MODULE *DistributedTransaction*

EXTENDS *Integers*, *FiniteSets*

The set of all keys.

CONSTANTS *KEY*

The sets of optimistic clients and pessimistic clients.

CONSTANTS *OPTIMISTIC_CLIENT*, *PESSIMISTIC_CLIENT*

$CLIENT \triangleq PESSIMISTIC\_CLIENT \cup OPTIMISTIC\_CLIENT$

CLIENT_KEY is a set of [Client → SUBSET *KEY*]
representing the involved keys of each client.

CONSTANTS *CLIENT_KEY*

ASSUME $\forall c \in CLIENT : CLIENT\_KEY[c] \subseteq KEY$

CLIENT_PRIMARY is the primary key of each client.

CONSTANTS *CLIENT_PRIMARY*

ASSUME $\forall c \in CLIENT : CLIENT\_PRIMARY[c] \in CLIENT\_KEY[c]$

Timestamp of transactions.

$Ts \triangleq Nat \setminus \{0\}$

$NoneTs \triangleq 0$

The algorithm is easier to understand in terms of the set of *msgs* of all messages that have ever been sent. A more accurate model would use one or more variables to represent the messages actually in transit, and it would include actions representing message loss and duplication as well as message receipt.

In the current spec, there is no need to model message loss because we are mainly concerned with the algorithm's safety property. The safety part of the spec says only what messages may be received and does not assert that any message actually is received. Thus, there is no difference between a lost message and one that is never received.

VARIABLES *req_msgs*

VARIABLES *resp_msgs*

key_data[*k*] is the set of multi-version data of the key. Since we don't care about the concrete value of data, a *start_ts* is sufficient to represent one data version.

VARIABLES *key_data*

key_lock[*k*] is the set of lock (zero or one element). A lock is of a record of [*ts*: *start_ts*, primary: key, type: *lock_type*]. If primary equals to *k*, it is a primary lock, otherwise secondary lock. *lock_type* is one of {"prewrite_optimistic", "prewrite_pessimistic", "lock_key"}. *lock_key* denotes the pessimistic lock performed by *ServerLockKey* action, the *prewrite_pessimistic* denotes percolator optimistic lock who is transformed from a *lock_key* lock by action *ServerPrewritePessimistic*, and *prewrite_optimistic* denotes the classic optimistic lock.

In *TiKV*, *key_lock* has an additional *for_update_ts* field and the *LockType* is of four variants: $\{$"*PUT*", "*DELETE*", "*LOCK*", "*PESSIMISTIC*"$\}$.

In the spec, we abstract them by:
(1) $LockType \in \{\text{"PUT"}, \text{"DELETE"}, \text{"LOCK"}\} \land for\_update\_ts = 0 \equiv$
$type =$ "prewrite_optimistic"
(2) $LockType \in \{\text{"PUT"}, \text{"DELETE"}\} \land for\_update\_ts > 0 \equiv$
$type =$ "prewrite_pessimistic"
(3) $LockType =$ "PESSIMISTIC" $\equiv type =$ "lock_key"

VARIABLES key_lock

key_write[*k*] is a sequence of commit or rollback record of the key. It's a record of [*ts*, *start_ts*, type, [protected]]. type can be either "write" or "rollback". *ts* represents the *commit_ts* of "write" record. Otherwise, *ts* equals to *start_ts* on "rollback" record. "rollback" record has an additional protected field. protected signifies the rollback record would not be collapsed.

VARIABLES key_write

client_state[*c*] indicates the current transaction stage of client *c*.

VARIABLES $client\_state$

client_ts[*c*] is a record of [*start_ts*, *commit_ts*, *for_update_ts*]. Fields are all initialized to *NoneTs*.

VARIABLES $client\_ts$

client_key[*c*] is a record of [locking: {*key*}, *prewriting*: {*key*}]. Hereby, "locking" denotes the keys whose pessimistic locks haven't been acquired, "prewriting" denotes the keys that are pending for prewrite.

VARIABLES $client\_key$

next_ts is a globally monotonically increasing integer, representing the virtual clock of transactions. In practice, the variable is maintained by *PD*, the time oracle of a cluster.

VARIABLES $next\_ts$

$$msg\_vars \triangleq \langle req\_msgs, resp\_msgs \rangle$$
$$client\_vars \triangleq \langle client\_state, client\_ts, client\_key \rangle$$
$$key\_vars \triangleq \langle key\_data, key\_lock, key\_write \rangle$$
$$vars \triangleq \langle msg\_vars, client\_vars, key\_vars, next\_ts \rangle$$

$$SendReqs(msgs) \triangleq req\_msgs' = req\_msgs \cup msgs$$
$$SendResp(msg) \triangleq resp\_msgs' = resp\_msgs \cup \{msg\}$$

Type Definitions

$ReqMessages \triangleq$

$[start\_ts : Ts, primary : KEY, type : \{\text{"lock\_key"}\}, key : KEY, for\_update\_ts : Ts]$
$\cup [start\_ts : Ts, primary : KEY, type : \{\text{"prewrite\_optimistic"}\}, key : KEY]$
$\cup [start\_ts : Ts, primary : KEY, type : \{\text{"prewrite\_pessimistic"}\}, key : KEY]$
$\cup [start\_ts : Ts, primary : KEY, type : \{\text{"commit"}\}, commit\_ts : Ts]$
$\cup [start\_ts : Ts, primary : KEY, type : \{\text{"resolve\_rollbacked"}\}]$
$\cup [start\_ts : Ts, primary : KEY, type : \{\text{"resolve\_committed"}\}, commit\_ts : Ts]$

In $TiKV$, there's an extra flag "*rollback_if_not_exist*" in the *check_txn_status* request. If the primary key lock is missing (and no record in the write column), there are two cases: the prewrite request of the primary key is delayed, or is lost due to client ($TiDB$) crash. To distinguish these two, it must wait until the TTL of the lock to be resolved expired. In the TLA+ spec, the TTL is considered constantly expired when the action is taken, so there's no need to model the flag.

$\cup [start\_ts : Ts, primary : KEY, type : \{\text{"check\_txn\_status"}\}, resolving\_pessimistic\_lock : \text{BOOLEAN}]$

$RespMessages \triangleq$

$[start\_ts : Ts, type : \{\text{"prewrited"}, \text{"locked\_key"}\}, key : KEY]$
$\cup [start\_ts : Ts, type : \{\text{"lock\_failed"}\}, key : KEY, latest\_commit\_ts : Ts]$
$\cup [start\_ts : Ts, type : \{\text{"committed"}, \text{"commit\_aborted"}, \text{"prewrite\_aborted"}, \text{"lock\_key\_aborted"}\}]$

$TypeOK \triangleq$
$\wedge\ req\_msgs \in \text{SUBSET } ReqMessages$
$\wedge\ resp\_msgs \in \text{SUBSET } RespMessages$
$\wedge\ key\_data \in [KEY \to \text{SUBSET } Ts]$
$\wedge\ key\_lock \in [KEY \to \text{SUBSET } [ts : Ts, primary : KEY, type : \{\text{"prewrite\_optimistic"}, \text{"prewrite\_pessimistic"}, \text{"lock\_key"}\}]]$

At most one lock in $key\_lock[k]$

$\wedge\ \forall k \in KEY : Cardinality(key\_lock[k]) \leq 1$
$\wedge\ key\_write \in [KEY \to \text{SUBSET } ([ts : Ts, start\_ts : Ts, type : \{\text{"write"}\}] \cup [ts : Ts, start\_ts : Ts, type : \{\text{"rollback"}\}, protected : \text{BOOLEAN}])]$
$\wedge\ client\_state \in [CLIENT \to \{\text{"init"}, \text{"locking"}, \text{"prewriting"}, \text{"committing"}\}]$
$\wedge\ client\_ts \in [CLIENT \to [start\_ts : Ts \cup \{NoneTs\}, commit\_ts : Ts \cup \{NoneTs\}, for\_update\_ts : Ts \cup \{NoneTs\}]]$
$\wedge\ client\_key \in [CLIENT \to [locking : \text{SUBSET } KEY, prewriting : \text{SUBSET } KEY]]$
$\wedge\ next\_ts \in Ts$

Client Actions

```
ClientLockKey(c) ≜
  ∧ client_state[c] = "init"
  ∧ client_state' = [client_state EXCEPT ![c] = "locking"]
  ∧ client_ts' = [client_ts EXCEPT ![c].start_ts = next_ts, ![c].for_update_ts = next_ts]
  ∧ next_ts' = next_ts + 1
   Assume we need to acquire pessimistic locks for all keys
  ∧ client_key' = [client_key EXCEPT ![c].locking = CLIENT_KEY[c]]
  ∧ SendReqs({[type ↦ "lock_key",
               start_ts ↦ client_ts'[c].start_ts,
               primary ↦ CLIENT_PRIMARY[c],
               key ↦ k,
               for_update_ts ↦ client_ts'[c].for_update_ts] : k ∈ CLIENT_KEY[c]})
  ∧ UNCHANGED ⟨resp_msgs, key_vars⟩

ClientLockedKey(c) ≜
  ∧ client_state[c] = "locking"
  ∧ ∃ resp ∈ resp_msgs :
      ∧ resp.type = "locked_key"
      ∧ resp.start_ts = client_ts[c].start_ts
      ∧ resp.key ∈ client_key[c].locking
      ∧ client_key' = [client_key EXCEPT ![c].locking = @ \ {resp.key}]
      ∧ UNCHANGED ⟨msg_vars, key_vars, client_ts, client_state, next_ts⟩

ClientRetryLockKey(c) ≜
  ∧ client_state[c] = "locking"
  ∧ ∃ resp ∈ resp_msgs :
      ∧ resp.type = "lock_failed"
      ∧ resp.start_ts = client_ts[c].start_ts
      ∧ resp.latest_commit_ts > client_ts[c].for_update_ts
      ∧ client_ts' = [client_ts EXCEPT ![c].for_update_ts = resp.latest_commit_ts]
      ∧ SendReqs({[type ↦ "lock_key",
                   start_ts ↦ client_ts'[c].start_ts,
                   primary ↦ CLIENT_PRIMARY[c],
                   key ↦ resp.key,
                   for_update_ts ↦ client_ts'[c].for_update_ts]})
      ∧ UNCHANGED ⟨resp_msgs, key_vars, client_state, client_key, next_ts⟩

ClientPrewritePessimistic(c) ≜
  ∧ client_state[c] = "locking"
  ∧ client_key[c].locking = {}
  ∧ client_state' = [client_state EXCEPT ![c] = "prewriting"]
  ∧ client_key' = [client_key EXCEPT ![c].prewriting = CLIENT_KEY[c]]
  ∧ SendReqs({[type ↦ "prewrite_pessimistic",
```

$$
\begin{array}{l}
\quad\quad\quad\quad\quad\quad\quad start\_ts \mapsto client\_ts[c].start\_ts, \\
\quad\quad\quad\quad\quad\quad\quad primary \mapsto CLIENT\_PRIMARY[c], \\
\quad\quad\quad\quad\quad\quad\quad key \mapsto k] : k \in CLIENT\_KEY[c]\}) \\
\quad \land \text{UNCHANGED } \langle resp\_msgs, key\_vars, client\_ts, next\_ts \rangle
\end{array}
$$

$$
\begin{array}{l}
ClientPrewriteOptimistic(c) \triangleq \\
\quad \land client\_state[c] = \text{"init"} \\
\quad \land client\_state' = [client\_state \text{ EXCEPT } ![c] = \text{"prewriting"}] \\
\quad \land client\_ts' = [client\_ts \text{ EXCEPT } ![c].start\_ts = next\_ts] \\
\quad \land next\_ts' = next\_ts + 1 \\
\quad \land client\_key' = [client\_key \text{ EXCEPT } ![c].prewriting = CLIENT\_KEY[c]] \\
\quad \land SendReqs(\{[type \mapsto \text{"prewrite\_optimistic"}, \\
\quad\quad\quad\quad\quad\quad\quad start\_ts \mapsto client\_ts'[c].start\_ts, \\
\quad\quad\quad\quad\quad\quad\quad primary \mapsto CLIENT\_PRIMARY[c], \\
\quad\quad\quad\quad\quad\quad\quad key \mapsto k] : k \in CLIENT\_KEY[c]\}) \\
\quad \land \text{UNCHANGED } \langle resp\_msgs, key\_vars \rangle
\end{array}
$$

$$
\begin{array}{l}
ClientPrewrited(c) \triangleq \\
\quad \land client\_state[c] = \text{"prewriting"} \\
\quad \land client\_key[c].locking = \{\} \\
\quad \land \exists\, resp \in resp\_msgs : \\
\quad\quad \land resp.type = \text{"prewrited"} \\
\quad\quad \land resp.start\_ts = client\_ts[c].start\_ts \\
\quad\quad \land resp.key \in client\_key[c].prewriting \\
\quad\quad \land client\_key' = [client\_key \text{ EXCEPT } ![c].prewriting = @ \setminus \{resp.key\}] \\
\quad\quad \land \text{UNCHANGED } \langle msg\_vars, key\_vars, client\_ts, client\_state, next\_ts \rangle
\end{array}
$$

$$
\begin{array}{l}
ClientCommit(c) \triangleq \\
\quad \land client\_state[c] = \text{"prewriting"} \\
\quad \land client\_key[c].prewriting = \{\} \\
\quad \land client\_state' = [client\_state \text{ EXCEPT } ![c] = \text{"committing"}] \\
\quad \land client\_ts' = [client\_ts \text{ EXCEPT } ![c].commit\_ts = next\_ts] \\
\quad \land next\_ts' = next\_ts + 1 \\
\quad \land SendReqs(\{[type \mapsto \text{"commit"}, \\
\quad\quad\quad\quad\quad\quad\quad start\_ts \mapsto client\_ts'[c].start\_ts, \\
\quad\quad\quad\quad\quad\quad\quad primary \mapsto CLIENT\_PRIMARY[c], \\
\quad\quad\quad\quad\quad\quad\quad commit\_ts \mapsto client\_ts'[c].commit\_ts]\}) \\
\quad \land \text{UNCHANGED } \langle resp\_msgs, key\_vars, client\_key \rangle
\end{array}
$$

Server Actions

Write the write column and unlock the lock iff the lock exists.

$$
\begin{array}{l}
commit(pk, start\_ts, commit\_ts) \triangleq \\
\quad \exists\, l \in key\_lock[pk] : \\
\quad\quad \land l.ts = start\_ts \\
\quad\quad \land key\_lock' = [key\_lock \text{ EXCEPT } ![pk] = \{\}]
\end{array}
$$

$\land\ key\_write' = [key\_write\ \text{EXCEPT}\ ![pk] = @ \cup \{[ts \mapsto commit\_ts,$
$\qquad type \mapsto \text{"write"},$
$\qquad start\_ts \mapsto start\_ts]\}]$

Rollback the transaction that starts at $start\_ts$ on key k.

$rollback(k,\ start\_ts)\ \triangleq$
LET

Rollback record on the primary key of a pessimistic transaction needs to be protected from being collapsed. If we can't decide whether it suffices that because the lock is missing or mismatched, it should also be protected.

$protected\ \triangleq\ \lor\ \exists\, l \in key\_lock[k] :$
$\qquad \land\ l.ts = start\_ts$
$\qquad \land\ l.primary = k$
$\qquad \land\ l.type \in \{\text{"lock\_key"},\ \text{"prewrite\_pessimistic"}\}$
$\quad \lor\ \exists\, l \in key\_lock[k] : l.ts \neq start\_ts$
$\quad \lor\ key\_lock[k] = \{\}$

IN

If a lock exists and has the same ts, unlock it.

$\land\ \text{IF}\ \exists\, l \in key\_lock[k] : l.ts = start\_ts$
$\quad \text{THEN}\ key\_lock' = [key\_lock\ \text{EXCEPT}\ ![k] = \{\}]$
$\quad \text{ELSE}\ \ \text{UNCHANGED}\ key\_lock$
$\land\ key\_data' = [key\_data\ \text{EXCEPT}\ ![k] = @ \setminus \{start\_ts\}]$
$\land\ \text{IF}$
$\qquad \land\ \neg\exists\, w \in key\_write[k] : w.ts = start\_ts$
$\quad \text{THEN}$
$\qquad key\_write' = [key\_write\ \text{EXCEPT}$
$\qquad\quad ![k] =$

collapse rollback

$\qquad\quad (@ \setminus \{w \in @ : w.type = \text{"rollback"} \land \neg w.protected \land w.ts < start\_ts\})$

write rollback record

$\qquad\quad \cup \{[ts \mapsto start\_ts,$
$\qquad\qquad start\_ts \mapsto start\_ts,$
$\qquad\qquad type \mapsto \text{"rollback"},$
$\qquad\qquad protected \mapsto protected]\}]$
$\quad \text{ELSE}$
$\qquad \text{UNCHANGED}\ \langle key\_write \rangle$

$ServerLockKey\ \triangleq$
$\exists\, req \in req\_msgs :$
$\quad \land\ req.type = \text{"lock\_key"}$
$\quad \land\ \text{LET}$
$\qquad k\ \triangleq\ req.key$
$\qquad start\_ts\ \triangleq\ req.start\_ts$
$\quad\ \ \text{IN}$

Pessimistic lock is allowed only if no stale lock exists. If there is one, wait until *ServerCleanupStaleLock* to clean it up.

$\land$ $key\_lock[k] = \{\}$

$\land$ LET

$latest\_write \triangleq \{w \in key\_write[k] : \forall w2 \in key\_write[k] : w.ts \geq w2.ts\}$

$all\_commits \triangleq \{w \in key\_write[k] : w.type = \text{"write"}\}$

$latest\_commit \triangleq \{w \in all\_commits : \forall w2 \in all\_commits : w.ts \geq w2.ts\}$

IN

IF $\exists w \in key\_write[k] : w.start\_ts = start\_ts \land w.type = \text{"rollback"}$

THEN

If corresponding rollback record is found, which indicates that the transcation is rollbacked, abort the transaction.

$\land$ $SendResp([start\_ts \mapsto start\_ts,\ type \mapsto \text{"lock\_key\_aborted"}])$

$\land$ UNCHANGED $\langle req\_msgs,\ client\_vars,\ key\_vars,\ next\_ts\rangle$

ELSE

Acquire pessimistic lock only if *for_update_ts* of *req* is greater or equal to the latest "write" record. Because if the latest record is "write", it means that a new version is committed after *for_update_ts*, which violates Read Committed guarantee.

$\lor$ $\land \neg\exists w \in latest\_commit : w.ts > req.for\_update\_ts$

$\land$ $key\_lock' = [key\_lock \text{ EXCEPT } ![k] = \{[ts \mapsto start\_ts,\ primary \mapsto req.primary,\ type \mapsto \text{"lock\_key"}]\}]$

$\land$ $SendResp([start\_ts \mapsto start\_ts,\ type \mapsto \text{"locked\_key"},\ key \mapsto k])$

$\land$ UNCHANGED $\langle req\_msgs,\ client\_vars,\ key\_data,\ key\_write,\ next\_ts\rangle$

Otherwise, reject the request and let client to retry with new *for_update_ts*.

$\lor$ $\exists w \in latest\_commit :$

$\land$ $w.ts > req.for\_update\_ts$

$\land$ $SendResp([start\_ts \mapsto start\_ts,\ type \mapsto \text{"lock\_failed"},\ key \mapsto k,\ latest\_commit\_ts \mapsto w.ts])$

$\land$ UNCHANGED $\langle req\_msgs,\ client\_vars,\ key\_vars,\ next\_ts\rangle$

$ServerPrewritePessimistic \triangleq$

$\exists req \in req\_msgs :$

$\land$ $req.type = \text{"prewrite\_pessimistic"}$

$\land$ LET

$k \triangleq req.key$

$start\_ts \triangleq req.start\_ts$

IN

Pessimistic prewrite is allowed only if pressimistic lock is acquired, otherwise abort the transaction.

$\land$ IF $\exists\, l \in key\_lock[k] : l.ts = start\_ts$
  THEN
    $\land$ $key\_lock' = [key\_lock \text{ EXCEPT } ![k] = \{[ts \mapsto start\_ts,$
    $\quad primary \mapsto req.primary,$
    $\quad type \mapsto \text{“prewrite\_pessimistic”}]\}]$
    $\land$ $key\_data' = [key\_data \text{ EXCEPT } ![k] = @ \cup \{start\_ts\}]$
    $\land$ $SendResp([start\_ts \mapsto start\_ts,\ type \mapsto \text{“prewrited”},\ key \mapsto k])$
    $\land$ UNCHANGED $\langle req\_msgs,\ client\_vars,\ key\_write,\ next\_ts\rangle$
  ELSE
    $\land$ $SendResp([start\_ts \mapsto start\_ts,\ type \mapsto \text{“prewrite\_aborted”}])$
    $\land$ UNCHANGED $\langle req\_msgs,\ client\_vars,\ key\_vars,\ next\_ts\rangle$

$ServerPrewriteOptimistic \triangleq$
$\exists\, req \in req\_msgs :$
  $\land$ $req.type = \text{“prewrite\_optimistic”}$
  $\land$ LET
    $k \triangleq req.key$
    $start\_ts \triangleq req.start\_ts$
  IN
  $\land$ IF $\exists\, w \in key\_write[k] : w.ts \geq start\_ts$
    THEN
      $\land$ $SendResp([start\_ts \mapsto start\_ts,\ type \mapsto \text{“prewrite\_aborted”}])$
      $\land$ UNCHANGED $\langle req\_msgs,\ client\_vars,\ key\_vars,\ next\_ts\rangle$
    ELSE

Optimistic prewrite is allowed only if no stale lock exists. If there is one, wait until *ServerCleanupStaleLock* to clean it up.

      $\land$ $\lor$ $key\_lock[k] = \{\}$
        $\lor$ $\exists\, l \in key\_lock[k] : l.ts = start\_ts$
      $\land$ $key\_lock' = [key\_lock \text{ EXCEPT } ![k] = \{[ts \mapsto start\_ts,$
      $\quad primary \mapsto req.primary,$
      $\quad type \mapsto \text{“prewrite\_optimistic”}]\}]$
      $\land$ $key\_data' = [key\_data \text{ EXCEPT } ![k] = @ \cup \{start\_ts\}]$
      $\land$ $SendResp([start\_ts \mapsto start\_ts,\ type \mapsto \text{“prewrited”},\ key \mapsto k])$
      $\land$ UNCHANGED $\langle req\_msgs,\ client\_vars,\ key\_write,\ next\_ts\rangle$

$ServerCommit \triangleq$
$\exists\, req \in req\_msgs :$
  $\land$ $req.type = \text{“commit”}$
  $\land$ LET
    $pk \triangleq req.primary$
    $start\_ts \triangleq req.start\_ts$
  IN
  IF $\exists\, w \in key\_write[pk] : w.start\_ts = start\_ts \land w.type = \text{“write”}$

THEN

Key has already been committed. Do nothing.

$\land$ $SendResp([start\_ts \mapsto start\_ts,\ type \mapsto \text{"committed"}])$

$\land$ UNCHANGED $\langle req\_msgs,\ client\_vars,\ key\_vars,\ next\_ts \rangle$

ELSE

IF $\exists\, l \in key\_lock[pk] : l.ts = start\_ts$

THEN

Commit the key only if the prewrite lock exists.

$\land$ $commit(pk,\ start\_ts,\ req.commit\_ts)$

$\land$ $SendResp([start\_ts \mapsto start\_ts,\ type \mapsto \text{"committed"}])$

$\land$ UNCHANGED $\langle req\_msgs,\ client\_vars,\ key\_data,\ next\_ts \rangle$

ELSE

Otherwise, abort the transaction.

$\land$ $SendResp([start\_ts \mapsto start\_ts,\ type \mapsto \text{"commit\_aborted"}])$

$\land$ UNCHANGED $\langle req\_msgs,\ client\_vars,\ key\_vars,\ next\_ts \rangle$

In the spec, the primary key with a lock may clean up itself spontaneously. There is no need to model a client to request clean up because there is no difference between a optimistic client trying to read a key that has lock timeouted and the key trying to unlock itself.

$ServerCleanupStaleLock \triangleq$

$\exists\, k \in KEY :$

$\exists\, l \in key\_lock[k] :$

$\land$ $SendReqs(\{[type \mapsto \text{"check\_txn\_status"},$
$start\_ts \mapsto l.ts,$
$primary \mapsto l.primary,$
$resolving\_pessimistic\_lock \mapsto l.type = \text{"lock\_key"}$
$]\})$

$\land$ UNCHANGED $\langle resp\_msgs,\ client\_vars,\ key\_vars,\ next\_ts \rangle$

Clean up stale locks by checking the status of the primary key. It can be divided into two cases: *pk_lock* exists or not. Note that it is hard to model the *TTL* in TLA+ spec, so instead, the *TTL* is considered constantly expired when the action is taken.
The client does not care about the *resp* message, it cares about whether the lock is resolved, and resolving the lock is actually resolve the status of the corresponding transaction. Since, in the TLA+ spec, we ignored the *check_txn_status response message*.

$ServerCheckTxnStatus \triangleq$

$\exists\, req \in req\_msgs :$

$\land$ $req.type = \text{"check\_txn\_status"}$

$\land$ LET

$pk \triangleq req.primary$

$start\_ts \triangleq req.start\_ts$

$committed \triangleq \{w \in key\_write[pk] : w.start\_ts = start\_ts \land w.type = \text{"write"}\}$

$\quad$ IN

$\quad\quad$ IF $\exists\, lock \in key\_lock[pk] : lock.ts = start\_ts$

Found the matching lock whose TTL is expired.

THEN

IF

Pessimistic lock will be unlocked directly without rollback record.

$$
\begin{array}{l}
\exists\, lock \in key\_lock[pk] : \\
\quad \land\ lock.ts = start\_ts \\
\quad \land\ lock.type = \text{"lock\_key"} \\
\quad \land\ req.resolving\_pessimistic\_lock = \text{TRUE}
\end{array}
$$

THEN

$$
\begin{array}{l}
\land\ key\_lock' = [key\_lock \text{ EXCEPT } ![pk] = \{\}] \\
\land\ \text{UNCHANGED } \langle msg\_vars, key\_data, key\_write, client\_vars, next\_ts \rangle
\end{array}
$$

ELSE

$$
\begin{array}{l}
\land\ rollback(pk, start\_ts) \\
\land\ SendReqs(\{[type \mapsto \text{"resolve\_rollbacked"}, \\
\qquad\qquad\qquad start\_ts \mapsto start\_ts, \\
\qquad\qquad\qquad primary \mapsto pk]\}) \\
\land\ \text{UNCHANGED } \langle resp\_msgs, client\_vars, next\_ts \rangle
\end{array}
$$

Lock not found or start_ts on the lock mismatches.

ELSE

IF $committed \neq \{\}$ THEN

$$
\begin{array}{l}
\land\ SendReqs(\{[type \mapsto \text{"resolve\_committed"}, \\
\qquad\qquad\qquad start\_ts \mapsto start\_ts, \\
\qquad\qquad\qquad primary \mapsto pk, \\
\qquad\qquad\qquad commit\_ts \mapsto w.ts] : w \in committed\}) \\
\land\ \text{UNCHANGED } \langle resp\_msgs, client\_vars, key\_vars, next\_ts \rangle
\end{array}
$$

ELSE IF $req.resolving\_pessimistic\_lock = \text{TRUE}$ THEN

$\land\ \text{UNCHANGED } \langle vars \rangle$

ELSE

$$
\begin{array}{l}
\land\ rollback(pk, start\_ts) \\
\land\ SendReqs(\{[type \mapsto \text{"resolve\_rollbacked"}, \\
\qquad\qquad\qquad start\_ts \mapsto start\_ts, \\
\qquad\qquad\qquad primary \mapsto pk]\}) \\
\land\ \text{UNCHANGED } \langle resp\_msgs, client\_vars, next\_ts \rangle
\end{array}
$$

$$
\begin{array}{l}
ServerResolveCommitted \triangleq \\
\quad \exists\, req \in req\_msgs : \\
\qquad \land\ req.type = \text{"resolve\_committed"} \\
\qquad \land\ \text{LET} \\
\qquad\qquad start\_ts \triangleq req.start\_ts \\
\qquad\quad \text{IN} \\
\qquad\qquad \exists\, k \in KEY : \\
\qquad\qquad\quad \exists\, l \in key\_lock[k] : \\
\qquad\qquad\qquad \land\ l.primary = req.primary
\end{array}
$$

$\land\ l.ts = start\_ts$
$\land\ commit(k,\ start\_ts,\ req.commit\_ts)$
$\land$ UNCHANGED $\langle msg\_vars,\ client\_vars,\ key\_data,\ next\_ts\rangle$

$ServerResolveRollbacked\ \triangleq$
$\exists\, req \in req\_msgs :$
$\land\ req.type =$ "resolve_rollbacked"
$\land$ LET
$start\_ts\ \triangleq\ req.start\_ts$
IN
$\exists\, k \in KEY :$
$\exists\, l \in key\_lock[k] :$
$\land\ l.primary = req.primary$
$\land\ l.ts = start\_ts$
$\land\ rollback(k,\ start\_ts)$
$\land$ UNCHANGED $\langle msg\_vars,\ client\_vars,\ next\_ts\rangle$

Specification

$Init\ \triangleq$
$\land\ next\_ts = 1$
$\land\ req\_msgs = \{\}$
$\land\ resp\_msgs = \{\}$
$\land\ client\_state = [c \in CLIENT \mapsto$ "init"$]$
$\land\ client\_key = [c \in CLIENT \mapsto [locking \mapsto \{\},\ prewriting \mapsto \{\}]]$
$\land\ client\_ts = [c \in CLIENT \mapsto [start\_ts \mapsto NoneTs,$
$commit\_ts \mapsto NoneTs,$
$for\_update\_ts \mapsto NoneTs]]$
$\land\ key\_lock = [k \in KEY \mapsto \{\}]$
$\land\ key\_data = [k \in KEY \mapsto \{\}]$
$\land\ key\_write = [k \in KEY \mapsto \{\}]$

$Next\ \triangleq$
$\lor\ \exists\, c \in OPTIMISTIC\_CLIENT :$
$\lor\ ClientPrewriteOptimistic(c)$
$\lor\ ClientPrewrited(c)$
$\lor\ ClientCommit(c)$
$\lor\ \exists\, c \in PESSIMISTIC\_CLIENT :$
$\lor\ ClientLockKey(c)$
$\lor\ ClientLockedKey(c)$
$\lor\ ClientRetryLockKey(c)$
$\lor\ ClientPrewritePessimistic(c)$
$\lor\ ClientPrewrited(c)$
$\lor\ ClientCommit(c)$
$\lor\ ServerLockKey$
$\lor\ ServerPrewritePessimistic$

$\lor$ *ServerPrewriteOptimistic*
$\lor$ *ServerCommit*
$\lor$ *ServerCleanupStaleLock*
$\lor$ *ServerCheckTxnStatus*
$\lor$ *ServerResolveCommitted*
$\lor$ *ServerResolveRollbacked*

$Spec \triangleq Init \land \Box[Next]_{vars}$

Consistency Invariants

Check whether there is a "write" *record in key_write[k] corresponding to start_ts.*

$$
\begin{aligned}
&keyCommitted(k, start\_ts) \triangleq \\
&\quad \exists\, w \in key\_write[k] : \\
&\qquad \land\, w.start\_ts = start\_ts \\
&\qquad \land\, w.type = \text{“write”}
\end{aligned}
$$

A transaction can t be both committed and aborted.

$$
\begin{aligned}
&UniqueCommitOrAbort \triangleq \\
&\quad \forall\, resp, resp2 \in resp\_msgs : \\
&\qquad (resp.type = \text{“committed”}) \land (resp2.type = \text{“commit\_aborted”}) \Rightarrow \\
&\qquad\quad resp.start\_ts \neq resp2.start\_ts
\end{aligned}
$$

If a transaction is committed, the primary key must be committed and the secondary keys of the same transaction must be either committed or locked.

$$
\begin{aligned}
&CommitConsistency \triangleq \\
&\quad \forall\, resp \in resp\_msgs : \\
&\qquad (resp.type = \text{“committed”}) \Rightarrow \\
&\qquad\quad \exists\, c \in CLIENT : \\
&\qquad\qquad \land\, client\_ts[c].start\_ts = resp.start\_ts
\end{aligned}
$$

Primary key must be committed

$$\land\, keyCommitted(CLIENT\_PRIMARY[c], resp.start\_ts)$$

Secondary key must be either committed or locked by the *start_ts* of the transaction.

$$
\begin{aligned}
&\land\, \forall\, k \in CLIENT\_KEY[c] : \\
&\quad (\neg \exists\, l \in key\_lock[k] : l.ts = resp.start\_ts) = \\
&\qquad keyCommitted(k, resp.start\_ts)
\end{aligned}
$$

If a transaction is aborted, all key of that transaction must be not committed.

$$
\begin{aligned}
&AbortConsistency \triangleq \\
&\quad \forall\, resp \in resp\_msgs : \\
&\qquad (resp.type = \text{“commit\_aborted”}) \Rightarrow \\
&\qquad\quad \forall\, c \in CLIENT :
\end{aligned}
$$

$$(client\_ts[c].start\_ts = resp.start\_ts) \Rightarrow$$
$$\neg keyCommitted(CLIENT\_PRIMARY[c], resp.start\_ts)$$

For each write, the *commit_ts* should be strictly greater than the *start_ts* and have data written into *key_data*[k]. For each rollback, the *commit_ts* should equals to the *start_ts*.

$$\begin{aligned}
&WriteConsistency \triangleq \\
&\quad \forall k \in KEY : \\
&\quad\quad \forall w \in key_write[k] : \\
&\quad\quad\quad \lor \land w.type = \text{"write"} \\
&\quad\quad\quad\quad \land w.ts > w.start_ts \\
&\quad\quad\quad\quad \land w.start_ts \in key_data[k] \\
&\quad\quad\quad \lor \land w.type = \text{"rollback"} \\
&\quad\quad\quad\quad \land w.ts = w.start_ts
\end{aligned}$$

When the lock exists, there can't be a corresponding commit record, vice versa.

$$\begin{aligned}
&UniqueLockOrWrite \triangleq \\
&\quad \forall k \in KEY : \\
&\quad\quad \forall l \in key_lock[k] : \\
&\quad\quad\quad \forall w \in key_write[k] : \\
&\quad\quad\quad\quad w.start_ts \neq l.ts
\end{aligned}$$

For each key, ecah record in write column should have a unique *start_ts*.

$$\begin{aligned}
&UniqueWrite \triangleq \\
&\quad \forall k \in KEY : \\
&\quad\quad \forall w, w2 \in key_write[k] : \\
&\quad\quad\quad (w.start_ts = w2.start_ts) \Rightarrow (w = w2)
\end{aligned}$$

Snapshot Isolation

Asserts that *next_ts* is monotonically increasing.

$$NextTsMonotonicity \triangleq \Box[next\_ts' \geq next\_ts]_{vars}$$

Asserts that no *msg* would be deleted once sent.

$$\begin{aligned}
&MsgMonotonicity \triangleq \\
&\quad \land \Box[\forall req \in req_msgs : req \in req_msgs']_{vars} \\
&\quad \land \Box[\forall resp \in resp_msgs : resp \in resp_msgs']_{vars}
\end{aligned}$$

Asserts that all messages sent should have *ts* less than *next_ts*.

$$\begin{aligned}
&MsgTsConsistency \triangleq \\
&\quad \land \forall req \in req_msgs : \\
&\quad\quad\quad \land req.start_ts \leq next_ts \\
&\quad\quad\quad \land req.type \in \{\text{"commit"}, \text{"resolve_committed"}\} \Rightarrow \\
&\quad\quad\quad\quad req.commit_ts \leq next_ts \\
&\quad \land \forall resp \in resp_msgs : resp.start_ts \leq next_ts
\end{aligned}$$

SnapshotIsolation is implied from the following assumptions (but is not necessary) because *SnapshotIsolation* means that:

(1) Once a transaction is committed, all keys of the transaction should be always readable or have a lock on secondary *keys*(*eventually readable*).

PROOF BY *CommitConsistency*, *MsgMonotonicity*

(2) For a given transaction, all transaction that commits after that transaction should have greater *commit_ts* than the *next_ts* at the time that the given transaction commits, so as to be able to distinguish the transactions that have committed before and after from all transactions that preserved by (1).

PROOF BY *NextTsConsistency*, *MsgTsConsistency*

(3) All aborted transactions would be always not readable.

PROOF BY *AbortConsistency*, *MsgMonotonicity*

$$
\begin{aligned}
SnapshotIsolation \triangleq\ & \land CommitConsistency \\
& \land AbortConsistency \\
& \land NextTsMonotonicity \\
& \land MsgMonotonicity \\
& \land MsgTsConsistency
\end{aligned}
$$

THEOREM $Safety \triangleq$

$$
\begin{aligned}
Spec \Rightarrow \Box(\ & \land TypeOK \\
& \land UniqueCommitOrAbort \\
& \land CommitConsistency \\
& \land AbortConsistency \\
& \land WriteConsistency \\
& \land UniqueLockOrWrite \\
& \land UniqueWrite \\
& \land SnapshotIsolation)
\end{aligned}
$$
