle$, $ct(e) = vc\big[r' \mapsto vc[r'] + 1\big]$ shows the visibility vector clock of e. Let Obj be the set of data store objects, and $obj : Events \rightarrow \wp(Obj)$ be a function mapping each event to a subset of objects that the event reads or updates. Then, we define function $relEvents : Events \times Rs \rightarrow \wp(Events)$ mapping every event e to a subset of events that act on at least one shared object as e does on the specified replica. For $e = \langle msg, r, vc \rangle$, $relEvents$ is formally defined as $relEvents(e, r'') = \big\{\langle msg', r', vc' \rangle \mid r'' = r' \wedge obj(e) \cap obj(e') \neq \emptyset\big\}$.



(a) Serializability Cons. (b) Eventual Cons. (c) Causal Cons.

Figure 2: Examples of schedules allowed by different consistency models.

3.1 Consistency Models

In this section, we introduce three well-known consistency models and formalize the dependency restrictions of each model. The three models are *Serializability Consistency* (SR), *Eventual Consistency* (EC), and *Causal Consistency* (CC), and we informally specify them as follows:

Serializability Consistency (SR) guarantees that every transaction observes the effect of all other transactions updating shared objects before executing, and no such transactions are allowed to execute concurrently [35]. Figure 2a gives an example schedule allowed by strong consistency.

Eventual Consistency (EC) guarantees that the effect of a transaction is eventually transmitted and delivered to all other replicas [40]. Figure 2b gives an example schedule allowed by eventual consistency.

Causal Consistency (CC) guarantees that the effect of a transaction is transmitted and delivered to every other replica after all of its dependencies (i.e., other transactions it depends on) have been delivered to that replica [2, 33]. Figure 2c gives an example schedule allowed by causal consistency.

To formalize these models, we first define a dependency function $\mathit{updDep} : \mathit{CM} \times \mathit{Events} \times \mathcal{H} \rightarrow \mathit{Events}$, where $\mathit{CM} = \{\mathit{SR}, \mathit{EC}, \mathit{CC}\}$ is the set of consistency models. Function updDep determines the dependency of an event by updating its vector clock based on the given system consistency model and history on which it is operating. Note that updDep is parameterized by the system consistency model. We also define a helper predicate $\mathit{isAllowed} : \mathit{CM} \times \mathit{Events} \times \mathcal{H} \rightarrow \mathbb{B}$ that determines if a given event is allowed to execute on its target replica under the specified consistency model, i.e., if all of events it depends on have already been executed. We leverage these definitions to formalize the three consistency models by specifying function updDep for each model.

To execute a transaction t, the *Serializability Consistency* model requires the synchronization of every other transaction that updates objects shared with t. Thus, transaction t depends on all other transactions in the system that update its objects. We define function $isAllowed$ for event $e = \langle msg, r, vc\rangle$ where $msg = \langle t, r'\rangle$ under this consistency model as follows. Suppose

$$depClock = \max_{\substack{e' \in relEvents(e,r'') \\ \forall r'' \in Rs}} ct(e'),\ obsClock = \max_{e' \in relEvents(e,r)} ct(e'),$$

where $depClock$ denotes the maximum visibility time of the related events and $obsClock$ denotes the time when the related events are observable. Then,

$$isAllowed(SR, e, \mathcal{H}) = \begin{cases} true & (t \in Txns \wedge obsClock = depClock) \vee \\ & (t \in Logs \wedge vc = obsClock) \\ false & \text{otherwise.} \end{cases}$$

Finally, the $updDep$ function for *Serializability Consistency* is defined as follows:

$$updDep(SR, \langle msg, r, vc\rangle, \mathcal{H}) = \begin{cases} \langle msg, r, depClock\rangle & isAllowed(SR, \langle msg, r, vc\rangle, s) \\ \langle msg, r, vc\rangle & \text{otherwise.} \end{cases}$$

In other words, a transaction should have seen all other transactions' effects that update shared objects in the system before starting.

In the *Eventual Consistency* model, there is no dependency between transactions, and messages can be delivered to a replica in any order. Thus, function $isAllowed$ returns $true$ for any event in the system

$$isAllowed(EC, \langle msg, r, vc\rangle, \mathcal{H}) = true,$$

and function $updDep$ is then defined as follows:

$$updDep(EC, \langle msg, r, vc\rangle, \mathcal{H}) = \langle msg, r, \{0\}^n\rangle.$$

In the *Causal Consistency* model, a transaction t depends on all transactions that update shared objects whose effects have been seen by t. We define function $isAllowed$ for event $e = \langle msg, r, vc\rangle$ where $msg = \langle t, r'\rangle$ under this consistency model as follows. Suppose $obsClock = \max_{e' \in relEvents(e,r)} ct(e')$, denotes the time when the related events are observable. Then,

$$isAllowed(CC, e, \mathcal{H}) = \begin{cases} true & vc \preceq obsClock \\ false & \text{otherwise.} \end{cases}$$

Finally, the updDep function for *Causal Consistency* is defined as follows:

$$\mathit{updDep}(CC, \langle \mathit{msg}, r, \mathit{vc}\rangle, \mathcal{H}) = \begin{cases} \langle \mathit{msg}, r, \mathit{obsClock}\rangle & \mathit{isAllowed}(CC, \langle \mathit{msg}, r, \mathit{vc}\rangle, \mathcal{H}) \\ \langle \mathit{msg}, r, \mathit{vc}\rangle & \text{otherwise.} \end{cases}$$

According to these definitions of $\mathit{isAllowed}$ and updDep for all three consistency models, an event e is always allowed to operate on a replica r under *Eventual Consistency*, is allowed under *Causal Consistency* if all its dependent transactions have already been delivered to r, and finally is allowed under *Serializability Consistency* if all transactions on its shared objects in the whole system have already been delivered to r.

3.2 Scheduler

In this section, we give a basic scheduler definition parameterized by a consistency model. A scheduler $M = \langle CM, D, \mathtt{empty}, \mathtt{give}, \mathtt{take}\rangle$ is a tuple consisting of a consistency model CM, a datatype $D = \langle DS \times \mathcal{H}\rangle$ of scheduler objects (where DS is a datatype for maintaining scheduling events and set $\mathcal{H}$ is history as defined in the previous section), a scheduler constructor $\mathtt{empty} \in D$, the function $\mathtt{give} : D \times \mathit{Events} \to D$ that receives posted events, and the function $\mathtt{take} : CM \times D \to \wp(D \times \mathit{Events})$ that determines which event at which replica operates next.

For the given consistency model cm, the scheduler M is deterministic if for all $m \in DS$, $\mathtt{take}(cm, \langle m, \mathcal{H}\rangle)$ has at most one element. It is non-blocking if all scheduled events are allowed, more formally if for all $e \in \mathit{Events}$ and $m, m' \in DS$:

$$\langle\langle m', \mathcal{H} \cup e\rangle, e\rangle \in \mathtt{take}(cm, \langle m, \mathcal{H}\rangle) \implies \mathit{isAllowed}(cm, e, \mathcal{H}).$$

That is, if an event is taken to operate on a replica, it must be allowed to do so.

Definition 1 (Bag Scheduler) *The multiset-based scheduler* `bag` *is defined on the multiset domain* D_{bag} *of events as*

$$\begin{aligned} \mathtt{empty}_{\mathtt{bag}} &:= \emptyset \\ \mathtt{give}_{\mathtt{bag}}(\langle m, \mathcal{H}\rangle, e) &:= \langle m \cup \{e\}, \mathcal{H}\rangle \\ \mathtt{take}_{\mathtt{bag}}(cm, \langle m, \mathcal{H}\rangle) &:= \{\langle\langle m\backslash\{e\}, \mathcal{H} \cup \{e\}\rangle, e\rangle \mid e \in m\}. \end{aligned}$$

According to this definition, $\mathtt{take}_{\mathtt{bag}}$ returns a set of allowed events and thus the bag scheduler is nondeterministic. Based on this basic scheduler definition, in the next section we propose two different scheduling strategies, namely random and delay-bounded [19].

4 Scheduling Strategies

In this section, we propose two scheduling strategies for weakly consistent programs. Later in Section 6 we empirically evaluate and compare the two strategies.

4.1 Random Scheduling

We define a random scheduler, which randomly exercises possible program schedules. When an event is posted, it is added to a bag of events. Then, the random scheduler randomly selects and dispatches one of the legal events in the bag. We formally define such a random scheduler as a tuple $M = \langle CM, D_{bag}, \texttt{empty}_{\texttt{bag}}, \texttt{give}_{\texttt{bag}}, \texttt{take}_{\texttt{bag}} \rangle$, and we call it the ***C****onsistency-****A****ware* ***R****andom* (CAR) scheduler. The scheduler proceeds if the current event either (1) completes its operation or (2) is not allowed.

Definition 2 (Bag-based CAR Scheduler) *Let BCAR be a bag-based scheduler defined as a tuple:*

$$\mathit{BCAR} = \langle CM, (\{\mathit{Events}\} \times \wp(\mathit{Events})), \langle \epsilon, \mathcal{H}_0 \rangle, \texttt{give}_{\texttt{bag}}, \texttt{take}_{\texttt{bag}} \rangle.$$

where functions $\texttt{give}_{\texttt{bag}}$ *and* $\texttt{take}_{\texttt{bag}}$ *are defined as shown in Algorithm 1.*

Let m, m' be two bags, where m maintains all events to be scheduled, and m' maintains all legal events with respect to the current history $\mathcal{H}$ and under the specified consistency model. Suppose output is a subset of $D \times \mathit{Events}$, and $e = \langle msg, r, vc \rangle$ where $msg = \langle t, r' \rangle$, such that t is in either $Txns$ or $Logs$. Function $\texttt{give}_{\texttt{bag}}$ takes a scheduler object $\langle m, \mathcal{H} \rangle$ and an event e as the input, and then it updates the scheduler to $\langle m \cup \{e\}, \mathcal{H} \rangle$. Function $\texttt{take}_{\texttt{bag}}$ takes the underlying consistency model cm and a scheduler object $\langle m, \mathcal{H} \rangle$ as the input. If either m is an empty bag or there is no legal event e in m for the specified history $\mathcal{H}$, no event is scheduled, i.e., $\texttt{take}_{\texttt{bag}}$ returns an empty set. Otherwise, all legal events in m with respect to cm and $\mathcal{H}$ are maintained in m'. Thereby, for every event $e = \langle \langle t, r' \rangle, r, vc \rangle$ in m', $\texttt{take}_{\texttt{bag}}$ does the following: (1) updates the dependency of e if $t \in Txns$, according to $\mathit{updDep}(cm, e, \mathcal{H})$ as defined in Section 3; (2) adds e to the history $\mathcal{H}$; (3) adds the tuple $\langle \langle m \backslash \{e\}, \mathcal{H} \rangle, e \rangle$ to the output set; and (4) returns the set output.

Algorithm 1 Functions for Bag-based CAR scheduler

```
$\mathcal{H}, m, m'$: bags of events
$output \subseteq D \times Events$
$msg = \langle t, r' \rangle$                          ▷ Where $t \in Txns \cup Logs$ and $r' \in Rs$
$e = \langle msg, r, vc \rangle$                       ▷ Where $r \in Rs$ and $vc \in \mathbb{Z}^n_{\geq 0}$

function give$_{\texttt{bag}}$($\langle m, \mathcal{H} \rangle, e$)
    $m \leftarrow m \cup \{e\}$
    return $\langle m, \mathcal{H} \rangle$
end function

function take$_{\texttt{bag}}$($cm, \langle m, \mathcal{H} \rangle$)
    if $m = \emptyset$ or $\forall e \in m : \neg isAllowed(cm, e, \mathcal{H})$ then
        return $\emptyset$
    end if
    for all $Event\ e \in m$ do
        if $isAllowed(cm, e, \mathcal{H})$ then
            $m' \leftarrow m' \cup \{e\}$
        end if
    end for
    for all $Event\ e \in m'$ do
        if $t \in Txns$ then
            $e := \langle msg, r, vc' \rangle \leftarrow updDep(cm, e, \mathcal{H})$
        end if
        $\mathcal{H} \leftarrow \mathcal{H} \cup \{e\}$
        $output \leftarrow output \cup \{\langle \langle m \backslash \{e\}, \mathcal{H} \rangle, e \rangle\}$
    end for
    return $output$
end function
```

4.2 Delay-bounded Scheduling

Delay-bounded scheduling as introduced by Emmi et al. [19] parameterizes a program search space by a deterministic scheduler and delay bound k. A k-delay bounded scheduler generates different schedules by delaying the execution of up to k events in the deterministic scheduler.

In this paper we propose a delay-bounded scheduler that is aware of the consistency model of the underlying data store. In so doing, to limit the nondeterminism in the default scheduler, we employ a deterministic scheduler, and explore a limited number of deviations from that deterministic schedule. We define such delaying scheduler as $M =$

$\big\langle CM, D, \texttt{empty}, \texttt{give}, \texttt{take}, \texttt{delay}\big\rangle$. The function $\texttt{delay} : D \times \mathit{Events} \rightarrow D$ allows the scheduler to postpone the execution of an event. When an event is posted, it is enqueued, and its execution could be postponed at the dispatch time. We call such a scheduler, augmented with delay function, the ***C**onsistency-**A**ware **D**elay-bounded* scheduler (CAD). The scheduler advances to the next event when the current event either (1) completes its operation, (2) is not allowed, or (3) is delayed. An execution is k-CAD when the number of delay operations in that execution is at most k.

Definition 3 (List-based CAD Scheduler) *Let LCAD be the list-based delaying scheduler defined as a tuple:*

$$LCAD = \big\langle CM, \mathit{Events}^{*} \times \mathit{Events}^{*} \times \mathbb{Z}_{\geq 0} \times \wp(\mathit{Events}), \langle \epsilon, \epsilon, 0, \mathcal{H}_0\rangle, \texttt{give}, \texttt{take}, \texttt{delay}\big\rangle.$$

where functions `delay`*,* `give`*, and* `take` *are defined as shown in Algorithm 2.*

Let $\mathcal{H}$ be a set of events, denoting the history of the system, and m_r and m_d be two lists, where m_r maintains the events to be scheduled and m_d delayed events. Also, let event $e = \langle msg, r, vc\rangle$, where $msg = \langle t, r'\rangle$. Function `delay` takes a scheduler object $\langle m_r, m_d, i, \mathcal{H}\rangle$ and an event e as the input. Then, it delays the execution of e by appending it to m_d and returns the scheduler object $\langle m_r, m_d \oplus_l, i, \mathcal{H}\rangle$, where l is the length of m_d, and $\oplus_l$ operator inserts e to m_d at the position l (i.e., at the end of m_d). Function `give` takes a scheduler object $\langle m_r, m_d, i, \mathcal{H}\rangle$ and an event $e = \langle\langle t, r'\rangle, r, vc\rangle$ as the input. If $t \in Txns$, it inserts t in m_r at the position i and increments i; otherwise, if $t \in \mathit{Logs}$, it appends t to m_r. In the end, it returns the scheduler with the updated m_r, m_d, and i. Function `take` accepts the underlying consistency model and a scheduler object as the input. If either both m_r and m_d are empty lists or there is no legal event in m_r with respect to the specified consistency model and the current history $\mathcal{H}$ of the system, then no event is scheduled and an empty set is returned. Otherwise, if all events in m_r have been either scheduled or delayed, the scheduler substitutes m_r with m_d and m_d with an empty list and also sets i to 1. Then, while $m_r[i]$ is not a legal event, it delays $m_r[i]$ and increments i. Considering $m_r[i] = \langle\langle t, r'\rangle, r, vc\rangle$ as a legal event, this function first updates the dependency of $m_r[i]$ using $updDep(cm, m_r[i], \mathcal{H})$, if $t \in Txns$. Then, it adds $m_r[i]$ to the history $\mathcal{H}$ and returns a set consisting of a single tuple of the updated scheduler object and $m_r[i]$.

5 Implementation

We implement the proposed schedule exploration strategies in a prototype stateless model checker for weakly consistent programs named COMMANDER [16]. COMMANDER allows

Algorithm 2 Functions for List-based CAD scheduler

```
$\mathcal{H}$: set of events
$m_r$: list of events
$m_d$: list of delayed events
$msg = \langle t, r' \rangle$                                  ▷ Where $t \in Txns \cup Logs$ and $r' \in Rs$
$e = m_r[i] = \langle msg, r, vc \rangle$                          ▷ Where $r \in Rs$ and $vc \in \mathbb{Z}^n_{\geq 0}$

**function** delay($\langle m_r, m_d, i, \mathcal{H} \rangle$, $e$)
    $l := length(m_d);\ m_d \leftarrow m_d \oplus_l e$          ▷ operator $\oplus_l$ inserts $e$ into $m_d$ at position $l$
    return $\langle m_r, m_d, i, \mathcal{H} \rangle$
**end function**

**function** give($\langle m_r, m_d, i, \mathcal{H} \rangle$, $e$)
    **if** $t \in Txns$ **then**
        $m_r \leftarrow m_r \oplus_i e;\ i \leftarrow i + 1$
    **else if** $t \in Logs$ **then**
        $l := length(m_r);\ m_r \leftarrow m_r \oplus_l e$
    **end if**
    return $\langle m_r, m_d, i, \mathcal{H} \rangle$
**end function**

**function** take($cm$, $\langle m_r, m_d, i, \mathcal{H} \rangle$)
    **if** $i > |m_r|$ **then**
        $m_r := m_d;\ m_d := \epsilon;\ i := 1$
    **end if**
    **if** $m_r = \epsilon$ or $\forall e \in m_r \oplus m_d : \neg isAllowed(cm, e, \mathcal{H})$ **then**
        return $\emptyset$
    **end if**
    **while** $\neg isAllowed(cm, m_r[i], \mathcal{H})$ **do**
        $\langle m_r, m_d, i, \mathcal{H} \rangle \leftarrow$ delay($\langle m_r, m_d, i, \mathcal{H} \rangle, m_r[i]$); $i \leftarrow i + 1$
    **end while**
    **if** $t \in Txns$ **then**
        $e := \langle msg, r, vc' \rangle \leftarrow updDep(cm, \langle msg, r, vc \rangle, \mathcal{H})$
    **end if**
    $\mathcal{H} \leftarrow \mathcal{H} \cup \{e\};\ i \leftarrow i + 1$
    return $(\langle m_r, m_d, i, \mathcal{H} \rangle, e)$
**end function**
```

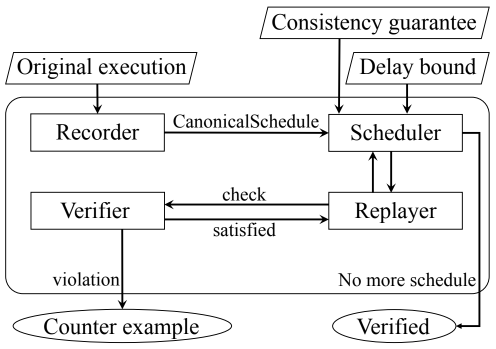


Figure 3: Overview of the COMMANDER architecture.

developers to systematically check if different weakly consistent schedules can result in failures of the provided program-specific invariants. We have implemented COMMANDER using Erlang [20], which is a commonly used programming language for distributed systems development. At the high level, COMMANDER first records an ordering of transactions applied on each replica, and their subsequent delivery to every remote replica in the system. Then, it systematically enumerates and replays different orderings of the recorded events using one of our consistency-aware schedule exploration strategies. As shown in Figure 3, COMMANDER consists of four components.

Recorder This module is responsible for recording the events that occur during the execution of the test scenario written by the developer. In order to record each event, we instrumented the Antidote data store where a transaction is committed or its corresponding transaction log is transmitted to another data center. The recorded sequence of events is then ordered in *CanonicalSchedule*, which is a deterministic canonical schedule where all transactions coming from clients are ordered based on their data center identifiers, while transaction logs are ordered based on their receiving data center identifiers.

Scheduler This module reorders the events in *CanonicalSchedule*, using the selected consistency-aware scheduling strategy, which is currently either CAR or CAD. As described in Section 4, *Scheduler* preserves the required dependencies during schedule generation. For every event e in *CanonicalSchedule*, if e is a delaying event, transaction log of an already delayed event, or an event whose causal dependency is not satisfied, the CAD scheduler delays e and picks the next event in *CanonicalSchedule*. Otherwise, it schedules e and updates the dependency relation in the generated schedule, which is then used in scheduling subsequent events. Finally, CAD schedules all previouly delayed events. On the other hand, CAR checks only for the causal dependency of e. If it is not satisfied, CAR skips e and picks another event randomly, and so forth until all events are scheduled.

Replayer This module exercises the events in the ordering that *Scheduler* provides. An event can either be a client transaction or transaction log transmission and delivery. For executing a transaction at a data center, *Replayer* behaves like a client and executes the corresponding transaction at the specified data center. For transmitting and delivering a transaction log, we instrumented Antidote to be able to hijack the corresponding communication between data centers.

Verifier This module checks for program-specific invariant violations after each scheduled event is replayed. If they are not violated, *Replayer* replays the next scheduled event, and so on. Otherwise, *Verifier* provides the current uncompleted schedule as a counterexample to the developer. An invariant is specified by implementing the provided interface by COMMANDER.

We target programs layered over the Antidote data store [3]. Note that in this particular data store, replicas are referred to as data centers. Antidote guarantees causal consistency. That is, depending on the timing of the concurrent events occurring at data centers, different causal dependencies can be formed between those events in an execution. Our consistency-aware scheduling can cover all possible causal dependencies that can be introduced by different executions of a program. Given a recorded schedule, our scheduling strategies generate different possible schedules either having the same causality dependency as the initially recorded schedule or producing new causality dependencies. For each generated schedule, we preserve the formed dependencies at all replicas and hence we do not introduce any spurious schedules not satisfying the required consistency requirements. We describe the implementation details behind each of these modules next. For more information about how to use COMMANDER see Appendix A.

We use Common Test framework [1], which supports automated testing of Erlang nodes, to launch Antidote data centers communicating through TCP/IP connection and an Erlang node as a testing node in COMMANDER. The testing node is connected to all data centers in the system, and we set it up to communicate with them to record and replay events in the order defined by the *Scheduler* module (refer to Figure 4).

6 Empirical Evaluation

We empirically evaluate our approach using three synthetic benchmarks and one realistic benchmark. Since Antidote is a new data store, there are no real world benchmark written for it to date, called *FMK Medical Application*. *FMK Medical Application* shares a medical profile among different health institutions. The invariant we check for this benchmark is that every prescription must be present in the corresponding patient's prescription list.

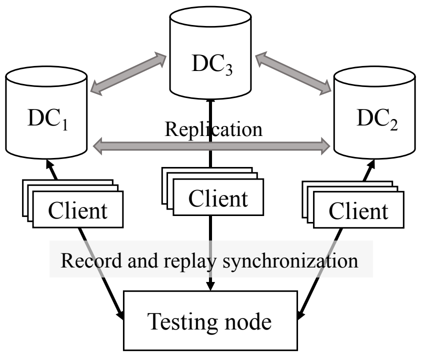


Figure 4: Testing environment for empirical evaluation.

In addition, we develop three benchmarks after the realistic use cases from the SyncFree project [39].

FMK Medical Application shares a medical profile among different health institutions. The invariant we check for this benchmark is that every prescription must be present in the corresponding patient's prescription list.

Virtual Wallet manages virtual economies of distributed computer game clients. Each virtual wallet has a balance that stores the available amount of money (see Section 2). Each client maintains a local state and issues credits and debits to it. Given the often very large number of clients in distributed games, virtual wallet has to be extremely scalable and ensure low latency; hence, the underlying consistency constraints typically have to be relaxed. This makes CRDTs appropriate data types to be used in such a program. The invariant we check for this benchmark is that the balance must not become negative.

Ad Counter implements a distributed counter. Advertising platforms keep track of impressions and clicks for ads in order to analyze advertising data. They use distributed counters to store the total number of ad views. Typically, there is an upper bound for an ad after which it should not be displayed anymore. Dynamic and fault-prone environments of such systems make CRDT counters a promising solution in scaling them to a large number of clients. The invariant we check for this benchmark is that the number of ad views must not exceed the upper bound.

Business to Business (B2B) Order plays the role of a traveling salesman for large manufacturers. Client store employees can see a catalogue of products and place orders from a mobile device using a B2B order program, concurrently. An employee might see an item is in stock and create an order, but the item goes out of stock while the device is synchronizing. Every store has a budget, and when the data is replicated over multiple data centers, two employees from the same store placing orders simultaneously may exceed the store budget. The invariants we check for this benchmark are that the store budget and number of items in stock must not become negative.

Table 1: Experimental results for the CAR scheduler with CC consistency model. Column *Txns* is the number of transactions in a benchmark; *Events* is the number of events; *Time* is runtime in min:sec; *#s* is the number of schedules explored by COMMANDER before it discovers a bug.

			min		**max**		**median**		**mean**	
Benchmark	**Txns**	**Events**	**#s**	**Time**	#s	**Time**	**#s**	**Time**	**#s**	**Time**
Virtual Wallet	30	90	1	1	37	42:06	8	9:10	11	12:31
Ad Counter	6	18	1	0:21	10	3:32	5	1:46	5	1:45
B2B Order	18	54	1	0:42	22	16:23	4	2:55	6	4:29
FMK	70	210	1	0:21	11	27:07	4	8:07	4	8:07

Table 2: Experimental results for the for-CAD and back-CAD variations of the CAD scheduler with CC consistency model. Column *Txns* is the number of transactions in a benchmark; *Events* is the number of events; *Time* is runtime in min:sec; *#s* is the number of schedules explored by COMMANDER before it discovers a bug. With k we denote the delay bound, and with * we denote runs where COMMANDER misses a bug because of an insufficient delay bound.

			for-CAD Scheduler						**back-CAD Scheduler**					
			k=0		**k=1**		**k=2**		**k=0**		**k=1**		**k=2**	
Benchmark	**Txns**	**Events**	**#s**	**Time**	**#s**	**Time**	**#s**	**Time**	**#s**	**Time**	**#s**	**Time**	**#s**	**Time**
Virtual Wallet	30	90	1*	1:14	5	5:30	182	208:45	1*	1:15	54	61:32	53	60:04
Ad Counter	6	18	1*	0:26	8	2:48	10	3:27	1*	0:25	7	2:22	7	2:21
B2B Order	18	54	1*	0:50	7	5:07	173	128:52	1*	0:51	31	23:07	30	22:21
FMK	70	210	1*	2:41	5	13:22	1	2:39	1*	2:39	>127	>328:00	>127	>328:00

We perform our experiments in a testing environment with a topology consisting of three data centers (DCs) as shown in Figure 4. The connections between data centers denote data replication, which is handled by Antidote. In addition, we create the testing node that hosts COMMANDER and communicates with every DC and client to record and replay events as described in Section 5. We set up multiple DCs connected using TCP/IP protocol on a single 4.00 GHz Intel Core i7 machine with 62 GB of memory. Our experiments could be run on a real distributed system as well. However to avoid the latency in communication between the testing node and every DC we opt to set up the testing environment locally.

To evaluate the effectiveness of our approach in detecting invariant violations and to empirically compare the different scheduling strategies, we seed a bug in each of our three synthetic benchmarks. Then, we use COMMANDER to discover the seeded bugs using the proposed CAR and CAD schedulers. These bugs are hidden in the canonical schedule and could be revealed through some causally consistent schedule, where an update delivery violates the invariant (refer to Figure 1 in Section 2). However, schedules with SR consistency miss those bugs. In addition, in our realistic benchmark, FMK, we find a real bug

Table 3: Experimental results for the for-CAD variation of the CAD scheduler with SR consistency model. Column *Txns* is the number of transactions in a benchmark; *Events* is the number of events; *Time* is runtime in min:sec; *#s* is the number of schedules explored by COMMANDER before it discovers a bug. With k we denote the delay bound, and with * we denote runs where COMMANDER misses a bug because of stringent consistency model.

			for-CAD Scheduler					
			k=0		**k=1**		**k=2**	
Benchmark	**Txns**	**Events**	**#s**	**Time**	**#s**	**Time**	**#s**	**Time**
Virtual Wallet	30	90	1*	1:14	5382*	6183:53	>1880*	>2160:00
Ad Counter	6	18	1*	0:26	246*	90:03	>6188*	>2160:00
B2B Order	18	54	1*	0:50	2055*	1594:59	>2895*	>2160:00
FMK	70	210	1*	2:41	>836*	>2160:00	>835*	>2160:00
FMK	6	18	1*	0:23	2	0:43	1	0:18

which CAD scheduler with delay bound of 0 misses it. The FMK system allows updating an entity, e.g., patient information, using its ID, even if that patient does not exist in a DC. Therefore, creating that patient later in a remote DC, after the update has been delivered, fails. The developers of the FMK system fixed this bug after we reported it.

Table 1 shows our experimental result for CAR scheduler. Given the inherent randomness of CAR scheduler, we run it 15 times with different random seeds on every benchmark, and we report min, max, median, and mean values for the numbers of explored schedules and runtimes. As the results show, CAR can generate buggy schedules almost immediately, but there is also a great variation in its effectiveness, which makes it brittle.

We implement two variations of our CAD scheduler. The first one is called for-CAD, and it delays events starting from the beginning of the canonical schedule. The second one is called back-CAD, and it delays events starting from the end of the canonical schedule. Table 2 shows our experimental for two CAD scheduler variations. As the results show, the for-CAD variation finds bugs faster than the back-CAD variation. Since the events coming from clients are being executed first according to our canonical schedule and events' dependencies are assigned only when they are coming from clients, for-CAD quickly alters the dependencies between events, which makes buggy schedules more likely to be caught. On the contrary, back-CAD starts delaying events from the end of our canonical schedule. Thus, it first delays events coming from other replicas, which do not allow their dependencies to be altered in order to preserve the underlying consistency model. Thereby, back-CAD first explores schedules with the same dependencies, which are less likely to be a buggy schedule. If we compare the for-CAD variation against the CAR scheduler, we notice that their effectiveness is comparable, while for-CAD had the advantage of being predictable.

To demonstrate that being aware of the underlying consistency model is important when trying to find bugs in weakly consistent systems, we evaluate our for-CAD delay scheduler with SR consistency model. As shown in Table 3, delay scheduler with SR consistency model misses the bugs in all synthetic benchmarks due to exercising limited and safe schedules. However, we expect our scheduler with SR consistency to find the existing real bug in FMK application, as it is not caused by an inappropriate choose of consistency for an entity, and it could be easily fixed by checking the existence of the corresponding entity before it is being updated. We set a timer to terminate COMMANDER after it runs for 36 hours. According to our experiments, our scheduler with SR consistency does not find that real bug in FMK application after 36 hours and exploring 836 schedules. To show the effectiveness of our scheduler with SR consistency in finding such bugs, which are not due to wrong consistency model, we simplify the test scenario for FMK application so that it has less number of transactions and consequently shorter schedules to exercise. Using the simplified test scenario, our scheduler finds the same real bug in FMK application immediately after exploring only a couple of schedules as shown in Table 3. That shows our scheduler is capable of finding not only consistency-related bugs, but also other implementation bugs. To reason about every single state, we sequentialize execution of every schedule. This sequentialization along with testing environment reset for every execution increases the analysis runtime.

7 Related Work

The most related approach to ours proposes a form of consistency called *explicit consistency* [7, 27]. Similarly to our work, users can specify the required consistency model, and unsafe operations are identified under concurrent executions using program-specific invariants. However, the consistency rules must be manually specified using additional program-specific invariants. Hence, the correctness of the approach relies on the correctness of the provided consistency rules. On the other hand, we guarantee the selected consistency model of the underlying data store and require users to specify only program-specific invariants. The approach based on explicit consistency has been implemented in CISE [34], which is a tool for statically reasoning about the correctness of weakly consistent programs.

When it comes to checking of weakly consistent programs, ECRacer [10] is a dynamic analysis tool that checks serializability of weakly consistent programs. It first records an execution of such a program and then performs an offline analysis to check for serializability. It does not take the dependency between user-initiated transactions into consideration, and therefore it can report false positives. Bouajjani et al. [9] propose a set of *bad patterns* to check causal consistency, causal memory, and causal convergence of an execution. If an execution contains a bad pattern with respect to a replicated data type, it is not consistent. They prove that checking those three consistency criteria of a single execution is an NP-hard problem, while it is undecidable for all executions.

In a recent effort, Zeller et al. propose a verification framework called Repliss [44], which includes a property-based testing engine [43] to check program specific invariants of programs built on top of weakly consistent data stores. The testing engine is based on the definition of the underlying data store schema, and it randomly exercises different executions of a given program. Finally, this technique provides a simple counter example by shrinking executions.

Lesani et al. propose Chapar [32], which includes a model checker targeting weakly consistent programs. Their work addresses an abstract model of programs in contrast to our work that performs execution-based model checking.

Kim et al. [30] propose a consistency oracle that simulates a distributed data store. Their consistency oracle could be integrated with any testing framework to verify storage system servers or client programs. The proposed consistency oracle requires total ordering that is determined by the oracle users. Consequently, it supports neither nor transactions which are being widely used in different data stores.

As a systematic concurrency testing approach, Walker and Runciman [41] propose a technique for distributed systems communicating through the messagepassing style, with no shared memory. Their approach considers a variety of network failures in the system while generating different schedules. That is, during a test execution, a message might be lost, reordered or duplicated. As opposed to our approach, this technique is applied to only strongly consistent system.

Deligiannis et al. [18] present a framework including P#, a language for programming highly reliable asynchronous such as distributed systems, and systematic concurrency testing techniques using DFS scheduling and randomized scheduling. They show that randomized scheduling is more effective than DFS scheduling. However, their work does not address weakly consistent programs, while our technique does.

In a recent work, Konnov et al. [31] extend the Byzantine Model Checker (ByMC) toolset [29] by adding the *short counterexample property* to it. ByMC takes as input a model of a distributed algorithm specified in an extended version of Promela, which is the modeling language of the SPIN [28, 38] model checker, and it checks the provided safety and liveness properties. Unlike our approach, ByMC allows for only strongly consistent systems to be analyzed.

There are also several projects that are focused on testing and model checking of Erlang programs *within single node*. QuickCheck [6] is a property-based testing tool with support for model-based exploration. It focuses on the input data nondeterminism by generating random inputs, whereas we address the problem of exploring concurrent nondeterministic interleavings. Its extension called PULSE [15] enables the exploration of interleavings, but it can only detect race conditions and supports just Erlang-level concurrency. The stateless model checker Concuerror [14] also explores interleaving nondeterminism to detect deadlocks, assertion violations, and abnormal termination. It explores possible executions systematically, and uses partial order reduction techniques to reduce the state space. However, it only explores interleavings between processes on a single Erlang node. The McErlang [24] model checker has similar capabilities since it only explores interleavings on a single node. Etomcrl [5] translates Erlang programs to mcrl (a process algebraic language) models, which are then checked by the CADP toolbox [25]. Hence, verification is done on the generated models, which can diverge from the original programs. None of these tools make assumptions of the consistency model of the underlying data store.

Verdi [42] is a framework for implementing verified distributed systems. Users first have to implement their system in Verdi, and the select the appropriate network semantics and fault model. Then, Verdi generates a verified system and compiles it to an executable to be deployed across the network. It is unclear how weak consistency would be supported within Verdi.

8 Conclusions and Future Work

In this paper, we adapt a model-checking-based verification approach to weakly consistent programs. Our proposed method is the first consistency-aware schedule exploration approach for geo-replicated distributed systems. We formalize two different scheduling strategies and implement them in our prototype model checker called COMMANDER. We employ COMMANDER to check and find bugs in programs layered over the Antidote data store, which uses CRDTs on causal consistency model. Our experiments show promising results of using COMMANDER for finding bugs on weakly consistent programs and provide an empirical comparison of the various schedule exploration strategies. Finally, the architecture of COMMANDER is modular, such that any scheduling strategy can be easily plugged into the tool.

As future work, we will add support for CRDTs with more stringent consistency guarantees than causal consistency, such as a bounded counter. In collaboration with the Antidote developers, we plan to apply COMMANDER on large real world programs to assess its scalability. We will explore partial order reduction methods to eliminate repetitious schedules resulting from delaying events that depend on other delayed events. Finally, we will explore extending COMMANDER so that it can check properties specified using Linear Temporal Logic (LTL).

9 Acknowledgments

We thank Annette Bieniusa and Tyler Crain for their guidance on using Antidote and Ryan Stutsman for his guidance on various consistency models. This research is supported in part by European FP7 project 609 551 SyncFree (2013–2016).

References

[1] Common Test Framework, `http://erlang.org/doc/apps/common_test/`

[2] Ahamad, M., Neiger, G., Burns, J.E., Kohli, P., Hutto, P.W.: Causal memory: Definitions, implementation, and programming. Distributed Computing 9(1) (1995)

[3] Akkoorath, D.D., Tomsic, A.Z., Bravo, M., Li, Z., Crain, T., Bieniusa, A., Preguiça, N., Shapiro, M.: Cure: Strong semantics meets high availability and low latency. In: ICDCS (2016)

[4] Antidote Reference Platform, `http://github.com/SyncFree/antidote`

[5] Arts, T., Benac Earle, C., Derrick, J.: Development of a verifed erlang program for resource locking. STTT (2004)

[6] Arts, T., Hughes, J., Johansson, J., Wiger, U.: Testing telecoms software with quviq quickcheck. In: ACM SIGPLAN Workshop on Erlang (2006)

[7] Balegas, V., Preguiça, N., Rodrigues, R., Duarte, S., Ferreira, C., Najafzadeh, M., Shapiro, M.: Putting consistency back into eventual consistency. In: EuroSys (2015)

[8] Baquero, C., Almeida, P.S., Shoker, A.: Making operation-based crdts operation-based. In: PaPEC (2014)

[9] Bouajjani, A., Enea, C., Guerraoui, R., Hamza, J.: On verifying causal consistency. In: POPL (2017)

[10] Brutschy, L., Dimitrov, D., Mller, P., Vechev, M.: Serializability for eventual consistency: Criterion, analysis, and applications. In: POPL (2017)

[11] Burckhardt, S.: Principles of Eventual Consistency (2014)

[12] Burckhardt, S., Gotsman, A., Yang, H., Zawirski, M.: Replicated data types: Specification, verification, optimality. In: POPL (2014)

[13] Cerone, A., Bernardi, G., Gotsman, A.: A framework for transactional consistency models with atomic visibility. In: CONCUR (2015)

[14] Christakis, M., Gotovos, A., Sagonas, K.: Systematic testing for detecting concurrency errors in erlang programs. In: ICST (2013)

[15] Claessen, K., Palka, M., Smallbone, N., Hughes, J., Svensson, H., Arts, T., Wiger, U.: Finding race conditions in erlang with quickcheck and pulse. In: ICFP (2009)

[16] Commander, `http://github.com/Maryam81609/commander`

[17] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's highly available key-value store. In: SOSP (2007)

[18] Deligiannis, P., Donaldson, A.F., Ketema, J., Lal, A., Thomson, P.: Asynchronous programming, analysis and testing with state machines. In: PLDI (2015)

[19] Emmi, M., Qadeer, S., Rakamarić, Z.: Delay-bounded scheduling. In: POPL (2011)

[20] Erlang Programing Language, `https://www.erlang.org/`

[21] EUnit - A Lightweight Unit Testing Framework for Erlang, `http://erlang.org/doc/apps/eunit/chapter.html`

[22] Fidge, C.J.: Timestamps in message-passing systems that preserve the partial ordering (1987)

[23] FMK Medical Application, `http://github.com/Maryam81609/FMKe`

[24] Fredlund, L.A., Svensson, H.: Mcerlang: A model checker for a distributed functional programming language. In: ICFP (2007)

[25] Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2011: A toolbox for the construction and analysis of distributed processes. STTT (2013)

[26] Gilbert, S., Lynch, N.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. ACM SIGACT News (2002)

[27] Gotsman, A., Yang, H., Ferreira, C., Najafzadeh, M., Shapiro, M.: 'cause I'm strong enough: Reasoning about consistency choices in distributed systems. In: POPL (2016)

[28] Holzmann, G.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley Professional (2003)

[29] John, A., Konnov, I., Schmid, U., Veith, H., Widder, J.: Towards modeling and model checking fault-tolerant distributed algorithms. In: SPIN (2013)

[30] Kim, B.H., Oh, S., Lie, D.: Consistency oracles: Towards an interactive and flexible consistency model specification. In: HotOS XVI (2017)

[31] Konnov, I., Lazić, M., Veith, H., Widder, J.: A short counterexample property for safety and liveness verification of fault-tolerant distributed algorithms. In: POPL

[32] Lesani, M., Bell, C.J., Chlipala, A.: Chapar: Certified causally consistent distributed key-value stores. In: POPL (2016)

[33] Lloyd, W., Freedman, M.J., Kaminsky, M., Andersen, D.G.: Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In: SOSP (2011)

[34] Najafzadeh, M., Gotsman, A., Yang, H., Ferreira, C., Shapiro, M.: The cise tool: Proving weakly-consistent applications correct. In: PaPoC (2016)

[35] Papadimitriou, C.H.: The serializability of concurrent database updates. JACM (1979)

[36] Riak - A Key-Value Store, `http://basho.com/products/riak-overview`

[37] Shapiro, M., Preguia, N.M., Baquero, C., Zawirski, M.: Conflict-free replicated data types. In: SSS (2011)

[38] SPIN - A Model Checker, `http://spinroot.com/spin/whatispin.html`

[39] SyncFree Project, `https://syncfree.lip6.fr`

[40] Terry, D.B., Theimer, M.M., Petersen, K., Demers, A.J., Spreitzer, M.J., Hauser, C.H.: Managing update conflicts in bayou, a weakly connected replicated storage system. In: SOSP (1995)

[41] Walker, M., Runciman, C.: Systematic testing for distributed systems. In: YDS (2016)

[42] Wilcox, J.R., Woos, D., Panchekha, P., Tatlock, Z., Wang, X., Ernst, M.D., Anderson, T.: Verdi: A framework for implementing and formally verifying distributed systems. In: PLDI (2015)

[43] Zeller, P.: Testing properties of weakly consistent programs with repliss. In: PaPoC (2017)

[44] Zeller, P., Poetzsch-Heffter, A.: Towards a Proof Framework for Information Systems with Weak Consistency (2016)

A COMMANDER Usage

A typical COMMANDER usage flow is as follow (refer to Figure 3):

1. A developer writes a test scenario in Erlang for a program under test by specifying the transactions executed at each data center.
2. The developer provides program invariants to be checked using the interface provided by COMMANDER and Erlang Unit Testing Framework (EUnit) [21]. In addition, the developer selects a consistency model (currently we support only causal consistency) and scheduling strategy; when CAD is selected, a delay bound has to be provided as well.
3. COMMANDER executes the test case while the *Recorder* module records the executed schedule of events and generates a canonical schedule we refer to as *CanonicalSchedule*.
4. The *Scheduler* module generates the set of all possible schedules for the recorded *CanonicalSchedule*, based on the chosen scheduling strategy and consistency model.
5. The *Replayer* module executes each generated schedule.
6. The *Verifier* module checks for invariant violations during the execution of each schedule.

A COMMANDER test case is an Erlang module, which implements `comm_test` behavior. This behavior defines the following three common functionalities, defined as callback functions, which every test case is compelled to implement:

`check` is the callback function, which implements the main body of the test case and always returns the Erlang atom `pass`.

`handle_event` provides a callback function, where every call to the SUT is performed inside it. To call an API of the SUT, developers are constrained to first call `event` function from `comm_test` behavior, which consequently will call the `handle_event` function in the test case.

`handle_object_invariant` provides developers with a callback function to specify the object invariant. Through this function, the developers are allowed to use any arbitrary Erlang code and EUnit assertions to verify the object invariant. The `objects` function from the `comm_test` behavior, is called inside the `check` function, and it takes a list of objects as an argument. Then, it calls `handle_object_invariant` function, which enables COMMANDER to check the invariant for the specified objects at each state.

Figure 5 shows how essential parts of a COMMANDER test case are implemented.

More detailed information about the `comm_test` behavior and how to use COMMANDER is available on the COMMANDER website [16]. In addition, all synthetic benchmarks [16] and the real world benchmark FMK [23] are publicly available on GitHub.

```
-module(wallet_comm).

-behavior(comm_test).

-include_lib("eunit/include/eunit.hrl").

%% Callbacks
-export([check/1, handle_event/1, handle_object_invariant/2]).

-define(INIT_VAL, 26500).

check(Config) ->
    ct:print("Entered␣check!"),
    [Node1, Node2, Node3] = proplists:get_value(sut_nodes, Config),

    Wallet = {wallet_key, riak_dt_pncounter, bucket},
    %% Specify invariant arguments
    comm_test:objects(?MODULE, [Wallet]),

    {_Re, CT} = comm_test:event(?MODULE, [2, Node1, ignore,
        [Wallet, ?INIT_VAL]]),

    [?assertEqual(rpc:call(Node, wallet, get_val, [Wallet, CT]),
        ?INIT_VAL) || Node <- [Node1, Node2, Node3]],
    pass.

handle_event([1, Node, ST, AppArgs]) ->
    [Wallet, N] = AppArgs,
    {R1, {_, CT1}} = rpc:call(Node, wallet, debit, [Wallet, N, ST]),
    {R1, CT1};

handle_event([2, Node, ST, AppArgs]) ->
    [Wallet, N] = AppArgs,
    {R2, {_, CT2}} = rpc:call(Node, wallet, credit, [Wallet, N, ST]),
    {R2, CT2};

handle_object_invariant(Node, [Wallet]) ->
    WalletVal =  rpc:call(Node, wallet, get_val, [Wallet, ignore]),
    ?assert(WalletVal >= 0),
    true.
```

Figure 5: A minimum COMMANDER test case for the virtual wallet program.