

# Crash-Corruption Disentanglement in Log-Based Storage

Ramnatthan Alagappan, Aishwarya Ganesan, Eric Lee<sup>†</sup>, Aws Albarghouthi,  
Vijay Chidambaram<sup>†</sup>, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau  
University of Wisconsin – Madison    <sup>†</sup> University of Texas at Austin

## 1 Introduction

In storage systems, a checksum mismatch for a piece of data could occur due to two distinct underlying reasons: a *storage corruption* or a *system crash*. In the first case, a piece of data could be safely persisted but could be *corrupted* at some later point (e.g., due to a faulty disk). In the second, the system could have *crashed* (e.g., due to a power failure) in the middle of an update, resulting in partially written data, causing a checksum mismatch.

It is critical for a storage system to distinguish the two conditions to perform recovery actions correctly. If crashes and corruptions are not treated separately, serious consequences such as data loss may arise. For example, many log-based storage systems conflate crashes and corruptions, always treating mismatches as crashes, resulting in serious consequences such as data loss [1].

Crash-corruption disentanglement is a fundamental problem applicable to any log-based storage system. Consequently, it also arises in the context of *replicated logs*, a basic abstraction to implement replicated state machines (RSM). In RSM systems, the commands to the state machine are stored as entries in a log, replicated across many nodes.

CTRL [1] intends to protect RSM systems against storage corruptions and recover corrupted entries from redundant copies. CTRL realizes that if a mismatch were really due to a crash, it is safe to discard the partially written data; it is safe because the node would not have acknowledged to any external entity that it has written the entry. However, if an entry is corrupted, the entry cannot be simply discarded since it could be globally committed; therefore, it has to be recovered from copies on other nodes. Furthermore, if a mismatch can be correctly attributed to a crash, the faulty entry can be quickly discarded locally, avoiding the distributed recovery.

As a prerequisite to achieve the above goals, the storage layer of CTRL needs to correctly distinguish crashes and corruptions in the replicated log.

Many storage systems use a commit record to mark

the successful completion of a data update. Similarly, CTRL also writes a persist record after writing an entry to the log. For now, assume that an entry is ordered before its persist record. During recovery, if the persist record is *not present* and a checksum mismatch occurs for an entry, CTRL’s recovery code can determine that the mismatch is due to an interrupted update (i.e., a crash). Conversely, if the persist record is *present* and a checksum mismatch occurs, the recovery code can conclude that it is a storage corruption.

The above disentanglement logic works correctly when an entry is explicitly ordered before its persist record using a *fsync* system call. However, such additional *fsync* calls could affect the log-update performance significantly. For this reason, CTRL does *not* explicitly order an entry before its persist record.

Without explicitly ordering a data item before its persist record, CTRL can still distinguish crashes from corruptions in most cases. However, if a checksum mismatch happens for the *last entry* in the log and if its persist record is present, CTRL cannot determine whether the mismatch is due to storage corruption or a system crash. The inability to disentangle the last entry when its persist record is present is not specific to CTRL, but rather a fundamental limitation in any log-based system. The following section presents the proof of this claim.

## 2 Impossibility of Last-Entry Disentanglement without Ordering

**Identifiers.** In CTRL, the persist record of a log entry also acts as its *identifier*. The identifier (or equivalently the persist record) of a log entry contains vital information about that entry; this information helps CTRL’s distributed protocol to recover corrupted entries from copies on other nodes. Thus, in the remainder of the discussion, we will use the term *identifiers* instead of *persist records*.

**Log state.** We model the log  $L$  as two disjoint lists, one list  $L_e$  that stores entries and one list  $L_{id}$  that stores identifiers.

**Instructions.** We assume we only have two kinds of instructions:

- $write(v)$ , which updates  $L_e$  or  $L_{id}$  (depending on if  $v$  is an entry or identifier).
- $fsync()$  commits all previous writes.

**Sequences.** A *disentangled sequence* of transactions  $\sigma = t_1, \dots, t_n$ , where  $n > 1$  is one where each  $t_i$  is a subsequence of three instructions:  $a_i^1, a_i^2, a_i^3$ , where:

- $a_i^1$  is of the form  $write(e_i)$ .
- $a_i^2$  is of the form  $write(id_i)$ .
- $a_i^3$  is of the form  $fsync()$ .

where  $e_i$  is the entry to be written and  $id_i$  is its respective identifier. For simplicity, we assume a single log.

**Log appends.** Suppose we are given a disentangled sequence  $\sigma = t_1, \dots, t_n$ . We use  $L^I$  to denote the initial state of the log. We use  $\sigma L^I$  to denote the state of the log after executing the sequence  $\sigma$  beginning from state  $L^I$ .

**Corruption and Crash.** We distinguish between two *bad* events: corruptions  $co$  and crashes  $cr$ .

- A corruption  $co_i$  changes element  $e_i$  in  $L_e$  to some new  $e'_i$  where  $e'_i \neq e_i$ .
- We assume a crash  $cr_i$  can only happen between  $a_i^2$  and  $a_i^3$ , i.e., right before the  $fsync$ , for a sequence  $t_1, \dots, t_n$ , as defined above.

Given sequence  $\sigma$ , we use  $\sigma_{cr_i}$  to denote  $\sigma$  with a crash in  $s_i$ . Given  $\sigma$ , we use  $\sigma_{co_i}$  to denote  $\sigma$  with a corruption event  $co_i$  appended at the end.

**Theorem 1 (Disentanglement).** *Suppose we are given the disentangled sequence  $\sigma$  and log  $L$ .*

- **Case 1:** Let  $L^1 = \sigma_{cr_n} L^I$ , and let  $L^2 = \sigma_{co_n} L^I$ . Suppose we are provided  $L^I$ ,  $\sigma$ , and one of the logs  $L^1$  and  $L^2$ . We cannot detect whether  $\sigma_{cr_n}$  or  $\sigma_{co_n}$  is the one that executed resulting in  $L^1$  or  $L^2$ .
- **Case 2:** Let  $L^{co_i} = \sigma_{co_i} L^I$ , where  $i \in [1, n]$ . Provided  $L^I, \sigma$ , and  $L^{co_i}$ , we can conclude that  $\sigma_{cr_j}$  did not execute, where  $j \in [1, n]$ .

*Proof.* First, we note that by being able to detect whether a crash or corruption happened, we mean

that there exists a deterministic algorithm that will return whether a crash or corruption happened.

**Case 1:** We prove the first case with a simple construction. Let  $\sigma = s_1$ , where

$$s_1 = write(e_1), write(id_1), fsync()$$

Let  $L^I$  be the empty log. Let  $L^1 = \sigma_{cr_1} L^I$  and  $L^2 = \sigma_{co_1} L^I$ .

Assume that when the crash  $cr_1$  happened, only a strict subset of  $e_1$  was written in addition to  $id_1$ . Let the strict subset of  $e_1$  that was written be  $e'_1$ . The above condition can arise because  $write(e_1)$  need not be atomic and writes can be reordered by the underlying file system on a crash. Now, assume that the corruption  $co_1$  turns  $e_1$  to  $e'_1$ .

We can now prove the first case by contradiction: Suppose there is an algorithm  $M$  that can take (i) the initial state of the log, (ii) the current state of the log, and (iii) the sequence of transactions  $\sigma$  that lead to the current state (minus  $co$  and  $cr$  events), and deterministically returns whether a crash or corruption happened. In the above example,  $L^1 = L^2$  by construction. So,  $M(L^I, L^1, \sigma) = M(L^I, L^2, \sigma)$ . Therefore, no such  $M$  exists.

**Case 2:** Fix  $i, j$  as in theorem statement. Let  $L^{cr_j} = \sigma_{cr_j} L^I$ . Assume  $L^{cr_j} = L^{co_i}$ . If  $j \neq i$ , then entry  $e_i$  cannot be affected by the crash, and therefore the  $L^{cr_j} \neq L^{co_i}$ . If  $j = i$ , since  $i < n$ , then  $e_i$  is fixed by recovery. Therefore,  $L^{cr_j} \neq L^{co_i}$  ■

## References

- [1] Ramnatthan Alagappan, Aishwarya Ganesan, Eric Lee, Aws Albarghouthi, Vijay Chidambaram, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Protocol-Aware Recovery for Consensus-Based Storage. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST '18)*, Oakland, CA, February 2018.