# The IMAP CmRDT 

Tim Jungnickel, Lennart Oldenburg, Matthias Loibl

September 13, 2023


#### 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 "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 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 pluto, our research prototype of a planetaryscale 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 $\mathcal{N}$ ) and messages (of type $\mathcal{M}$ ). 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 $\mathcal{M}$ and that the generated tags of create and expunge are handled explicitly. This makes the formalization slightly easier, because less type variables are introduced. The concrete definition can be found in the IMAP-CRDT Definitions section of the IMAP-def.thy file.

### 1.2 Proof Guide

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 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 nitpick to identify corner cases in our implementation. We prove the commutativity in Section 3 of the IMAP-proof-commute.thy file. The critical conditions to satisfy in order to commute, can be summarized as follows:

- The tags of a create and expunge operation or the messages of an append and store operation are never in the removed-set of a concurrent delete operation.
- The message of an append operation is never the message that is deleted by a concurrent store or expunge operation.
- The message inserted by a store operation is never the message that is deleted by a concurrent store or 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
    payload map \(u: \mathcal{N} \rightarrow \mathcal{P}(\mathrm{ID}) \times \mathcal{P}(\mathcal{M}) \quad \triangleright\{\) foldername \(f \mapsto(\{\operatorname{tag} t\},\{\operatorname{msg} m\}), \ldots\}\)
        initial \((\lambda x .(\varnothing, \varnothing))\)
    update create (foldername f)
        atSource
            let \(\alpha=\) unique()
        downstream \((f, \alpha)\)
            \(u(f) \mapsto\left(u(f)_{1} \cup\{\alpha\}, u(f)_{2}\right)\)
    update delete (foldername \(f\) )
        atSource ( \(f\) )
            let \(R_{1}=u(f)_{1}\)
            let \(R_{2}=u(f)_{2}\)
        downstream \(\left(f, R_{1}, R_{2}\right)\)
            \(u(f) \mapsto\left(u(f)_{1} \backslash R_{1}, u(f)_{2} \backslash R_{2}\right)\)
    update append (foldername \(f\), message \(m\) )
        atSource ( \(m\) )
            pre \(m\) is globally unique
        downstream \((f, m)\)
            \(u(f) \mapsto\left(u(f)_{1}, u(f)_{2} \cup\{m\}\right)\)
    update expunge (foldername \(f\), message \(m\) )
        atSource ( \(f, m\) )
                pre \(m \in u(f)_{2}\)
                let \(\alpha=\) unique ()
        downstream \((f, m, \alpha)\)
            \(u(f) \mapsto\left(u(f)_{1} \cup\{\alpha\}, u(f)_{2} \backslash\{m\}\right)\)
    update store (foldername \(f\), message \(m_{\text {old }}\), message \(m_{\text {new }}\) )
        atSource \(\left(f, m_{\text {old }}, m_{\text {new }}\right)\)
                pre \(m_{\text {old }} \in u(f)_{2}\)
                pre \(m_{\text {new }}\) is globally unique
        downstream ( \(f, m_{\text {old }}, m_{\text {new }}\) )
            \(u(f) \mapsto\left(u(f)_{1},\left(u(f)_{2} \backslash\left\{m_{\text {old }}\right\}\right) \cup\left\{m_{\text {new }}\right\}\right)\)
```

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-proofindependent.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 ' \(a \mid\)
    Append 'id 'a \(\mid\)
    Expunge 'a 'id 'id |
    Store 'a 'id 'id
type-synonym \(\left({ }^{\prime} i d,{ }^{\prime} a\right)\) state \(={ }^{\prime} a \Rightarrow\left({ }^{\prime} i d\right.\) set \(\times\) 'id set \()\)
definition op-elem \(::\left(' i d,{ }^{\prime} a\right)\) operation \(\Rightarrow{ }^{\prime} a\) where
    op-elem oper \(\equiv\) case oper of
        Create \(i e \Rightarrow e \mid\)
        Delete is \(e \Rightarrow e \mid\)
        Append \(i e \Rightarrow e\)
        Expunge e mo \(i \Rightarrow e \mid\)
        Store e mo \(i \Rightarrow e\)
definition interpret-op :: ('id, 'a) operation \(\Rightarrow\left({ }^{\prime} i d,{ }^{\prime} a\right)\) state \(\rightharpoonup\left({ }^{\prime} i d,{ }^{\prime} a\right)\) state
    ( \(\langle-\rangle[0]\) 1000) where
    interpret-op oper state \(\equiv\)
        let metadata \(=\) fst \((\) state \((\) op-elem oper \())\);
        files \(=\) snd \((\) state \((\) op-elem oper \()) ;\)
        after \(=\) case oper of
            Create \(i e \Rightarrow(\) metadata \(\cup\{i\}\), files \() \mid\)
            Delete is \(e \Rightarrow\) (metadata - is, files - is) \(\mid\)
            Append \(i e \Rightarrow(\) metadata, files \(\cup\{i\}) \mid\)
            Expunge e mo \(i \Rightarrow(\) metadata \(\cup\{i\}\), files \(-\{m o\}) \mid\)
            Store e mo \(i \Rightarrow(\) metadata, insert \(i(\) files \(-\{m o\}))\)
        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 \(\Rightarrow{ }^{\prime} i d \times\left(' i d,{ }^{\prime} a\right)\) operation \(\Rightarrow\) bool where
    valid-behaviours state \(\mathrm{msg} \equiv\)
    case msg of
    ( \(i\), Create \(j e) \Rightarrow i=j \mid\)
    \((i\), Delete is e) \(\Rightarrow i s=f s t(\) state e) \(\cup\) snd (state e) \(\mid\)
    ( \(i\), Append \(j e\) ) \(\Rightarrow i=j \mid\)
    \((i\), Expunge e mo \(j) \Rightarrow i=j \wedge m o \in\) snd (state e) \(\mid\)
    ( \(i\), Store e mo \(j\) ) \(\Rightarrow i=j \wedge m o \in\) snd (state e)
```

locale imap $=$ network-with-constrained-ops - interpret-op $\lambda x .(\{ \},\{ \})$ 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 \langleCreate i1 e1\rangle\triangleright \Create i2 e2 \}=\langle\mathrm{ Create i2 e2 }\rangle\triangleright\langle\mathrm{ Create i1 e1
    <proof\rangle
lemma (in imap) create-delete-commute:
    assumes i\not\inis
    shows \langleCreate i e1\rangle\triangleright \Delete is e2 }\rangle=\langle\mathrm{ Delete is e2 }\rangle\triangleright\langle\mathrm{ Create i e1 
    <proof\rangle
lemma (in imap) create-append-commute:
    shows \langleCreate i1 e1\rangle\triangleright \Append i2 e2 \rangle}=\langle\mathrm{ Append i2 e2 }\rangle\triangleright\langle\mathrm{ Create i1 e1}
    <proof\rangle
lemma (in imap) create-expunge-commute:
    shows \langleCreate i1 e1\rangle\triangleright <Expunge e2 mo i2\rangle=\langleExpunge e2 mo i2\rangle\triangleright \Create i1 e1\rangle
    <proof\rangle
lemma (in imap) create-store-commute:
    shows \langleCreate i1 e1\rangle\triangleright \langleStore e2 mo i2 \rangle}=\langle\mathrm{ Store e2 mo i2 }\rangle\triangleright\langle\mathrm{ Create i1 e1 
    <proof\rangle
lemma (in imap) delete-delete-commute:
    shows }\langle\mathrm{ Delete i1 e1}\rangle\triangleright\langle\mathrm{ Delete i2 e2 }\rangle=\langle\mathrm{ Delete i2 e2 }\rangle\triangleright\langle\mathrm{ Delete i1 e1}
    <proof\rangle
lemma (in imap) delete-append-commute:
    assumes i\not\inis
    shows }\langle\mathrm{ Delete is e1}\rangle\triangleright\langle\mathrm{ Append i e2 }\rangle=\langle\mathrm{ Append i e2 }\rangle\triangleright\langle\mathrm{ Delete is e1}
    \langleproof\rangle
lemma (in imap) delete-expunge-commute:
    assumes i\not\inis
    shows }\langle\mathrm{ Delete is e1 }\rangle\triangleright\langle\mathrm{ Expunge e2 mo i }\rangle=\langle\mathrm{ Expunge e2 mo i }\rangle\triangleright\langle\mathrm{ Delete is e1 }
    <proof\rangle
```

lemma（in imap）delete－store－commute：
assumes $i \notin i s$
shows $\langle$ Delete is e1 $\rangle \triangleright\langle$ Store e2 mo $i\rangle=\langle$ Store e2 mo $i\rangle \triangleright\langle$ Delete is e1 $\rangle$
$\langle p r o o f\rangle$
lemma（in imap）append－append－commute：
shows $\langle$ Append i1 e1 $\rangle \triangleright\langle$ Append i2 e2 $\rangle=\langle$ Append i2 e2 $\rangle \triangleright\langle$ Append i1 e1 $\rangle$
$\langle p r o o f\rangle$
lemma（in imap）append－expunge－commute：
assumes $i 1 \neq m o$
shows $(\langle$ Append i1 e1 $\rangle \triangleright\langle$ Expunge e2 mo i2 $\rangle)=(\langle$ Expunge e2 mo i2 $\rangle \triangleright\langle$ Append i1 e1 $\rangle)$
〈proof〉
lemma（in imap）append－store－commute：
assumes $i 1 \neq m o$
shows $(\langle$ Append i1 e1 $\rangle \triangleright\langle$ Store e2 mo i2 $\rangle)=(\langle$ Store e2 mo i2 $\rangle \triangleright\langle$ Append i1 e1 $\rangle)$〈proof〉
lemma（in imap）expunge－expunge－commute：
shows $(\langle$ Expunge e1 mo1 i1 $\rangle \triangleright\langle$ Expunge e2 mo2 i2 $\rangle)=(\langle$ Expunge e2 mo2 i2 $\rangle \triangleright\langle$ Expunge e1 mo1 i1 $\rangle$ ）
$\langle p r o o f\rangle$
lemma（in imap）expunge－store－commute：
assumes $i 1 \neq m o 2$ and $i 2 \neq m o 1$
shows $(\langle$ Expunge e1 mo1 i1 $\rangle \triangleright\langle$ Store e2 mo2 i2 $\rangle)=(\langle$ Store e2 mo2 i2 $\rangle \triangleright\langle$ Expunge e1 mo1
i1〉）
$\langle p r o o f\rangle$
lemma（in imap）store－store－commute：
assumes $i 1 \neq m o 2$ and $i 2 \neq m o 1$
shows $(\langle$ Store e1 mo1 i1 $\rangle \triangleright\langle$ Store e2 mo2 i2 $\rangle)=(\langle$ Store e2 mo2 i2 $\rangle \triangleright\langle$ Store e1 mo1 i1 $\rangle)$〈proof〉
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
```


## begin

```
lemma (in imap) apply-operations-never-fails:
    assumes xs prefix of i
    shows apply-operations xs }\not==\mathrm{ None
    <proof>
lemma (in imap) create-id-valid:
    assumes xs prefix of j
        and Deliver (i1, Create i2 e) \in set xs
    shows i1 = i2
<proof>
lemma (in imap) append-id-valid:
    assumes xs prefix of j
        and Deliver (i1, Append i2 e) \in set xs
    shows i1 = i2
<proof>
lemma (in imap) expunge-id-valid:
    assumes xs prefix of j
        and Deliver (i1, Expunge e mo i2) \in set xs
    shows i1 = i2
\langleproof\rangle
lemma (in imap) store-id-valid:
    assumes xs prefix of j
        and Deliver (i1, Store e mo i2) \in set xs
    shows i1 = i2
<proof>
```

definition (in imap) added-ids :: ('id $\times\left({ }^{\prime} i d,{ }^{\prime} b\right)$ operation) event list $\Rightarrow{ }^{\prime} b \Rightarrow{ }^{\prime}$ id list where
added-ids es $p \equiv$ List.map-filter ( $\lambda x$. case $x$ of
Deliver ( $i$, Create $j e) \Rightarrow$ if $e=p$ then Some $j$ else None
Deliver ( $i$, Expunge e mo $j) \Rightarrow$ if $e=p$ then Some $j$ else None
- $\Rightarrow$ None) es
definition (in imap) added-files :: ('id $\times$ ('id, 'b) operation) event list $\Rightarrow{ }^{\prime} b \Rightarrow$ 'id list where
added-files es $p \equiv$ List.map-filter ( $\lambda x$. case $x$ of
Deliver ( $i$, Append $j e) \Rightarrow$ if $e=p$ then Some $j$ else None $\mid$
Deliver ( $i$, Store e mo $j$ ) $\Rightarrow$ if $e=p$ then Some $j$ else None $\mid$
- $\Rightarrow$ 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
    \langleproof\rangle
lemma (in imap) added-files-Broadcast-collapse [simp]:
    shows added-files ([Broadcast e]) e' = []
    \langleproof\rangle
lemma (in imap) added-files-Deliver-Delete-collapse [simp]:
    shows added-files ([Deliver (i, Delete is e)]) e' = []
    \langleproof\rangle
lemma (in imap) added-files-Deliver-Create-collapse [simp]:
    shows added-files ([Deliver (i, Create j e)]) e' = []
    \langleproof\rangle
lemma (in imap) added-files-Deliver-Expunge-collapse [simp]:
    shows added-files ([Deliver (i,Expunge e mo j)]) e'= []
    \langleproof\rangle
lemma (in imap) added-files-Deliver-Append-diff-collapse [simp]:
    shows e\not=\mp@subsup{e}{}{\prime}\Longrightarrow\mathrm{ added-files ([Deliver (i,Append je)]) e}\mp@subsup{e}{}{\prime}=[]
    \langleproof\rangle
lemma (in imap) added-files-Deliver-Append-same-collapse [simp]:
    shows added-files ([Deliver (i,Append j e)]) e= [j]
    \langleproof\rangle
lemma (in imap) added-files-Deliver-Store-diff-collapse [simp]:
    shows e f= ''\Longrightarrow added-files ([Deliver (i,Store e mo j)]) e' = []
    \langleproof\rangle
lemma (in imap) added-files-Deliver-Store-same-collapse [simp]:
    shows added-files ([Deliver (i, Store e mo j)]) e= [j]
    \langleproof\rangle
lemma (in imap) [simp]:
    shows added-ids [] e= []
    \langleproof\rangle
lemma (in imap) split-ids [simp]:
    shows added-ids (xs @ ys) e=added-ids xs e @ added-ids ys e
    \langleproof\rangle
lemma (in imap) added-ids-Broadcast-collapse [simp]:
    shows added-ids ([Broadcast e]) e' = []
    \langleproof
lemma (in imap) added-ids-Deliver-Delete-collapse [simp]:
```

```
shows added-ids ([Deliver (i, Delete is e)]) }\mp@subsup{e}{}{\prime}=[
\langleproof\rangle
```

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

lemma (in imap) added-ids-Deliver-Create-diff-collapse [simp]:
shows $e \neq e^{\prime} \Longrightarrow$ added-ids $([$ Deliver ( $i$, Create $\left.j e)]\right) e^{\prime}=[]$
$\langle p r o o f\rangle$
lemma (in imap) added-ids-Deliver-Expunge-diff-collapse [simp]:
shows $e \neq e^{\prime} \Longrightarrow$ added-ids $\left(\left[\right.\right.$ Deliver ( $i$, Expunge e mo $j$ )]) $e^{\prime}=[]$
$\langle p r o o f\rangle$
lemma (in imap) added-ids-Deliver-Create-same-collapse [simp]:
shows added-ids $([$ Deliver ( $i$, Create $j e)]) e=[j]$
$\langle p r o o f\rangle$
lemma (in imap) added-ids-Deliver-Expunge-same-collapse [simp]:
shows added-ids ([Deliver (i, Expunge e mo j)]) e=[j]
$\langle p r o o f\rangle$
lemma (in imap) expunge-id-not-in-set:

shows $i 1 \neq i 2$
$\langle p r o o f\rangle$
lemma (in imap) apply-operations-added-ids:
assumes es prefix of $j$
and apply-operations es $=$ Some $f$
shows $f$ st $(f x) \subseteq$ set (added-ids es $x)$
$\langle p r o o f\rangle$
lemma (in imap) apply-operations-added-files:
assumes es prefix of $j$
and apply-operations es $=$ Some $f$
shows snd $(f x) \subseteq$ set (added-files es $x)$
$\langle p r o o f\rangle$
lemma (in imap) Deliver-added-files:
assumes xs prefix of $j$
and $i \in \operatorname{set}$ (added-files xs e)
shows Deliver $(i$, Append $i e) \in$ set $x s \vee(\exists$ mo. Deliver $(i$, Store $e$ mo $i) \in$ set $x s)$
$\langle p r o o f\rangle$
end

## 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) \in set xs \vee
        (\exists mo2 . Deliver (mo, Store e mo2 mo) \in 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) \in set xs \vee
        moz . Deliver (mo, Store e mo2 mo) \in set xs)
<proof>
lemma (in imap) Deliver-added-ids:
    assumes xs prefix of j
        and i\in set (added-ids xs e)
    shows Deliver (i, Create i e) \in set xs }
        (\exists mo . Deliver (i, Expunge e mo i) \in set xs)
    <proof\rangle
lemma (in imap) Broadcast-Deliver-prefix-closed:
    assumes xs @ [Broadcast (r, Delete ix e)] prefix of j
        and i\inix
    shows Deliver (i, Create i e) \in set xs \vee
        Deliver (i, Append i e) \in set xs \vee
        (\exists mo. Deliver (i, Expunge e mo i) \in set xs) \vee
        (\exists mo.Deliver (i,Store e mo i) \in set xs)
<proof\rangle
```

lemma (in imap) concurrent-create-delete-independent-technical:
assumes $i \in$ is
and xs prefix of $j$
and ( $i$, Create $i$ e) $\in$ set (node-deliver-messages xs)
and (ir, Delete is e) $\in$ set (node-deliver-messages xs)
shows hb (i, Create ie) (ir, Delete is e)

```
\langleproof\rangle
```

```
lemma (in imap) concurrent-store-expunge-independent-technical:
    assumes xs prefix of j
    and (i, Store e mo i) \in set (node-deliver-messages xs)
    and (r, Expunge e i r) \in set (node-deliver-messages xs)
    shows hb (i, Store e mo i)(r, Expunge e i r)
<proof\rangle
```

lemma (in imap) concurrent-store-expunge-independent-technical2:
assumes xs prefix of $j$
and $(i$, Store e1 mo2 $i) \in$ set (node-deliver-messages $x s$ )
and ( $r$, Expunge e mo $r$ ) $\in$ set (node-deliver-messages $x s$ )
shows mo2 $\neq r$
$\langle p r o o f\rangle$
lemma (in imap) concurrent-store-delete-independent-technical:
assumes $i \in i s$
and xs prefix of $j$
and $(i$, Store e mo $i) \in$ set (node-deliver-messages $x s)$
and (ir, Delete is e) $\in$ set (node-deliver-messages xs)
shows $h b$ ( $i$, Store e mo $i$ ) (ir, Delete is e)
$\langle p r o o f\rangle$
lemma (in imap) concurrent-append-delete-independent-technical:
assumes $i \in i s$
and xs prefix of $j$
and ( $i$, Append $i$ e) $\in$ set (node-deliver-messages $x s$ )
and (ir, Delete is e) $\in$ set (node-deliver-messages xs)
shows $h b$ ( $i$, Append $i$ e) (ir, Delete is e)
$\langle p r o o f\rangle$
lemma (in imap) concurrent-append-expunge-independent-technical:
assumes $i=m o$
and xs prefix of $j$
and $(i$, Append $i e) \in \operatorname{set}($ node-deliver-messages $x s)$
and $(r$, Expunge e mo $r) \in$ set (node-deliver-messages $x s$ )
shows $h b$ ( $i$, Append $i$ e) ( $r$, Expunge e mo $r$ )
〈proof〉
lemma (in imap) concurrent-append-store-independent-technical:
assumes $i=m o$
and xs prefix of $j$
and ( $i$, Append $i$ e) $\in$ set (node-deliver-messages $x s$ )
and $(r$, Store e mo $r) \in$ set (node-deliver-messages $x s$ )
shows $h b$ ( $i$, Append $i$ e) ( $r$, Store e mo $r$ )
〈proof〉
lemma（in imap）concurrent－expunge－delete－independent－technical：

```
assumes i\inis
    and xs prefix of j
    and (i, Expunge e mo i) \in set (node-deliver-messages xs)
    and (ir, Delete is e) \in set (node-deliver-messages xs)
    shows hb (i, Expunge e mo i) (ir, Delete is e)
\langleproof\rangle
lemma (in imap) concurrent-store-store-independent-technical:
    assumes xs prefix of j
    and (i,Store e mo i) \in set (node-deliver-messages xs)
    and (r, Store e i r) \in set (node-deliver-messages xs)
    shows hb (i, Store e mo i) (r, Store e i r)
<proof\rangle
lemma (in imap) expunge-delete-tag-causality:
    assumes i\inis
    and xs prefix of j
    and (i, Expunge e1 mo i) \in set (node-deliver-messages xs)
    and (ir, Delete is e2) \in set (node-deliver-messages xs)
    and pre@[Broadcast (ir,Delete is e2)] prefix of k
    shows Deliver (i, Expunge e2 mo i) \in set (history k)
<proof\rangle
lemma (in imap) expunge-delete-ids-imply-messages-same:
    assumes i\inis
    and xs prefix of j
    and (i, Expunge e1 mo i) \in set (node-deliver-messages xs)
    and (ir, Delete is e2) \in set (node-deliver-messages xs)
    shows e1 = e2
<proof\rangle
lemma (in imap) store-delete-ids-imply-messages-same:
    assumes i\inis
    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 e1 = e2
<proof\rangle
lemma (in imap) create-delete-ids-imply-messages-same:
    assumes i\inis
    and xs prefix of j
    and (i,Create i e1) \in set (node-deliver-messages xs)
    and (ir, Delete is e2) \in set (node-deliver-messages xs)
    shows e1 = e2
<proof\rangle
lemma (in imap) append-delete-ids-imply-messages-same:
    assumes i\inis
```

and $x s$ prefix of $j$
and（ $i$ ，Append $i$ e1）$\in$ set（node－deliver－messages $x s$ ）
and（ir，Delete is e2）$\in$ set（node－deliver－messages xs）
shows $e 1=e 2$
$\langle$ proof $\rangle$
lemma（in imap）append－expunge－ids－imply－messages－same：
assumes $i=m o$
and xs prefix of $j$
and（ $i$ ，Append i e1）$\in$ set（node－deliver－messages $x s$ ）
and（ $r$ ，Expunge e2 mo $r$ ）$\in$ set（node－deliver－messages xs）
shows $e 1=e 2$
$\langle p r o o f\rangle$
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 $x s$ ）
shows $e 1=e 2$
$\langle$ proof $\rangle$
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 ir）$\in$ set（node－deliver－messages xs）
shows $e 1=e 2$
〈proof〉
lemma（in imap）store－store－ids－imply－messages－same：
assumes $x s$ prefix of $j$
and（ $i$ ，Store e1 mo $i$ ）$\in$ set（node－deliver－messages $x s$ ）
and（ $r$ ，Store e2 $i r$ ）$\in$ set（node－deliver－messages xs）
shows $e 1=e 2$
＜proof〉
end

## 6 Convergence of the IMAP－CRDT

In this final section show that concurrent updates commute and thus Strong Eventual Conver－ gence 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 \(\neg h b(i\), Create \(i\) e1 \()(\) ir, Delete is e2)
    and \(\neg h b\) (ir, Delete is e2) ( \(i\), Create \(i\) e1)
    and xs prefix of \(j\)
    and \((i\), Create \(i\) e1 \() \in \operatorname{set}(\) node-deliver-messages \(x s)\)
    and (ir, Delete is eZ) \(\in\) set (node-deliver-messages xs)
    shows \(i \notin i s\)
    \(\langle p r o o f\rangle\)
```

corollary (in imap) concurrent-append-delete-independent:
assumes $\neg h b(i$, Append $i$ e1) $(i r$, Delete is e2)
and $\neg h b$ (ir, Delete is e2) ( $i$, Append $i$ e1)
and xs prefix of $j$
and $(i$, Append $i$ e1) $\in$ set (node-deliver-messages $x s)$
and (ir, Delete is e2) $\in$ set (node-deliver-messages xs)
shows $i \notin i s$
$\langle p r o o f\rangle$
corollary (in imap) concurrent-append-expunge-independent:
assumes $\neg h b(i$, Append $i$ e1) ( $r$, Expunge e2 mo $r$ )
and $\neg h b(r$, Expunge e2 mo $r)(i$, Append $i$ e1 $)$
and xs prefix of $j$
and $(i$, Append $i$ e1) $\in$ set (node-deliver-messages $x s)$
and $(r$, Expunge e2 mo $r) \in$ set (node-deliver-messages $x s)$
shows $i \neq m o$
$\langle p r o o f\rangle$
corollary (in imap) concurrent-append-store-independent:
assumes $\neg h b$ ( $i$, Append $i$ e1) ( $r$, Store e2 mo r)
and $\neg h b(r$, Store e2 mo $r)(i$, Append $i$ e1)
and $x s$ prefix of $j$
and $(i$, Append $i$ e1) $\in$ set (node-deliver-messages $x s$ )
and $(r$, Store e2 mo $r$ ) $\in$ set (node-deliver-messages $x s$ )
shows $i \neq m o$
$\langle p r o o f\rangle$
corollary (in imap) concurrent-expunge-delete-independent:
assumes $\neg h b$ ( $i$, Expunge e1 mo $i$ ) (ir, Delete is e2)
and $\neg h b$ (ir, Delete is e2) ( $i$, Expunge e1 mo i)
and xs prefix of $j$
and $(i$, Expunge e1 mo $i) \in$ set (node-deliver-messages $x s$ )
and (ir, Delete is e2) $\in$ set (node-deliver-messages xs)
shows $i \notin i s$
〈proof〉
corollary (in imap) concurrent-store-delete-independent:

```
assumes \(\neg h b\) ( \(i\), Store e1 mo \(i\) ) (ir, Delete is e2)
    and \(\neg h b\) (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 \(x s\) )
    and (ir, Delete is e2) \(\in\) set (node-deliver-messages xs)
shows \(i \notin i\) is
\(\langle p r o o f\rangle\)
```

corollary (in imap) concurrent-store-expunge-independent:
assumes $\neg h b(i$, Store e1 mo $i)(r$, Expunge e2 mo2 r)
and $\neg h b(r$, 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 $x s)$
shows $i \neq$ mo2 $\wedge r \neq m o$
$\langle p r o o f\rangle$
corollary (in imap) concurrent-store-store-independent:
assumes $\neg h b(i$, Store e1 mo $i)(r$, Store e2 mo2 r)
and $\neg h b(r$, 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 $x s$ )
and $(r$, Store e2 mo2 $r$ ) $\in$ set (node-deliver-messages $x s)$
shows $i \neq m o 2 \wedge r \neq m o$
$\langle p r o o f\rangle$
lemma (in imap) concurrent-operations-commute:
assumes xs prefix of $i$
shows hb.concurrent-ops-commute (node-deliver-messages xs)
$\langle p r o o f\rangle$
theorem (in imap) convergence:
assumes set (node-deliver-messages $x s)=$ set (node-deliver-messages ys)
and $x s$ prefix of $i$
and $y s$ prefix of $j$
shows apply-operations $x s=$ apply-operations $y s$
$\langle p r o o f\rangle$

## context imap begin

sublocale sec: strong-eventual-consistency weak-hb hb interp-msg $\lambda o p s . \exists x s i$. xs prefix of $i \wedge$ node-deliver-messages $x s=$ ops $\lambda x .(\{ \},\{ \})$ $\langle p r o o f\rangle$
end
end

## References

[1] V. B. F. Gomes, M. Kleppmann, D. P. Mulligan, and A. R. Beresford. A framework for establishing Strong Eventual Consistency for Conflict-free Replicated Datatypes. Archive of Formal Proofs, 2017. http://isa-afp.org/entries/CRDT.html.
[2] V. B. F. Gomes, M. Kleppmann, D. P. Mulligan, and A. R. Beresford. Verifying Strong Eventual Consistency in Distributed Systems. ArXiv e-prints, 2017.
[3] T. Jungnickel, L. Oldenburg, and M. Loibl. Designing a Planetary-Scale IMAP Service with Conflict-free Replicated Data Types. In 21th International Conference on Principles of Distributed Systems (OPODIS 2017), Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.
[4] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Technical report, 2011.
[5] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free Replicated Data Types. In International Symposium on Stabilization, Safety, and Security of Distributed Systems, SSS'11, pages 386-400, 2011.

