Ensuring referential integrity under causal consistency
Marc Shapiro, Annette Bieniusa, Peter Zeller, Gustavo Petri

To cite this version:
Marc Shapiro, Annette Bieniusa, Peter Zeller, Gustavo Petri. Ensuring referential integrity under causal consistency. PaPoC 2018 - 5th Workshop on Principles and Practice of Consistency for Distributed Data, Apr 2018, Porto, Portugal. hal-01727207

HAL Id: hal-01727207
https://inria.hal.science/hal-01727207
Submitted on 9 Mar 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Ensuring referential integrity under causal consistency

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

Peter Zeller
TU Kaiserslautern, Germany

Annette Bieniusa
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:

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

\[ P \implies Q: \text{If a reference to an object exists, the object can accessed (has not been deallocated).} \]

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.
Object A \(a = \circ\) . . .

Object B \(b = \text{null}\) . . .

Object X . . .

(a) Initial state.

Object A \(a = \circ\) . . .

Object B \(b = \circ\) . . .

Object X . . .

(b) On Replica 1.

Object A \(a = \circ\) . . .

Object B \(b = \circ\) . . .

Object X . . .

(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 \(\text{outref}\) for every attribute that refers to another object. A (target) object is associated with exactly one \(\text{inref}\). The \(\text{inref}\) identifies the currently-known sources pointing to this target. Creating a new reference initialises both the \(\text{inref}\) and an \(\text{outref}\). The only application-level operations supported by \(\text{inref}\) are initialisation and testing whether deleting the target is allowed.

An \(\text{outref}\) can invoke its target, but this makes sense only if it has a single (non-null) target. If the \(\text{outref}\) 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:

\[
\text{init}(A.a, X); \quad \text{init}(C.c, Y); \\
B.b := A.a \quad \big|\big| \quad B.b := A.a \quad \big|\big| \quad B.b := C.c; \\
\]

Our algorithm design hinges on two principles that can be implemented assuming only causal consistency: (1) before an \(\text{outref}\) is assigned to a source object (in initialisation or assignment), we ensure that the corresponding \(\text{inref}\) 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 \(\text{inref}\) 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 \textit{key}; a key uniquely identifies a single discrete (but possibly replicated) object.

Our system model is based on invocation split into two phases: the \textit{generator} executes at a single replica, and generates a list of downstream messages that are eventually received at all replicas and executed by corresponding \textit{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 \textit{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 \textit{stable}, i.e., evaluating
the precondition to true does not change under any concurrent operation \[6\].

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.

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_ref` 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 GitHub \[3\]. 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.

**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**


\[5\] 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, Hong Kong, 394–401. https://doi.org/10.1109/ICDCS.1996.507987


**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.

```plaintext
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 info 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(outref, 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 newuid != nullkey
        effector1 #1(outTo, source_key, new_key, to_reset, newuid)
    else
        effector2 #1(outTo, source_key, new_key, to_reset, newuid)

effector1 #1(outTo, source_key, new_key, to_reset, newuid)
% first insert into new target
    new_key . inref . rev_refs . add ((source_key, newuid))
    effector1 #2(outTo, source_key, new_key, to_reset, newuid)

effector2 #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
    effector3 #k (k, u, source_key)

% 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(outref, 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 newuid != nullkey
        effector1 #1(outTo, source_key, new_key, to_reset, newuid)
    else
        effector2 #1(outTo, source_key, new_key, to_reset, newuid)

effector1 #1(outTo, source_key, new_key, to_reset, newuid)
% first insert into new target
    new_key . inref . rev_refs . add ((source_key, newuid))
    effector1 #2(outTo, source_key, new_key, to_reset, newuid)

effector2 #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
    effector3 #k (k, u, source_key)

% create a reference from outref to inref
init (outref: outref of T, inref: inref)
    generator (outref, inref)
    precondition ! inref . inuse % call init only once
    effector1 (inref)
    % run _outref_update effectors after effector1
    _outref_update(outref, inref.object_key)
    effector1 (inref)

% 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
    precondition inref . may_delete()
    effector

% 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)

% 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)
    := (outTo: outref of T, outVal: outref of T)
    assign (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)
% % 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_keys: set of (Antidote_key, uid)
%       = ( fold (last.refs, lambda (r) cons (r.object_outref, ".-")) )
% return stably inref . rev.refs = last_keys: set of outref, default emptyset
effector ()
skip