Ensuring referential integrity under causal consistency

Marc Shapiro, Annette Bieniusa, Peter Zeller, Gustavo Petri

Ensuring referential integrity under causal consistency*

Marc Shapiro
Sorbonne Université-LIP6, Paris, France
Inria, Paris, France

Annette Bieniusa
TU Kaiserslautern, Germany

Peter Zeller
TU Kaiserslautern, Germany

Gustavo Petri
IRIF, Paris Diderot – Paris 7, France

ABSTRACT

Referential integrity (RI) is an important correctness property of a shared, distributed object storage system. It is sometimes thought that enforcing RI requires a strong form of consistency. In this paper, we argue that causal consistency suffices to maintain RI. We support this argument with pseudocode for a *reference* CRDT data type that maintains RI under causal consistency. QuickCheck has not found any errors in the model.

ACM Reference Format:
Marc Shapiro, Annette Bieniusa, Peter Zeller, and Gustavo Petri. 2018. Ensuring referential integrity under causal consistency. In *PaPoC'18: 5th Workshop on Principles and Practice of Consistency for Distributed Data, April 23–26, 2018, Porto, Portugal.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3194261.3194262

1 REFERENCES AND REFERENTIAL INTEGRITY

Consider a shared store (memory) of objects, and a *reference* data type for linking objects in the store. Let's call a referencing object the *source* of the reference, and the referenced object its *target*. Intuitively, the *referential integrity* (RI) invariant states that if an application can reference some target, then the target "exists," in the sense that the application can access the target safely. A referenced object must not be deleted; conversely, when an object cannot be reached by any reference, deleting it is allowed.

We say that an object is *unreachable* if it *is not* the target of a reference, and *never will be* in the future (the latter clause is problematic under weak consistency). The RI property that we wish to achieve is the following:

- *Safety:* An object can be deleted only if it is unreachable.
- *Liveness:* Unreachability of an object will eventually be detected.

In a storage system where the application can delete objects explicitly, the programmer must be careful to preserve the RI invariant. This problem has been studied in the context of (concurrent) garbage collection for decades. Folklorically, it is often thought that enforcing RI requires synchronisation and strong consistency. In fact, previous work has stated otherwise [2, 4, 12]. The main purpose of this paper is to construct a reference data type demonstrating that causal consistency (with progress guarantees) suffices to ensure RI and to implement a safe deletion operation. We support this claim with pseudocode.

The solution that we sketch in this paper uses a form of reference counting (designed for distributed systems), called *reference listing* [4, 5, 10]. Objects with a non-empty reference list must not be deleted.

2 REFERENTIAL INTEGRITY AND CAUSAL CONSISTENCY

The safety property of RI is an instance of an implication invariant $P \implies Q$: *If a reference to an object exists, the object can accessed (has not been deallocated).* Elementary logic tells us that the sequential pattern of first making Q true, followed by making P true, will maintain such an invariant (the "backward pattern"). Similarly, making P false followed by making Q false (the "forward pattern") also works. The backward pattern translates to "first allocate the object, then assign reference to it," and the forward pattern to "first delete all references to object, then delete the object."

In a concurrent system with causal consistency [1], if two updates are ordered by happened-before [7], then all processes observe them in the same order. Therefore, we expect the same patterns to extend to such a system. Unfortunately, this does not suffice to maintain RI, because both patterns may be executing in parallel.

It is encouraging to remember that some datatypes can be engineered to support apparently-conflicting concurrent updates. For instance, a set can support concurrent insertion and removal of the same element, by making one operation "win" deterministically, the other one being superseded [11]. However, we cannot re-use this design directly since handling references also requires to handle the referred objects accordingly (including transitive reachability). Furthermore, while it is easy to ensure safety by never deleting anything, we also require liveness.

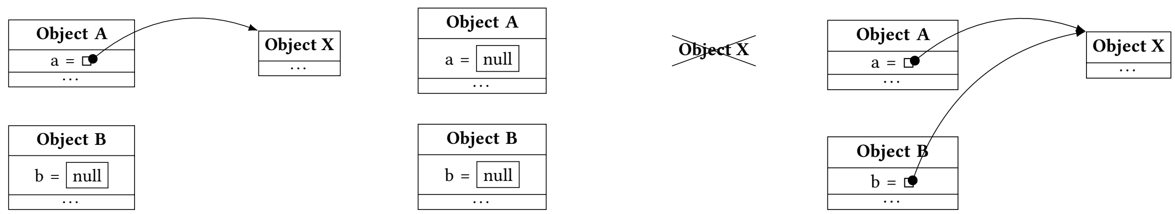
Note that causal consistency is only a safety property; it allows arbitrarily old versions to be observed. We need to add a progress guarantee assumption to ensure that our algorithm is live.

We assume that the objects of interest are accessed only via the reference datatype discussed herein. We do not address the more complex problem of objects that are accessible via some external means, e.g., through a well-known key, through a URL, or via a database query. These are called "root" objects (in garbage-collection parlance), which for our purposes are never deleted.

*Produces the permission block, and copyright information

PaPoC'18, April 23–26, 2018, Porto, Portugal

ACM ISBN 978-1-4503-5655-8/18/04…$15.00
https://doi.org/10.1145/3194261.3194262



(a) Initial state. (b) On Replica 1. (c) On Replica 2.

Figure 1: Concurrently creating references and deleting objects can lead to dangling references. How should the replicas be reconciled?

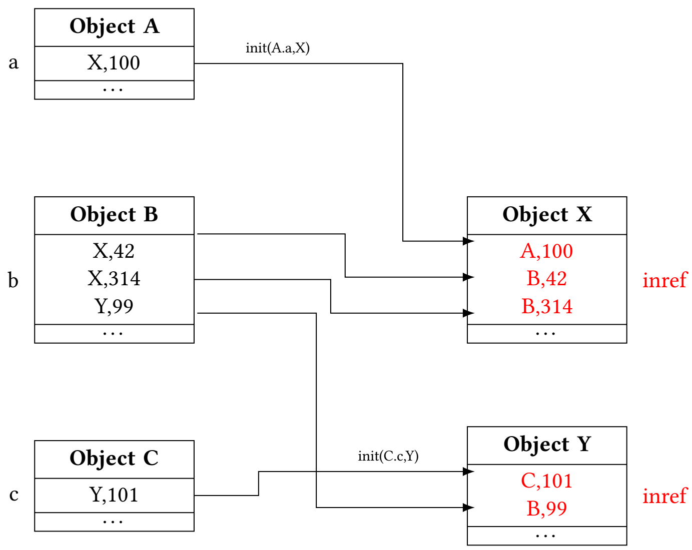


Figure 2: References with inref/outref after concurrently assigning `B.b` to `A.a` (twice) and `C.c`.

3 HIGH-LEVEL DESCRIPTION

We sketch in the following the reference handling protocol; a pseudocode description is given in Appendix A. A source object contains an instance of a data type called `outref` for every attribute that refers to another object. A (target) object is associated with exactly one `inref`. The inref identifies the currently-known sources pointing to this target. Creating a new reference initialises both the inref and an outref. The only application-level operations supported by inref are initialisation and testing whether deleting the target is allowed.

A (source) object contains any number of distinct outrefs. An outref supports the following application-level operations: *(i)* initialisation, *(ii)* assigning from another outref, *(iii)* assigning `null` (we assume that deleting an object first automatically nulls out all of its outrefs), and *(iv)* invocation, detailed shortly. To support concurrency, assigning an outref behaves much like a *Multi-Value Register*. Assignment overwrites its previous value; when concurrent assignments occur, the resulting reconciled value contains all the concurrently-assigned values. To simplify the semantics, we check that the right-hand side of the assignment is single-valued.

An outref can invoke its target, but this makes sense only if it has a single (non-null) target. If the outref contains multiple values, the invocation fails (the application can fix this by performing a new assignment).

Figure 2 illustrates three source objects `A, B, C`, each containing an attribute single outref named `a, b, c` respectively, and two target objects `X` and `Y`. The state illustrated might result from the following code snippet:

```
init (A.a, X); init (C.c, Y);
B.b := A.a || B.b := A.a || B.b := C.c;
```

Our algorithm design hinges on two principles that can be implemented assuming only causal consistency: (1) *before* an outref is assigned to a source object (in initialisation or assignment), we ensure that the corresponding inref has been added to the target object; importantly, causal consistency is enough to enforce this ordering of updates. (2) To delete a target, we require that no inref exists, nor will later be added, for this target. This property can be checked by well-known mechanisms which rely only on causal consistency and progress guarantees [14]. The combination of these properties is sufficient to ensure RI as defined in the introduction.

4 SYSTEM MODEL AND PSEUDOCODE

The pseudocode for references is listed in Appendix A. Some preliminary explanations are required.

References are layered above a lower-level unmanaged addressing mechanism (similar to a memory address used by the JVM), which we call *key*; a key uniquely identifies a single discrete (but possibly replicated) object.

Our system model is based on invocation split into two phases: the *generator* executes at a single replica, and generates a list of downstream messages that are eventually received at all replicas and executed by corresponding *effectors* [6, 8, 11]. At the source replica, the downstream messages are processed atomically with the generator. Other replicas may observe delays between the different downstream messages, but they will always receive them in the order specified by the generator. The generator may check preconditions (noted `precond`) against shared state; if any precondition is false, the operation fails. A generator may not have side effects on shared state. The effector must have the same effect at every replica, and therefore may not depend on testing shared state. We assume an operation's preconditions are *stable*, i.e., evaluating the precondition to true does not change under any concurrent operation [6].[1]

We assume causal consistency, i.e., one operation's effector is delivered (to some replica) only after the effectors of operations that are visible to it. We consider two alternatives for composed operations:

- Atomic: an operation is the atomic composition of all of its sub-operations. All the sub-generators (resp. sub-effectors) compose into a single atomic generator (resp. effector). This is somewhat similar to closed-nested transactions, without the isolation property.
- Pure causal: An effector updates a single object, but effectors can be chained, respecting the order defined in the code. This is somewhat similar to transaction chaining.

In both cases, if any precondition is false, the whole operation fails. Appendix A provides pseudocode for the latter option.[2]

The logic is relatively simple. On creating or copying a reference, avoid races by following the backward direction, first adding to the target, then to the source. On resetting (removing) a reference, follow the forward direction, first removing from the source, then from the target. We deal with concurrency by ensuring every reference has a unique identifier, and being careful of not losing any information. The details are tedious, but hopefully explained in the comments.

The `may_delete` operation merits a more detailed explanation. This operation returns true if and only if the `inref` argument is not reachable; however, in order to break circular reference patterns, the `last_refs` argument lists references to ignore. The `stably` notation in `may_delete` and in the third invariant means that the assertion is true, and that there are no concurrent mutations that could make it false.[3] Detecting `stably` boils down to detecting termination. Its implementation is well understood, requiring replicas to know about each other in order to exchange information on their progress [14]. Note that causal consistency is usually defined as a safety guarantee only [1, 13]. In order to ensure that a `stably` check eventually succeeds, we must add an assumption of progress, i.e., reads do not indefinitely return an old version.

Correctness. In order to validate the correctness of our CRDT references implementation, we formalized the system model and pseudocode implementation in Isabelle/HOL [9] and tested it with Haskell QuickCheck [3]. The corresponding code is available on GitHub.[4] The QuickCheck tests generate random executions and then check the first and the third invariant described in the pseudocode. To generate interesting random executions, we let each generated event depend on two randomly chosen previous events. Then, we randomly decide how many of their effector messages have been delivered to the new event. By doing this, it is likely that an event observes other events only partially, which is a common source of bugs. Indeed, we were able to discover some flaws in earlier drafts of the implementation and were able to fix them. For the updated implementation, our tests did not find a problem after 50 000 random executions.

[1] The pseudocode also makes use of `local_precond`, which does not need to be stable. Our use of the term "stable" in this section follows the terminology used in rely-guarantee logic.

[2] The "atomic" version is easier to read, but we prefer to minimise the assumptions. It is obtained from the pure-causal version by replacing the chained effectors by a single atomic one with the same text.

[3] This is called a "stable" property in the literature on distributed algorithms; we use "stably" to distinguish from the usage in Footnote 1.

[4] https://github.com/peterzeller/ref-crdt

ACKNOWLEDGMENTS

This research is supported in part by European H2020 project number 732 505 LightKone, and by the RainbowFS project of Agence Nationale de la Recherche, France, number ANR-16-CE25-0013-01.

REFERENCES

[1] Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. 1995. Causal memory: definitions, implementation, and programming. *Distributed Computing* 9, 1 (March 1995), 37–49. https://doi.org/10.1007/BF01784241

[2] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2014. Coordination Avoidance in Database Systems. *Proc. VLDB Endow.* 8, 3 (Nov. 2014), 185–196. https://doi.org/10.14778/2735508.2735509Int. Conf. on Very Large Data Bases (VLDB) 2015, Waikoloa, Hawai'i, USA.

[3] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP)*. https://doi.org/10.1145/351240.351266

[4] Paulo Ferreira and Marc Shapiro. 1994. Garbage Collection and DSM Consistency. In *Symp. on Op. Sys. Design and Implementation (OSDI)*. ACM, Monterey CA, USA, 229–241. http://www.usenix.org/publications/library/proceedings/osdi/ferr.html

[5] Paulo Ferreira and Marc Shapiro. 1996. Larchant: Persistence by Reachability in Distributed Shared Memory through Garbage Collection. In *Int. Conf. on Distributed Comp. Sys. (ICDCS)*. Hong Kong, 394–401. https://doi.org/10.1109/ICDCS.1996.507987

[6] Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 2016. 'Cause I'm Strong Enough: Reasoning about Consistency Choices in Distributed Systems. In *Symp. on Principles of Prog. Lang. (POPL)*. St. Petersburg, FL, USA, 371–384. https://doi.org/10.1145/2837614.2837625

[7] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (July 1978), 558–565. http://doi.acm.org/10.1145/359545.359563

[8] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. 2012. Making Geo-Replicated Systems Fast as Possible, Consistent when Necessary. In *Symp. on Op. Sys. Design and Implementation (OSDI)*. Hollywood, CA, USA, 265–278.

[9] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. 2002. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. LNCS, Vol. 2283. Springer.

[10] Marc Shapiro, Peter Dickman, and David Plainfossé. 1992. *SSP Chains: Robust, Distributed References Supporting Acyclic Garbage Collection.* Rapport de Recherche 1799. Institut National de la Recherche en Informatique et Automatique (Inria), Rocquencourt, France. http://lip6.fr/Marc.Shapiro/papers/SSPC_rr1799.pdf

[11] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-free Replicated Data Types. In *Int. Symp. on Stabilization, Safety, and Security of Dist. Sys. (SSS) (Lecture Notes in Comp. Sc.)*, Xavier Défago, Franck Petit, and V. Villain (Eds.), Vol. 6976. Springer-Verlag, Grenoble, France, 386–400. https://doi.org/10.1007/978-3-642-24550-3_29

[12] Marcin Skubiszewski and Patrick Valduriez. 1998. Using GC-Consistent Cuts for Concurrent Garbage Collection in O_2. *Networking and Information Systems* 1, 2-3 (1998), 213–230.

[13] Paolo Viotti and Marko Vukolić. 2016. Consistency in Non-Transactional Distributed Storage Systems. 49, 1 (jul 2016), 19:1–19:34. https://doi.org/10.1145/2926965

[14] Gene T. J. Wuu and Arthur J. Bernstein. 1984. Efficient solutions to the replicated log and dictionary problems. In *Symp. on Principles of Dist. Comp. (PODC)*. Vancouver, BC, Canada, 233–242. http://doi.acm.org/10.1145/800222.806750

A PSEUDOCODE

The following pseudocode describes the pure-causal version of references. The atomic version differs essentially by replacing the cascaded effectors with a single atomic one.

Block structure is indicated by indentation. Comments are preceded by the "%" character.

```
datatype outref of T
    % A reference to an object of type T containing an inref.

    % Key of embedding object. If object has multiple outrefs, assume
```

```
   % each one has a distinct object_key.
   object_key: write_once register of Key
   % routing information to referenced object
   dest_keys: MV_register of (outkey: Key, id: uid),
                                 initial (nullkey, nulluid)

datatype inref
   % key of embedding object
   object_key: write_once register of Key
   % set of reverse references
   rev_refs: 2P_set of (inkey: Key, id: uid),
                              initial emptyset
   % call "init" only once
   inuse: CRDT_flag, initial false

% if outref exists, inref exists
invariant
   forall r: outref of T
      (k,u) in r . dest_keys ==> (r . object_key, u) in k . inref . rev_refs

% correct type
invariant
   forall r: outref of T
      (k,u) in r . dest_keys ==>  k in T

% once an inref is unreachable, it remains unreachable
invariant
   forall i: inref
      % "stably" = true at all replicas and no concurrent updates in flight
      stably { i . rev_refs = emptyset }
         ==> henceforth { i . rev_refs = emptyset }

%% constructor; not part of API
_create_inref (k: Key, inref: inref)
   % the inref is embedded inside the object with key k
   inref . object_key := k

%% constructor; not part of API
_create_outref (k: Key of T, outref: outref of T)
   % the outref is embedded inside the object with key k
   outref . object_key := k

%% updates outref with new key value; not part of API
_outref_update(outTo: outref of T, new_key: Key)
    generator(outTo, new_key)
        let source_key = outTo . object_key
        let to_reset = outTo . dest_keys . getall ()
        let newuid =  new_uid()
        % explicit effector chaining
        if new_key != nullkey
           effector#1 (outTo, source_key, new_key, to_reset, newuid)
        else
           effector#2 (outTo, source_key, new_key, to_reset, newuid)

    effector#1 (outTo, source_key, new_key, to_reset, newuid)
        % first insert into new target
        new_key . inref . rev_refs . add ((source_key, newuid))
        effector#2 (outTo, source_key, new_key, to_reset, newuid)

    effector#2 (outTo, source_key, new_key, to_reset, newuid)
        % then assign source
        outTo . dest_keys := (new_key, newuid) % conc. assign possible
        forall (k, u) in to_reset
            % chain reset
            effector#3 (k, u, source_key)
    effector#3 (k, u, source_key)
        % finally, remove old reverse refs
        k . inref . rev_refs . remove ((source_key, u))
```

```
% create a reference from outref to inref
init (outref: outref of T, inref: inref)
    generator (outref, inref)
        precond ! inref . inuse          % call init only once
        effector#1(inref)
        % run _outref_update effectors after effector#1
        _outref_update(outref, inref.object_key)
    effector#1(inref)
        inref.inuse := true

% Remove an inref.
% Deleting the object that embeds inref calls this; therefore, the
% outer delete will fail if there are any remaining references.
reset (inref: inref)
    generator
        % Non-reachability is monotonic
        precond inref . may_delete()
    effector
        skip

% Remove an outgoing reference
% Deleting the object that embeds the outref calls this.
reset (outref: outref of T)
    generator (outref)
        % same as assigning nullkey:
        _outref_update(outref, nullkey)

:= (outTo: outref of T, outVal: outref of T)
   assign (outTo, outVal)

% outTo := outVal
%
% Copy outVal into outTo; reset outTo; in that order.  Either may be
% initially null.  No-op if outVal target already in outTo.
%
% Concurrent "assign"s to outref store multiple values inside MV_register.
% The user should resolve by a subsequent "assign"
%
assign (outTo: outref of T, outVal: outref of T)
    generator (outTo, outVal)
        % simplification: ensure outVal has no more than one target
        % local check, not necessarily stable
        local_precond outVal . dest_keys . count() = 1
        let (new_key, _) = outVal . dest_keys . get1 ()
        _outref_assign(outTo, newKey)

% Use a reference to call the target object
deref (outref: outref of T, invocation: invocation of T)
    generator (outref, invocation)
        % local checks, not necessarily stable
        local_precond outref . dest_keys . count() = 1
        local_precond outref . dest_keys . get1() != (nullkey, _)
        let (key1, _) = outref . dest_keys.get1()
    effector(key1, invocation)
        invoke (key1, invocation)

% Is target object reliably not referenced?
% To be tested in a generator. last_refs: if only these exist we are
% still OK, because the effector will to reset them shortly.
may_delete (inref: inref,
            last_refs: set of outref, default emptyset
           ): boolean
    generator (inref, last_refs)
        % check that the only remaining rev_refs are those in last_refs
        % (none by default)
        last_keypairs: set of (Antidote_key, uid)
                       = { fold (last_refs,
                                lambda (r) cons (r.object_outref, "_")) }
        return stably inref . rev_refs = last_keypairs
```

```
effector ()
    skip
```