The IMAP CmRDT
Tim Jungnickel, Lennart Oldenburg, Matthias Loibl
February 23, 2021

Abstract

We provide our Isabelle/HOL formalization of a Conflict-free Replicated Data Type for Internet Message Access Protocol commands. To this end, we show that Strong Eventual Consistency (SEC) is guaranteed by proving the commutativity of concurrent operations. We base our formalization on the recently proposed \textit{"framework for establishing Strong Eventual Consistency for Conflict-free Replicated Datatypes"} (AFP.CRDT) by Gomes et al. Hence, we provide an additional example of how the recently proposed framework can be used to design and prove CRDTs.

Contents

1 Preface 1
   1.1 The IMAP CmRDT .................................................. 2
   1.2 Proof Guide .................................................. 2

2 IMAP-CRDT Definitions 3

3 Commutativity of IMAP Commands 5

4 Proof Helpers 6

5 Independence of IMAP Commands 10

6 Convergence of the IMAP-CRDT 13

1 Preface

A Conflict-free Replicated Data Type (CRDT) [5] ensures convergence of replicas without requiring a central coordination server or even a distributed coordination system based on consensus or locking. Despite the fact that Shapiro et al. provide a comprehensive collection of definitions for the most useful data types such as registers, sets, and lists [4], we observe that the use of CRDTs in standard IT services is rather uncommon. Therefore, we use the Internet Message Access Protocol (IMAP)—the de-facto standard protocol to retrieve and manipulate mail messages on an email server—as an example to show the feasibility of using CRDTs for replicating state of a standard IT service to achieve planetary scale.

Designing a \textit{correct} CRDT is a challenging task. A CmRDT, the operation-based variant of a CRDT, requires all operations to commute. To this end, Gomes et al. recently published a CmRDT verification framework [1] in Isabelle/HOL.

In our most recent work [3], we presented \textit{pluto}, our research prototype of a planetary-scale IMAP service. To achieve the claimed planet-scale, we designed a CmRDT that provides...
multi-leader replication of mailboxes without the need of synchronous operations. In order to ensure the correctness of our proposed IMAP CmRDT, we implemented it in the verification framework proposed by Gomes et al.

In this work, we present our Isabelle/HOL proof of the necessary properties and show that our CmRDT indeed guarantees Strong Eventual Consistency (SEC). We contribute not only the certainty that our CmRDT design is correct, but also provide one more example of how the verification framework can be used to prove the correctness of a CRDT.

1.1 The IMAP CmRDT

In the rest of this work, we show how we modeled our IMAP CmRDT in Isabelle/HOL. We start by presenting the original IMAP CmRDT, followed by the implementation details of the Isabelle/HOL formalization. The presentation of our CmRDT in Spec. 1 is based on the syntax introduced in [4]. We highly recommend reading the foundational work by Shapiro et al. prior to following our proof documentation.

In essence, the IMAP CmRDT represents the state of a mailbox, containing folders (of type \( N \)) and messages (of type \( M \)). Moreover, we introduce metadata in form of tags (of type \( ID \)). All modeling details and a more detailed description of the CmRDT are provided in the original paper [3].

The only notable difference between the presented specification and our Isabelle/HOL formalization is, that we no longer distinguish between sets \( ID \) and \( M \) and that the generated tags of \textit{create} and \textit{expunge} are handled explicitly. This makes the formalization slightly easier, because less type variables are introduced. The concrete definition can be found in the \textit{IMAP-CRDT Definitions} section of the \texttt{IMAP-def.thy} file.

1.2 Proof Guide

\textit{Hint:} In our proof, we build on top of the definitions given by Gomes et al. in [2]. We strongly recommend to read their paper first before following our proof. In fact, in our formalization we reuse the \textit{locales} of the proposed framework and therefore this work cannot be compiled without the reference to [1].

Operation-based CRDTs require all concurrent operations to commute in order to ensure convergence. Therefore, we begin our verification by proving the commutativity of every combination of possible concurrent operations. Initially, we used \texttt{nitpick} to identify corner cases in our implementation. We prove the commutativity in Section 3 of the \texttt{IMAP-proof-commute.thy} file. The critical conditions to satisfy in order to commute, can be summarized as follows:

- The tags of a \textit{create} and \textit{expunge} operation or the messages of an \textit{append} and \textit{store} operation are never in the removed-set of a concurrent \textit{delete} operation.

- The message of an \textit{append} operation is never the message that is deleted by a concurrent \textit{store} or \textit{expunge} operation.

- The message inserted by a \textit{store} operation is never the message that is deleted by a concurrent \textit{store} or \textit{expunge} operation.

The identified conditions obviously hold in regular traces of our system, because an item that has been inserted by one operation cannot be deleted by a concurrent operation. It simply cannot be present at the time of the initiation of the concurrent operation.

Next, we show that the identified conditions actually hold for all concurrent operations. Because all tags and all inserted messages are globally unique, it can easily be shown that
Specification 1 The IMAP CmRDT

1: payload map $u : N \to \mathcal{P}(\mathcal{I}) \times \mathcal{P}(\mathcal{M})$ \[\triangleright \{\text{foldername } f \mapsto (\{\text{tag } t\}, \{\text{msg } m\}), \ldots\}\]

3: update create (foldername $f$)

4: atSource

5: let $\alpha = \text{unique}()$

6: downstream ($f$, $\alpha$)

7: $u(f) \mapsto (u(f)_1 \cup \{\alpha\}, u(f)_2)$

8: update delete (foldername $f$)

9: atSource ($f$)

10: let $R_1 = u(f)_1$

11: let $R_2 = u(f)_2$

12: downstream ($f$, $R_1$, $R_2$)

13: $u(f) \mapsto (u(f)_1 \setminus R_1, u(f)_2 \setminus R_2)$

14: update append (foldername $f$, message $m$)

15: atSource ($m$)

16: pre $m$ is globally unique

17: downstream ($f$, $m$)

18: $u(f) \mapsto (u(f)_1, u(f)_2 \cup \{m\})$

19: update expunge (foldername $f$, message $m$)

20: atSource ($f$, $m$)

21: pre $m \in u(f)_2$

22: let $\alpha = \text{unique}()$

23: downstream ($f$, $m$, $\alpha$)

24: $u(f) \mapsto (u(f)_1 \cup \{\alpha\}, u(f)_2 \setminus \{m\})$

25: update store (foldername $f$, message $m_{\text{old}}$, message $m_{\text{new}}$)

26: atSource ($f$, $m_{\text{old}}$, $m_{\text{new}}$)

27: pre $m_{\text{old}} \in u(f)_2$

28: pre $m_{\text{new}}$ is globally unique

29: downstream ($f$, $m_{\text{old}}$, $m_{\text{new}}$)

30: $u(f) \mapsto (u(f)_1, (u(f)_2 \setminus \{m_{\text{old}}\}) \cup \{m_{\text{new}}\})$

all conditions are satisfied. In Isabelle/HOL, showing this fact takes some effort. Fortunately, we were able to reuse parts of the Isabelle/HOL implementation of the OR-Set proof in [1]. The Isabelle/HOL proofs for the critical conditions are encapsulated in the IMAP-proof-independent.thy file.

With the introduced lemmas, we prove the final theorem that states that convergence is guaranteed. Due to all operations being commutative in case the critical conditions are satisfied and the critical conditions indeed are holding for all concurrent updates, all concurrent operations commute. The Isabelle/HOL proof is contained in the IMAP-proof.thy file.

2 IMAP-CRDT Definitions

We begin by defining the operations on a mailbox state. In addition to the interpretation of the operations, we define valid behaviours for the operations as assumptions for the network. We use the network_with_constrained_ops locale from the framework.
theory
  IMAP_def
imports
  CRDT.Network
begin

datatype ('id, 'a) operation =
  Create 'id 'a |
  Delete 'id set 'a |
  Append 'id 'a |
  Expunge 'a 'id 'id |
  Store 'a 'id 'id

type-synonym ('id, 'a) state = 'a ⇒ ('id set × 'id set)

definition op-elem :: ('id, 'a) operation ⇒ 'a where
  op-elem oper ≡ case oper of
    Create i e ⇒ e |
    Delete is e ⇒ e |
    Append i e ⇒ e |
    Expunge e mo i ⇒ e |
    Store e mo i ⇒ e

definition interpret-op :: ('id, 'a) operation ⇒ ('id, 'a) state ⇒ ('id, 'a) state
  where
    interpret-op oper state ≡
      let metadata = fst (state (op-elem oper));
      files = snd (state (op-elem oper));
      after = case oper of
        Create i e ⇒ (metadata ∪ {i}, files) |
        Delete is e ⇒ (metadata, files ∪ {is}, files − is) |
        Append i e ⇒ (metadata, files ∪ {i}) |
        Expunge e mo i ⇒ (metadata ∪ {i}, files − {mo}) |
        Store e mo i ⇒ (metadata, insert i (files − {mo}))
      in Some (state ((op-elem oper) := after))

In the definition of the valid behaviours of the operations, we define additional assumption the state where the operation is executed. In essence, a the tag of a create, append, expunge, and store operation is identical to the message number and therefore unique. A delete operation deletes all metadata and the content of a folder. The store and expunge operations must refer to an existing message.

definition valid-behaviours :: ('id, 'a) state ⇒ 'id × ('id, 'a) operation ⇒ bool
  where
    valid-behaviours state msg ≡
      case msg of
        (i, Create j e) ⇒ i = j |
        (i, Delete is e) ⇒ is = fst (state e) ∪ snd (state e) |
        (i, Append j e) ⇒ i = j |
        (i, Expunge e mo j) ⇒ i = j ∧ mo ∈ snd (state e) |
        (i, Store e mo j) ⇒ i = j ∧ mo ∈ snd (state e)
locale imap = network-with-constrained-ops - interpret-op λx. { {}, {} } valid-behaviours
end

3 Commutativity of IMAP Commands

In this section we prove the commutativity of operations and identify the edge cases.

theory IMAP-proof-commute
imports IMAP-def
begin

lemma (in imap) create-create-commute:
  shows ⟨Create i1 e1⟩ ⊶ ⟨Create i2 e2⟩ = ⟨Create i2 e2⟩ ⊶ ⟨Create i1 e1⟩
  (proof)

lemma (in imap) create-delete-commute:
  assumes i /∈ is
  shows ⟨Create i e1⟩ ⊶ ⟨Delete is e2⟩ = ⟨Delete is e2⟩ ⊶ ⟨Create i e1⟩
  (proof)

lemma (in imap) create-append-commute:
  shows ⟨Create i1 e1⟩ ⊶ ⟨Append i2 e2⟩ = ⟨Append i2 e2⟩ ⊶ ⟨Create i1 e1⟩
  (proof)

lemma (in imap) create-expunge-commute:
  shows ⟨Create i1 e1⟩ ⊶ ⟨Expunge e2 mo i2⟩ = ⟨Expunge e2 mo i2⟩ ⊶ ⟨Create i1 e1⟩
  (proof)

lemma (in imap) create-store-commute:
  shows ⟨Create i1 e1⟩ ⊶ ⟨Store e2 mo i2⟩ = ⟨Store e2 mo i2⟩ ⊶ ⟨Create i1 e1⟩
  (proof)

lemma (in imap) delete-delete-commute:
  shows ⟨Delete i1 e1⟩ ⊶ ⟨Delete i2 e2⟩ = ⟨Delete i2 e2⟩ ⊶ ⟨Delete i1 e1⟩
  (proof)

lemma (in imap) delete-append-commute:
  assumes i /∈ is
  shows ⟨Delete is e1⟩ ⊶ ⟨Append i e2⟩ = ⟨Append i e2⟩ ⊶ ⟨Delete is e1⟩
  (proof)

lemma (in imap) delete-expunge-commute:
  assumes i /∈ is
  shows ⟨Delete is e1⟩ ⊶ ⟨Expunge e2 mo i⟩ = ⟨Expunge e2 mo i⟩ ⊶ ⟨Delete is e1⟩
  (proof)
lemma (in imap) delete-store-commute:
  assumes i \notin \text{is}
  shows \langle \text{Delete is e1} \rangle \triangleright \langle \text{Store e2 mo i} \rangle = \langle \text{Store e2 mo i} \rangle \triangleright \langle \text{Delete is e1} \rangle
  \langle \text{proof} \rangle

lemma (in imap) append-append-commute:
  shows \langle \text{Append i1 e1} \rangle \triangleright \langle \text{Append i2 e2} \rangle = \langle \text{Append i2 e2} \rangle \triangleright \langle \text{Append i1 e1} \rangle
  \langle \text{proof} \rangle

lemma (in imap) append-expunge-commute:
  assumes i1 \neq \text{mo}
  shows \langle \langle \text{Append i1 e1} \rangle \triangleright \langle \text{Expunge e2 mo i2} \rangle \rangle = \langle \langle \text{Expunge e2 mo i2} \rangle \triangleright \langle \text{Append i1 e1} \rangle \rangle
  \langle \text{proof} \rangle

lemma (in imap) append-store-commute:
  assumes i1 \neq \text{mo}
  shows \langle \langle \text{Append i1 e1} \rangle \triangleright \langle \text{Store e2 mo i2} \rangle \rangle = \langle \langle \text{Store e2 mo i2} \rangle \triangleright \langle \text{Append i1 e1} \rangle \rangle
  \langle \text{proof} \rangle

lemma (in imap) expunge-expunge-commute:
  shows \langle \langle \text{Expunge e1 mo1 i1} \rangle \triangleright \langle \text{Expunge e2 mo2 i2} \rangle \rangle = \langle \langle \text{Expunge e2 mo2 i2} \rangle \triangleright \langle \text{Expunge e1 mo1 i1} \rangle \rangle
  \langle \text{proof} \rangle

lemma (in imap) expunge-store-commute:
  assumes i1 \neq \text{mo2} \text{ and } i2 \neq \text{mo1}
  shows \langle \langle \text{Expunge e1 mo1 i1} \rangle \triangleright \langle \text{Store e2 mo2 i2} \rangle \rangle = \langle \langle \text{Store e2 mo2 i2} \rangle \triangleright \langle \text{Expunge e1 mo1 i1} \rangle \rangle
  \langle \text{proof} \rangle

lemma (in imap) store-store-commute:
  assumes i1 \neq \text{mo2} \text{ and } i2 \neq \text{mo1}
  shows \langle \langle \text{Store e1 mo1 i1} \rangle \triangleright \langle \text{Store e2 mo2 i2} \rangle \rangle = \langle \langle \text{Store e2 mo2 i2} \rangle \triangleright \langle \text{Store e1 mo1 i1} \rangle \rangle
  \langle \text{proof} \rangle

end

4 Proof Helpers

In this section we define and prove lemmas that help to show that all identified critical conditions hold for concurrent operations. Many of the following parts are derivations from the definitions and lemmas of Gomes et al.

theory
IMAP-proof-helpers
imports
IMAP-def
lemma (in imap) apply-operations-never-fails:
  assumes xs prefix of i
  shows apply-operations xs ≠ None
⟨proof⟩

lemma (in imap) create-id-valid:
  assumes xs prefix of j
  and Deliver (i1, Create i2 e) ∈ set xs
  shows i1 = i2
⟨proof⟩

lemma (in imap) append-id-valid:
  assumes xs prefix of j
  and Deliver (i1, Append i2 e) ∈ set xs
  shows i1 = i2
⟨proof⟩

lemma (in imap) expunge-id-valid:
  assumes xs prefix of j
  and Deliver (i1, Expunge e mo i2) ∈ set xs
  shows i1 = i2
⟨proof⟩

lemma (in imap) store-id-valid:
  assumes xs prefix of j
  and Deliver (i1, Store e mo i2) ∈ set xs
  shows i1 = i2
⟨proof⟩

definition (in imap) added-ids :: ('id × ('id, 'b) operation) event list ⇒ 'b ⇒ 'id list where
  added-ids es p ≡ List.map-filter (λx. case x of
    Deliver (i, Create j e) ⇒ if e = p then Some j else None |
    Deliver (i, Expunge e mo j) ⇒ if e = p then Some j else None |
    - ⇒ None) es

definition (in imap) added-files :: ('id × ('id, 'b) operation) event list ⇒ 'b ⇒ 'id list where
  added-files es p ≡ List.map-filter (λx. case x of
    Deliver (i, Append j e) ⇒ if e = p then Some j else None |
    Deliver (i, Store e mo j) ⇒ if e = p then Some j else None |
    - ⇒ None) es

— added files simplifier

lemma (in imap) [simp]:
  shows added-files [] e = []
⟨proof⟩
lemma (in imap) [simp]:
  shows added-files (xs @ ys) e = added-files xs e @ added-files ys e
  ⟨proof⟩

lemma (in imap) added-files-Broadcast-collapse [simp]:
  shows added-files ([Broadcast e]) e' = []
  ⟨proof⟩

lemma (in imap) added-files-Deliver-Delete-collapse [simp]:
  shows added-files ([Deliver (i, Delete is e)]) e' = []
  ⟨proof⟩

lemma (in imap) added-files-Deliver-Create-collapse [simp]:
  shows added-files ([Deliver (i, Create j e)]) e' = []
  ⟨proof⟩

lemma (in imap) added-files-Deliver-Expunge-collapse [simp]:
  shows added-files ([Deliver (i, Expunge e mo j)]) e' = []
  ⟨proof⟩

lemma (in imap) added-files-Deliver-Append-diff-collapse [simp]:
  shows e ≠ e' ⇒ added-files ([Deliver (i, Append j e)]) e' = []
  ⟨proof⟩

lemma (in imap) added-files-Deliver-Append-same-collapse [simp]:
  shows added-files ([Deliver (i, Append j e)]) e = [j]
  ⟨proof⟩

lemma (in imap) added-files-Deliver-Store-diff-collapse [simp]:
  shows e ≠ e' ⇒ added-files ([Deliver (i, Store e mo j)]) e' = []
  ⟨proof⟩

lemma (in imap) added-files-Deliver-Store-same-collapse [simp]:
  shows added-files ([Deliver (i, Store e mo j)]) e = [j]
  ⟨proof⟩

lemma (in imap) [simp]:
  shows added-ids [] e = []
  ⟨proof⟩

lemma (in imap) split-ids [simp]:
  shows added-ids (xs @ ys) e = added-ids xs e @ added-ids ys e
  ⟨proof⟩

lemma (in imap) added-ids-Broadcast-collapse [simp]:
  shows added-ids ([Broadcast e]) e' = []
  ⟨proof⟩

lemma (in imap) added-ids-Deliver-Delete-collapse [simp]:
shows added-ids ([Deliver (i, Delete is e)]) e' = []
(proof)

lemma (in imap) added-ids-Deliver-Append-collapse [simp]:
  shows added-ids ([Deliver (i, Append j e)]) e' = []
(proof)

lemma (in imap) added-ids-Deliver-Store-collapse [simp]:
  shows added-ids ([Deliver (i, Store e mo j)]) e' = []
(proof)

lemma (in imap) added-ids-Deliver-Create-diff-collapse [simp]:
  shows e ≠ e' → added-ids ([Deliver (i, Create j e)]) e' = []
(proof)

lemma (in imap) added-ids-Deliver-Expunge-diff-collapse [simp]:
  shows e ≠ e' → added-ids ([Deliver (i, Expunge e mo j)]) e' = []
(proof)

lemma (in imap) added-ids-Deliver-Create-same-collapse [simp]:
  shows added-ids ([Deliver (i, Create j e)]) e = [j]
(proof)

lemma (in imap) added-ids-Deliver-Expunge-same-collapse [simp]:
  shows added-ids ([Deliver (i, Expunge e mo j)]) e = [j]
(proof)

lemma (in imap) expunge-id-not-in-set:
  assumes i1 /∈ set (added-ids [Deliver (i, Expunge e mo i2)] e)
  shows i1 ≠ i2
(proof)

lemma (in imap) apply-operations-added-ids:
  assumes es prefix of j
  and apply-operations es = Some f
  shows fst (f x) ⊆ set (added-ids es x)
(proof)

lemma (in imap) apply-operations-added-files:
  assumes es prefix of j
  and apply-operations es = Some f
  shows snd (f x) ⊆ set (added-files es x)
(proof)

lemma (in imap) Deliver-added-files:
  assumes xs prefix of j
  and i ∈ set (added-files xs e)
  shows Deliver (i, Append i e) ∈ set xs ∨ (∃ mo. Deliver (i, Store e mo i) ∈ set xs)
(proof)
5 Independence of IMAP Commands

In this section we show that two concurrent operations that reference to the same tag must be identical.

theory IMAP−proof−independent
imports IMAP−def IMAP−proof−helpers
begin

lemma (in imap) Broadcast-Expunge-Deliver-prefix-closed:
  assumes xs @ [Broadcast (i, Expunge e mo i)] prefix of j
  shows Deliver (mo, Append mo e) ∈ set xs ∨
          (∃ mo2 . Deliver (mo, Store e mo2 mo) ∈ set xs)
⟨proof⟩

lemma (in imap) Broadcast-Store-Deliver-prefix-closed:
  assumes xs @ [Broadcast (i, Store e mo i)] prefix of j
  shows Deliver (mo, Append mo e) ∈ set xs ∨
          (∃ mo2 . Deliver (mo, Store e mo2 mo) ∈ set xs)
⟨proof⟩

lemma (in imap) Deliver-added-ids:
  assumes xs prefix of j
  and i ∈ set (added-ids xs e)
  shows Deliver (i, Create i e) ∈ set xs ∨
          (∃ mo . Deliver (i, Expunge e mo i) ∈ set xs)
⟨proof⟩

lemma (in imap) Broadcast-Deliver-prefix-closed:
  assumes xs @ [Broadcast (r, Delete ix e)] prefix of j
  and i ∈ ix
  shows Deliver (i, Create i e) ∈ set xs ∨
          Deliver (i, Append i e) ∈ set xs ∨
          (∃ mo . Deliver (i, Expunge e mo i) ∈ set xs) ∨
          (∃ mo . Deliver (i, Store e mo i) ∈ set xs)
⟨proof⟩

lemma (in imap) concurrent-create-delete-independent-technical:
  assumes i ∈ is
  and xs prefix of j
  and (i, Create i e) ∈ set (node-deliver-messages xs)
  and (ir, Delete is e) ∈ set (node-deliver-messages xs)
  shows hb (i, Create i e) (ir, Delete is e)
lemma (in imap) concurrent-store-expunge-independent-technical:
assumes xs prefix of j
and (i, Store e mo i) ∈ set (node-deliver-messages xs)
and (r, Expunge e i r) ∈ set (node-deliver-messages xs)
shows hb (i, Store e mo i) (r, Expunge e i r)
⟨proof⟩

lemma (in imap) concurrent-store-expunge-independent-technical2:
assumes xs prefix of j
and (i, Store e1 mo2 i) ∈ set (node-deliver-messages xs)
and (r, Expunge e mo r) ∈ set (node-deliver-messages xs)
shows mo2 ≠ r
⟨proof⟩

lemma (in imap) concurrent-store-delete-independent-technical:
assumes i ∈ is
and xs prefix of j
and (i, Store e mo i) ∈ set (node-deliver-messages xs)
and (ir, Delete is e) ∈ set (node-deliver-messages xs)
shows hb (i, Store e mo i) (ir, Delete is e)
⟨proof⟩

lemma (in imap) concurrent-append-delete-independent-technical:
assumes i ∈ is
and xs prefix of j
and (i, Append i e) ∈ set (node-deliver-messages xs)
and (ir, Delete is e) ∈ set (node-deliver-messages xs)
shows hb (i, Append i e) (ir, Delete is e)
⟨proof⟩

lemma (in imap) concurrent-append-expunge-independent-technical:
assumes i = mo
and xs prefix of j
and (i, Append i e) ∈ set (node-deliver-messages xs)
and (r, Expunge e mo r) ∈ set (node-deliver-messages xs)
shows hb (i, Append i e) (r, Expunge e mo r)
⟨proof⟩

lemma (in imap) concurrent-append-store-independent-technical:
assumes i = mo
and xs prefix of j
and (i, Append i e) ∈ set (node-deliver-messages xs)
and (r, Store e mo r) ∈ set (node-deliver-messages xs)
shows hb (i, Append i e) (r, Store e mo r)
⟨proof⟩

lemma (in imap) concurrent-expunge-delete-independent-technical:
assumes $i \in is$
and $xs$ prefix of $j$
and $(i, \text{Expunge } e \text{ mo } i) \in \text{set (node-deliver-messages xs)}$
and $(ir, \text{Delete is } e) \in \text{set (node-deliver-messages xs)}$
shows $hb (i, \text{Expunge } e \text{ mo } i) (ir, \text{Delete is } e)$

lemma (in imap) concurrent-store-store-independent-technical:
assumes $xs$ prefix of $j$
and $(i, \text{Store e mo } i) \in \text{set (node-deliver-messages xs)}$
and $(r, \text{Store e i r}) \in \text{set (node-deliver-messages xs)}$
shows $hb (i, \text{Store e mo } i) (r, \text{Store e i r})$

lemma (in imap) expunge-delete-tag-causality:
assumes $i \in is$
and $xs$ prefix of $j$
and $(i, \text{Expunge } e1 \text{ mo } i) \in \text{set (node-deliver-messages xs)}$
and $(ir, \text{Delete is } e2) \in \text{set (node-deliver-messages xs)}$
and $\text{pre@[Broadcast } (ir, \text{Delete is } e2)] \text{ prefix of } k$
shows $\text{Deliver } (i, \text{Expunge } e2 \text{ mo } i) \in \text{set (history k)}$

lemma (in imap) expunge-delete-ids-imply-messages-same:
assumes $i \in is$
and $xs$ prefix of $j$
and $(i, \text{Expunge } e1 \text{ mo } i) \in \text{set (node-deliver-messages xs)}$
and $(ir, \text{Delete is } e2) \in \text{set (node-deliver-messages xs)}$
shows $e1 = e2$

lemma (in imap) store-delete-ids-imply-messages-same:
assumes $i \in is$
and $xs$ prefix of $j$
and $(i, \text{Store e1 mo } i) \in \text{set (node-deliver-messages xs)}$
and $(ir, \text{Delete is } e2) \in \text{set (node-deliver-messages xs)}$
shows $e1 = e2$

lemma (in imap) create-delete-ids-imply-messages-same:
assumes $i \in is$
and $xs$ prefix of $j$
and $(i, \text{Create i e1}) \in \text{set (node-deliver-messages xs)}$
and $(ir, \text{Delete is } e2) \in \text{set (node-deliver-messages xs)}$
shows $e1 = e2$

lemma (in imap) append-delete-ids-imply-messages-same:
assumes $i \in is$
and $xs$ prefix of $j$
and $(i, Append i e1) \in set (node-deliver-messages xs)$
and $(ir, Delete is e2) \in set (node-deliver-messages xs)$
shows $e1 = e2$

lemma (in imap) append-expunge-ids-imply-messages-same:
assumes $i = mo$
and $xs$ prefix of $j$
and $(i, Append i e1) \in set (node-deliver-messages xs)$
and $(r, Expunge e2 mo r) \in set (node-deliver-messages xs)$
shows $e1 = e2$

lemma (in imap) append-store-ids-imply-messages-same:
assumes $i = mo$
and $xs$ prefix of $j$
and $(i, Append i e1) \in set (node-deliver-messages xs)$
and $(r, Store e2 mo r) \in set (node-deliver-messages xs)$
shows $e1 = e2$

lemma (in imap) expunge-store-ids-imply-messages-same:
assumes $xs$ prefix of $j$
and $(i, Store e1 mo i) \in set (node-deliver-messages xs)$
and $(r, Expunge e2 i r) \in set (node-deliver-messages xs)$
shows $e1 = e2$

lemma (in imap) store-store-ids-imply-messages-same:
assumes $xs$ prefix of $j$
and $(i, Store e1 mo i) \in set (node-deliver-messages xs)$
and $(r, Store e2 i r) \in set (node-deliver-messages xs)$
shows $e1 = e2$

end

6 Convergence of the IMAP-CRDT

In this final section show that concurrent updates commute and thus Strong Eventual Convergence is achieved.

theory IMAP-proof
imports IMAP-def IMAP-proof-commute IMAP-proof-helpers
**IMAP-proof-independent**

begin

corollary (in imap) concurrent-create-delete-independent:
assumes \( \neg \, \text{hb} \,(i, \, \text{Create} \, i \, e1) \, (ir, \, \text{Delete} \, is \, e2) \)
and \( \neg \, \text{hb} \,(ir, \, \text{Delete} \, is \, e2) \,(i, \, \text{Create} \, i \, e1) \)
and \( xs \, \text{prefix} \, of \, j \)
and \( (i, \, \text{Create} \, i \, e1) \, \in \, \text{set} \,(\text{node-deliver-messages} \, xs) \)
and \( (ir, \, \text{Delete} \, is \, e2) \, \in \, \text{set} \,(\text{node-deliver-messages} \, xs) \)
shows \( i \notin is \)
(proof)

corollary (in imap) concurrent-append-delete-independent:
assumes \( \neg \, \text{hb} \,(i, \, \text{Append} \, i \, e1) \, (ir, \, \text{Delete} \, is \, e2) \)
and \( \neg \, \text{hb} \,(ir, \, \text{Delete} \, is \, e2) \,(i, \, \text{Append} \, i \, e1) \)
and \( xs \, \text{prefix} \, of \, j \)
and \( (i, \, \text{Append} \, i \, e1) \, \in \, \text{set} \,(\text{node-deliver-messages} \, xs) \)
and \( (ir, \, \text{Delete} \, is \, e2) \, \in \, \text{set} \,(\text{node-deliver-messages} \, xs) \)
shows \( i \notin is \)
(proof)

corollary (in imap) concurrent-append-expunge-independent:
assumes \( \neg \, \text{hb} \,(i, \, \text{Append} \, i \, e1) \, (r, \, \text{Expunge} \, e2 \, mo \, r) \)
and \( \neg \, \text{hb} \,(r, \, \text{Expunge} \, e2 \, mo \, r) \,(i, \, \text{Append} \, i \, e1) \)
and \( xs \, \text{prefix} \, of \, j \)
and \( (i, \, \text{Append} \, i \, e1) \, \in \, \text{set} \,(\text{node-deliver-messages} \, xs) \)
and \( (r, \, \text{Expunge} \, e2 \, mo \, r) \, \in \, \text{set} \,(\text{node-deliver-messages} \, xs) \)
shows \( i \neq mo \)
(proof)

corollary (in imap) concurrent-append-store-independent:
assumes \( \neg \, \text{hb} \,(i, \, \text{Append} \, i \, e1) \, (r, \, \text{Store} \, e2 \, mo \, r) \)
and \( \neg \, \text{hb} \,(r, \, \text{Store} \, e2 \, mo \, r) \,(i, \, \text{Append} \, i \, e1) \)
and \( xs \, \text{prefix} \, of \, j \)
and \( (i, \, \text{Append} \, i \, e1) \, \in \, \text{set} \,(\text{node-deliver-messages} \, xs) \)
and \( (r, \, \text{Store} \, e2 \, mo \, r) \, \in \, \text{set} \,(\text{node-deliver-messages} \, xs) \)
shows \( i \neq mo \)
(proof)

corollary (in imap) concurrent-expunge-delete-independent:
assumes \( \neg \, \text{hb} \,(i, \, \text{Expunge} \, e1 \, mo \, i) \, (ir, \, \text{Delete} \, is \, e2) \)
and \( \neg \, \text{hb} \,(ir, \, \text{Delete} \, is \, e2) \,(i, \, \text{Expunge} \, e1 \, mo \, i) \)
and \( xs \, \text{prefix} \, of \, j \)
and \( (i, \, \text{Expunge} \, e1 \, mo \, i) \, \in \, \text{set} \,(\text{node-deliver-messages} \, xs) \)
and \( (ir, \, \text{Delete} \, is \, e2) \, \in \, \text{set} \,(\text{node-deliver-messages} \, xs) \)
shows \( i \notin is \)
(proof)

corollary (in imap) concurrent-store-delete-independent:
assumes $\neg hb (i, Store e1 mo i)$ $(ir, Delete is e2)$ and $\neg hb (ir, Delete is e2)$ $(i, Store e1 mo i)$ and $xs$ prefix of $j$ and $(i, Store e1 mo i) \in set (node-deliver-messages xs)$ and $(ir, Delete is e2) \in set (node-deliver-messages xs)$ shows $i \not\in is$

\begin{proof}
\end{proof}

corollary (in imap) concurrent-store-expunge-independent:
assumes $\neg hb (i, Store e1 mo i)$ $(r, Expunge e2 mo2 r)$ and $\neg hb (r, Expunge e2 mo2 r)$ $(i, Store e1 mo i)$ and $xs$ prefix of $j$ and $(i, Store e1 mo i) \in set (node-deliver-messages xs)$ and $(r, Expunge e2 mo2 r) \in set (node-deliver-messages xs)$ shows $i \not= mo2 \land r \not= mo$

\begin{proof}
\end{proof}

corollary (in imap) concurrent-store-store-independent:
assumes $\neg hb (i, Store e1 mo i)$ $(r, Store e2 mo2 r)$ and $\neg hb (r, Store e2 mo2 r)$ $(i, Store e1 mo i)$ and $xs$ prefix of $j$ and $(i, Store e1 mo i) \in set (node-deliver-messages xs)$ and $(r, Store e2 mo2 r) \in set (node-deliver-messages xs)$ shows $i \not= mo2 \land r \not= mo$

\begin{proof}
\end{proof}

lemma (in imap) concurrent-operations-commute:
assumes $xs$ prefix of $i$
shows $hb.\text{concurrent-ops-commute} (node-deliver-messages xs)$

\begin{proof}
\end{proof}

theorem (in imap) convergence:
assumes set $(node-deliver-messages xs) = set (node-deliver-messages ys)$ and $xs$ prefix of $i$ and $ys$ prefix of $j$
shows $apply-operations xs = apply-operations ys$

\begin{proof}
\end{proof}

corollary (in imap) concurrent-store-expunge-independent:
assumes $\neg hb (i, Store e1 mo i)$ $(r, Expunge e2 mo2 r)$ and $\neg hb (r, Expunge e2 mo2 r)$ $(i, Store e1 mo i)$ and $xs$ prefix of $j$ and $(i, Store e1 mo i) \in set (node-deliver-messages xs)$ and $(r, Expunge e2 mo2 r) \in set (node-deliver-messages xs)$ shows $i \not\in is$

\begin{proof}
\end{proof}
}

\textbf{context imap begin}

\textbf{sublocale sec: strong-eventual-consistency weak-hb hb interp-msg}
$\lambda ops.\exists xs i. xs$ prefix of $i \land$ node-deliver-messages $xs = ops \lambda x.\{\},\{\}$

\begin{proof}
\end{proof}

end

end
References


