MODULE *LedgerChaining*

EXTENDS *Integers*, *FiniteSets*, *Sequences*, *SequencesExt*, *TLC*

This specification models how ledgers can be chained to form a single log.

It models a pool of clients ready to take leadership and perform writes to the segmented log. When a client gets elected as the leader, it:

1. closes the current ledger
2. creates a new ledger
3. appends the ledger to the ledger list
4. starts writing to the ledger

Leadership can change at anytime and the mechanics of leader election nor failure detection are not included.

the set of all clients

CONSTANTS *Clients*

the various states a client can be in (model values)

CONSTANTS *WAITING*,
GET_MD_FOR_CLOSING,
CLOSE_LAST_LEDGER,
PENDING_CREATE_LEDGER,
PENDING_APPEND_LEDGER,
HAS_OPEN_LEDGER

client state

VARIABLES *c_state*

metadata store state

VARIABLES *md_llist*,
md_llist_version,
md_ledgers,
md_leader,
md_next_lid

the ledgers written to bookies

VARIABLES *b_ledgers*

Auxilliary state

VARIABLES *next_entry_id* used for monotonicly increasing entry ids, needed for invariant checking

$vars \triangleq \langle c\_state, md\_llist, md\_llist\_version, md\_ledgers, md\_leader, md\_next\_lid, b\_ledgers, next\_entry\_id \rangle$

$NoLedgerMetadata \triangleq [id \mapsto 0, open \mapsto \text{FALSE}, version \mapsto -1]$

Starts with no leader and no ledgers

$Init \triangleq$
$\quad \wedge c\_state = [c \in Clients \mapsto [leader \mapsto \text{FALSE},$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad status \mapsto WAITING,$

```
                                    llist          ↦ ⟨⟩,
                                    llist_version  ↦ −1,
                                    ledger         ↦ NoLedgerMetadata]]
    ∧ md_leader = CHOOSE c ∈ Clients : TRUE
    ∧ md_llist_version = 0
    ∧ md_llist = ⟨⟩
    ∧ md_ledgers = ⟨⟩
    ∧ md_next_lid = 1
    ∧ b_ledgers = {}
    ∧ next_entry_id = 0
```

A leader is elected by the metadata store. We do not model why, how.

```
LeaderChosen(c) ≜
    ∧ md_leader ≠ c
    ∧ md_leader′ = c
    ∧ UNCHANGED ⟨c_state, md_llist, md_llist_version, md_ledgers, md_next_lid, b_ledgers, next_entry_id⟩
```

A client becomes aware it is the leader and assumes that role

```
BecomeLeader(c) ≜
    ∧ c_state[c].leader = FALSE
    ∧ md_leader = c
    ∧ c_state′ = [c_state EXCEPT ![c].leader = TRUE,
                                 ![c].status = GET_MD_FOR_CLOSING]
    ∧ UNCHANGED ⟨md_llist, md_llist_version, md_ledgers, md_leader, md_next_lid, b_ledgers, next_entry_i
```

A client that believes it is the leader becomes aware it is not the leader anymore and returns to *WAITING*. It does not need to close the ledger, though it could, as another client becoming the leader will do that.

```
Abdicate(c) ≜
    ∧ c_state[c].leader = TRUE
    ∧ md_leader ≠ c
    ∧ c_state′ = [c_state EXCEPT ![c].leader = FALSE,
                                 ![c].status = WAITING]
    ∧ UNCHANGED ⟨md_llist, md_llist_version, md_ledgers, md_leader, md_next_lid, b_ledgers, next_entry_i
```

A newly elected leader must first obtain the metadata of the ledger list and the last ledger in the ledger list. If the ledger list is empty, it transitions to the *PENDING_CREATE_LEDGER* state, else moves to the *PENDING_CREATE_LEDGER* state. The version of the ledger list is cached now which is important as any change made by another client in the meantime will be detected.

```
GetLastLedger(c) ≜
    ∧ c_state[c].leader = TRUE
    ∧ c_state[c].status = GET_MD_FOR_CLOSING
    ∧ ∨ ∧ md_llist ≠ ⟨⟩
          ∧ c_state′ = [c_state EXCEPT ![c].llist_version = md_llist_version,
                                       ![c].llist        = md_llist,
```

```
                                        ![c].ledger      = md_ledgers[Last(md_llist)],
                                        ![c].status      = CLOSE_LAST_LEDGER]
    ∨ ∧ md_llist = ⟨⟩
      ∧ c_state' = [c_state EXCEPT ![c].llist_version = md_llist_version,
                                   ![c].llist         = md_llist,
                                   ![c].status        = PENDING_CREATE_LEDGER]
  ∧ UNCHANGED ⟨md_llist, md_llist_version, md_ledgers, md_leader, md_next_lid, b_ledgers, next_entry_id⟩
```

The client leader sees that the last ledger is already closed, so moves to the *PENDING_CREATE_LEDGER* state.

```
LedgerAlreadyClosed(c) ≜
  ∧ c_state[c].leader = TRUE
  ∧ c_state[c].status = CLOSE_LAST_LEDGER
  ∧ c_state[c].ledger.open = FALSE
  ∧ c_state' = [c_state EXCEPT ![c].status = PENDING_CREATE_LEDGER]
  ∧ UNCHANGED ⟨md_llist, md_llist_version, md_ledgers, md_leader, md_next_lid, b_ledgers, next_entry_id⟩
```

The client leader closes the last ledger.

When fencing the ledger, if the ledger does not exist in the bookie then it gets created and fenced.

```
FenceLedger(ledger_id) ≜
  IF ∃ ledger ∈ b_ledgers : ledger.id = ledger_id
  THEN b_ledgers' = {IF l.id = ledger_id
                     THEN [l EXCEPT !.fenced = TRUE]
                     ELSE l : l ∈ b_ledgers}
  ELSE b_ledgers' = b_ledgers ∪ {[id      ↦ ledger_id,
                                  entry   ↦ −1,
                                  fenced ↦ TRUE]}

CloseLastLedgerSuccess(c) ≜
  ∧ c_state[c].leader = TRUE
  ∧ c_state[c].status = CLOSE_LAST_LEDGER
  ∧ c_state[c].ledger.open = TRUE
  ∧ LET ledger_id ≜ c_state[c].ledger.id
    IN
        ∧ c_state[c].ledger.version = md_ledgers[ledger_id].version
        ∧ md_ledgers' = [md_ledgers EXCEPT ![ledger_id].open = FALSE,
                                           ![ledger_id].version = @ + 1]
        ∧ c_state' = [c_state EXCEPT ![c].status = PENDING_CREATE_LEDGER]
        ∧ FenceLedger(ledger_id)
  ∧ UNCHANGED ⟨md_llist, md_llist_version, md_leader, md_next_lid, next_entry_id⟩
```

The client leader tries to close the last ledger, but another client has updated the ledger metadata in the meantime. The leader backs off and returns to the *GET_MD_FOR_CLOSING* state.

```
CloseLastLedgerBadVersion(c) ≜
    ∧ c_state[c].leader = TRUE
    ∧ c_state[c].status = CLOSE_LAST_LEDGER
    ∧ c_state[c].ledger.open = TRUE
    ∧ c_state[c].ledger.version < md_ledgers[c_state[c].ledger.id].version
    ∧ c_state' = [c_state EXCEPT ![c].status = GET_MD_FOR_CLOSING]
    ∧ UNCHANGED ⟨md_llist, md_llist_version, md_ledgers, md_leader, md_next_lid, b_ledgers, next_entry_i
```

The client leader creates a new ledger.

```
CreateLedger(c) ≜
    ∧ c_state[c].leader = TRUE
    ∧ c_state[c].status = PENDING_CREATE_LEDGER
    ∧ LET next_ledger ≜ [id      ↦ md_next_lid,
                         open    ↦ TRUE,
                         version ↦ 0]
      IN ∧ c_state' = [c_state EXCEPT ![c].ledger = next_ledger,
                                      ![c].status = PENDING_APPEND_LEDGER]
         ∧ md_ledgers' = md_ledgers @@ (next_ledger.id :> next_ledger)
         ∧ md_next_lid' = md_next_lid + 1
         ∧ UNCHANGED ⟨md_llist, md_llist_version, md_leader, b_ledgers, next_entry_id⟩
```

The client leader appends the new ledger to the ledger list. It uses the cached metadata from when it obtained the ledger list. The version in the metadata store has not changed so the update operation succeeds.

```
AppendLedgerSuccess(c) ≜
    ∧ c_state[c].leader = TRUE
    ∧ c_state[c].status = PENDING_APPEND_LEDGER
    ∧ c_state[c].llist_version = md_llist_version
    ∧ LET new_list     ≜ Append(c_state[c].llist, c_state[c].ledger.id)
          new_version  ≜ md_llist_version + 1
      IN
        ∧ md_llist' = new_list
        ∧ md_llist_version' = new_version
        ∧ c_state' = [c_state EXCEPT ![c].status = HAS_OPEN_LEDGER,
                                     ![c].llist = new_list,
                                     ![c].llist_version = new_version]
        ∧ UNCHANGED ⟨md_ledgers, md_leader, md_next_lid, b_ledgers, next_entry_id⟩
```

The client leader tries to append the new ledger to the ledger list. It uses the cached metadata from when it obtained the ledger list. But the version in the metadata store has changed indicating that another client has also appended a ledger in the meantime. The client abdicates.

```
AppendLedgerBadVersion(c) ≜
    ∧ c_state[c].leader = TRUE
    ∧ c_state[c].status = PENDING_APPEND_LEDGER
    ∧ c_state[c].llist_version < md_llist_version
    ∧ c_state' = [c_state EXCEPT ![c].leader = FALSE,
```

```
                                        ![c].status = WAITING]
    ∧ UNCHANGED ⟨md_llist, md_llist_version, md_ledgers, md_leader, md_next_lid, b_ledgers, next_entry_i
```

The client leader writes to the ledger. We only model a single entry as this is enough for verifying ordering/data loss and keeps the state space small. We have already modelled multiple entries in the *BookKeeperProtocol* spec.

```
WriteToLedger(c) ≜
    ∧ c_state[c].leader = TRUE
    ∧ c_state[c].status = HAS_OPEN_LEDGER
    ∧ ¬∃ ledger ∈ b_ledgers : ledger.id = c_state[c].ledger.id
    ∧ b_ledgers' = b_ledgers ∪ {[id      ↦ c_state[c].ledger.id,
                                 entry   ↦ next_entry_id,
                                 fenced  ↦ FALSE]}
    ∧ next_entry_id' = next_entry_id + 1
    ∧ UNCHANGED ⟨c_state, md_llist, md_llist_version, md_ledgers, md_leader, md_next_lid⟩
```

A client leader closes its own ledger and transitions to the *PENDING_CREATE_LEDGER* state so that it cannot chain a new ledger onto the list.

```
CloseOwnLedgerSuccess(c) ≜
    ∧ c_state[c].leader = TRUE
    ∧ c_state[c].status = HAS_OPEN_LEDGER
    ∧ LET ledger ≜ c_state[c].ledger
      IN
          ∧ ∃ l ∈ b_ledgers : l.id = ledger.id
          ∧ ledger.version = md_ledgers[ledger.id].version
          ∧ md_ledgers' = [md_ledgers EXCEPT ![ledger.id].open = FALSE,
                                             ![ledger.id].version = @ + 1]
          ∧ c_state' = [c_state EXCEPT ![c].status = PENDING_CREATE_LEDGER]
    ∧ UNCHANGED ⟨md_llist, md_llist_version, md_leader, md_next_lid, b_ledgers, next_entry_id⟩
```

A client leader tries to close its own ledger but can't as ledger metadata was previously updated by a different client.

```
CloseOwnLedgerBadVersion(c) ≜
    ∧ c_state[c].leader = TRUE
    ∧ c_state[c].status = HAS_OPEN_LEDGER
    ∧ LET ledger ≜ c_state[c].ledger
      IN
          ∧ ∃ l ∈ b_ledgers : l.id = ledger.id
          ∧ ledger.version < md_ledgers[ledger.id].version
          ∧ c_state' = [c_state EXCEPT ![c].leader = FALSE,
                                       ![c].status = WAITING]
    ∧ UNCHANGED ⟨md_llist, md_llist_version, md_ledgers, md_leader, md_next_lid, b_ledgers, next_entry_i


Next ≜
    ∃ c ∈ Clients :
```

```
        ∨ LeaderChosen(c)
        ∨ BecomeLeader(c)
        ∨ Abdicate(c)
        ∨ GetLastLedger(c)
        ∨ LedgerAlreadyClosed(c)
        ∨ CloseLastLedgerSuccess(c)
        ∨ CloseLastLedgerBadVersion(c)
        ∨ CreateLedger(c)
        ∨ AppendLedgerSuccess(c)
        ∨ AppendLedgerBadVersion(c)
        ∨ WriteToLedger(c)
        ∨ CloseOwnLedgerSuccess(c)
        ∨ CloseOwnLedgerBadVersion(c)
```

Types

```
ClientStatuses ≜ {
            WAITING,
            GET_MD_FOR_CLOSING,
            CLOSE_LAST_LEDGER,
            PENDING_CREATE_LEDGER,
            PENDING_APPEND_LEDGER,
            HAS_OPEN_LEDGER}

LedgerMetadata ≜ [id : Nat, open : BOOLEAN , version : Nat ∪ {−1}]
Ledger   ≜ [id : Nat, entry : Nat ∪ {−1}, fenced       : BOOLEAN ]

Client   ≜ [leader : BOOLEAN ,
            status : ClientStatuses,
            llist_version : Nat ∪ {−1},
            llist : Seq(Nat),
            ledger : LedgerMetadata]

TypeOK ≜
    ∧   c_state ∈ [Clients → Client]
    ∧   md_leader ∈ Clients
    ∧   md_llist_version ∈ Nat
    ∧   ∨ md_next_lid = 1
        ∨ ∧ md_next_lid > 1
          ∧ md_ledgers ∈ [1 .. (md_next_lid − 1) → LedgerMetadata]
          ∧ b_ledgers ∈ SUBSET Ledger
    ∧   md_next_lid ∈ Nat
```

Invariant: No ledgers that were written to ended up outside of the ledger list.

```
AllNonEmptyLedgersInLedgerList ≜
    IF b_ledgers ≠ {}
     THEN
```

```
      ¬∃ ledger_id ∈ 1 .. (md_next_lid − 1) :
          ∧ ∃ ledger ∈ b_ledgers : ledger.id = ledger_id
          ∧ ∨ md_llist = ⟨⟩
            ∨ ¬∃ mdl ∈ DOMAIN md_llist : ledger_id = md_llist[mdl]
   ELSE TRUE
```

Invariant: The entries written across the ledgers maintain temporal ordering

```
EntryOrderMaintained ≜
   ∀ l1 ∈ b_ledgers :
      ∧ ¬∃ l2 ∈ b_ledgers :
          ∧ l1.id < l2.id
          ∧ l1.entry > l2.entry
          \* neither is an empty fenced ledger
          ∧ l1.entry ≠ −1
          ∧ l2.entry ≠ −1
```

Invariant: There cannot be more than one open ledger in the ledger list at anytime.

```
OnlyOneLedgerOpenAtATime ≜
   IF md_llist ≠ ⟨⟩
   THEN ¬∃ l1, l2 ∈ DOMAIN md_llist :
             ∧ l1 ≠ l2
             ∧ md_ledgers[l1].open = TRUE
             ∧ md_ledgers[l2].open = TRUE
   ELSE TRUE
```

Constraints

```
LedgerLimit ≜ md_next_lid < 4
```

* Modification History
* *Last* modified *Thu Apr* 01 14:16:08 *CEST* 2021 by *jvanlightly*
* Created *Thu Apr* 01 12:05:02 *CEST* 2021 by *jvanlightly*