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*

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

Functions that maps a client to keys it wants to read, write.
representing the involved keys of each client.

CONSTANTS *CLIENT_READ_KEY*, *CLIENT_WRITE_KEY*

$CLIENT\_KEY \triangleq [c \in CLIENT \mapsto CLIENT\_READ\_KEY[c] \cup CLIENT\_WRITE\_KEY[c]]$

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"

There's an *min_commit_ts* field to indicate the minimum commit time It's used in non-blocked reading.
TODO: upd *min_commit_ts* comment.

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*, *min_commit_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

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

In *TiKV*, there's an extra flag *rollback_if_not_exist in the check_txn_status request.*

Because the client prewrites the primary key and secondary key in parallel, it s possible that the primary key lock is missing and also no commit or rollback record for the transaction is found in the write *CF*, while there is a lock on the secondary key (so other transaction is blocked, therefore this *check_txn_status* is sent). And there are two possible cases:

1. The prewrite request for the primary key has not reached yet.
2. The client is crashed after sending the prewrite request for the secondary key.

In order to address the first case, the client sending *check_txn_status should not rollback the primary key until the TTL on the secondary key is expired, and thus, rollback_if_not_exist should be set to false before the TTL expires(and set true afterward).*

In TLA + spec, the TTL is considered constantly expired when the action is taken, so the rollback_if_not_exist is assumed true, thus no need to carry it in the message.

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

$RespMessages \triangleq$
$[start\_ts : Ts, type : \{\text{"prewrited"}\}, key : KEY]$
$\cup\ [start\_ts : Ts, type : \{\text{"get\_resp"}\}, key : KEY, value : Ts, met\_optimistic\_lock : \text{BOOLEAN}]$

Conceptually, acquire a pessimistic lock of a key is equivalent to reading its value, and putting the value in the response can reduce communication. Also, as mentioned above, we don't care about the actual value here, so a timestamp can be used instead of the value.

$\cup\ [start\_ts : Ts, type : \{\text{"locked\_key"}\}, key : KEY, value\_ts : Ts]$

$\cup$ [$start\_ts$: Ts, $type$: {"lock_failed"}, key : KEY, $latest\_commit\_ts$: Ts,
$lock\_ts$: Ts, $lock\_type$: {"no_lock", "lock_key", "prewrite_pessimistic", "prewrite_optimistic"}]
$\cup$ [$start\_ts$: Ts, $type$: {"committed",
"commit_aborted",
"prewrite_aborted",
"lock_key_aborted"}]
$\cup$ [$start\_ts$: Ts, $type$: {"check_txn_status_resp"},
$action$: {"rollbacked",
"pessimistic_rollbacked",
"committed",
"min_commit_ts_pushed",
"lock_not_exist_do_nothing"}]

$TypeOK \triangleq$ $\wedge$ $req\_msgs \in$ SUBSET $ReqMessages$
$\wedge$ $resp\_msgs \in$ SUBSET $RespMessages$
$\wedge$ $key\_data \in [KEY \rightarrow$ SUBSET $Ts]$
$\wedge$ $key\_lock \in [KEY \rightarrow$ SUBSET [ts : Ts,
$primary$: KEY,
As defined above, $Ts \triangleq Nat \setminus 0$, here we use 0
to indicates that there s no *min_commit_ts* limit.
min_commit_ts : $Ts \cup \{NoneTs\}$,
$type$: {"prewrite_optimistic",
"prewrite_pessimistic",
"lock_key"}]]
At most one lock in $key\_lock[k]$
$\wedge \forall k \in KEY : Cardinality(key\_lock[k]) \leq 1$
$\wedge$ $key\_write \in [KEY \rightarrow$ SUBSET (
[ts : Ts, $start\_ts$: Ts, $type$: {"write"}]
$\cup$ [ts : Ts, $start\_ts$: Ts, $type$: {"rollback"}, $protected$: BOOLEAN])]
The reading phase only apply for optimistic transactions
$\wedge$ $client\_state \in [CLIENT \rightarrow$ {"init", "locking", "reading", "prewriting", "committing"}]
$\wedge$ $client\_ts \in [CLIENT \rightarrow$ [$start\_ts$: $Ts \cup \{NoneTs\}$,
$commit\_ts$: $Ts \cup \{NoneTs\}$,
for_update_ts : $Ts \cup \{NoneTs\}$,
min_commit_ts : $Ts \cup \{NoneTs\}$]]
$\wedge$ $client\_key \in [CLIENT \rightarrow$ [$locking$: SUBSET KEY, $prewriting$: SUBSET KEY]]
$\wedge \forall c \in CLIENT : client\_key[c].locking \cap client\_key[c].prewriting = \{\}$
$\wedge$ $next\_ts \in Ts$

Client Actions

$ClientReadKey(c) \triangleq$
$\wedge$ $client\_state[c] =$ "init"
$\wedge$ $c \in OPTIMISTIC\_CLIENT$
$\wedge$ $client\_state' = [client\_state$ EXCEPT $![c] =$ "reading"]

$\land\ client\_ts' = [client\_ts \text{ EXCEPT } ![c].start\_ts = next\_ts]$
$\land\ next\_ts' = next\_ts + 1$
$\land\ SendReqs(\{[type \mapsto \text{"get"},$
$\quad start\_ts \mapsto client\_ts'[c].start\_ts,$
$\quad primary \mapsto CLIENT\_PRIMARY[c],$
$\quad key \mapsto k] : k \in CLIENT\_READ\_KEY[c]\})$
$\land\ \text{UNCHANGED } \langle resp\_msgs, client\_key, key\_vars\rangle$

$ClientLockKey(c) \triangleq$
$\land\ client\_state[c] = \text{"reading"}$
$\land\ client\_state' = [client\_state \text{ EXCEPT } ![c] = \text{"locking"}]$
$\land\ client\_ts' = [client\_ts \text{ EXCEPT } ![c].start\_ts = next\_ts, ![c].for\_update\_ts = next\_ts]$
$\land\ next\_ts' = next\_ts + 1$
Assume we need to acquire pessimistic locks for all keys
$\land\ client\_key' = [client\_key \text{ EXCEPT } ![c].locking = CLIENT\_KEY[c]]$
$\land\ SendReqs(\{[type \mapsto \text{"lock\_key"},$
$\quad start\_ts \mapsto client\_ts'[c].start\_ts,$
$\quad primary \mapsto CLIENT\_PRIMARY[c],$
$\quad key \mapsto k,$
$\quad for\_update\_ts \mapsto client\_ts'[c].for\_update\_ts] : k \in CLIENT\_KEY[c]\})$
$\land\ \text{UNCHANGED } \langle resp\_msgs, key\_vars\rangle$

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

$ClientRetryLockKey(c) \triangleq$
$\land\ client\_state[c] = \text{"locking"}$
$\land\ \exists\, resp \in resp\_msgs :$
$\quad \land\ resp.type = \text{"lock\_failed"}$
$\quad \land\ resp.start\_ts = client\_ts[c].start\_ts$
$\quad \land\ resp.latest\_commit\_ts > client\_ts[c].for\_update\_ts$
$\quad \land\ client\_ts' = [client\_ts \text{ EXCEPT } ![c].for\_update\_ts = resp.latest\_commit\_ts]$
$\quad \land\ \text{IF } resp.lock\_type = \text{"lock\_key"} \land \neg resp.lock\_ts = client\_ts[c].start\_ts$
$\quad\quad \text{THEN}$
$\quad\quad \land\ SendReqs(\{[type \mapsto \text{"check\_txn\_status"},$
$\quad\quad\quad start\_ts \mapsto client\_ts[c].start\_ts,$
$\quad\quad\quad caller\_start\_ts \mapsto next\_ts,$
$\quad\quad\quad primary \mapsto CLIENT\_PRIMARY[c],$
$\quad\quad\quad resoving\_pessimistic\_lock \mapsto \text{TRUE}]\})$
$\quad\quad \land\ next\_ts' = next\_ts + 1$

```
            ∧ UNCHANGED ⟨resp_msgs, key_vars, client_state, client_key⟩
            ELSE IF ¬resp.lock_type = "no_lock"
            THEN
              ∧ SendReqs({[type ↦ "check_txn_status",
                           start_ts ↦ client_ts[c].start_ts,
                           caller_start_ts ↦ next_ts,
                           primary ↦ CLIENT_PRIMARY[c],
                           resoving_pessimistic_lock ↦ FALSE]})
              ∧ next_ts′ = next_ts + 1
              ∧ UNCHANGED ⟨resp_msgs, key_vars, client_state, client_key⟩
            ELSE
              ∧ UNCHANGED ⟨resp_msgs, key_vars, client_state, client_key, next_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]})

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",
               start_ts ↦ client_ts[c].start_ts,
               primary ↦ CLIENT_PRIMARY[c],
               key ↦ k] : k ∈ CLIENT_KEY[c]})
  ∧ UNCHANGED ⟨resp_msgs, key_vars, client_ts, next_ts⟩
```

Add a function like *ClientRetryReadKey*(?)

```
ClientCheckTxnStatus(c) ≜
  ∧ client_state[c] = "reading"
  ∧ ∃ resp ∈ resp_msgs :
    ∧ resp.type = "get_resp"
    ∧ resp.met_optimistic_lock = TRUE
    ∧ SendReqs({[type ↦ "check_txn_status",
                 start_ts ↦ client_ts[c].start_ts,
                 caller_start_ts ↦ next_ts,
                 primary ↦ CLIENT_PRIMARY[c],
                 resovling_pessimistic_lock ↦ FALSE]})
    ∧ UNCHANGED ⟨resp_msgs, client_vars, key_vars⟩

ClientPrewriteOptimistic(c) ≜
  ∧ client_state[c] = "reading"
  ∧ client_state′ = [client_state EXCEPT ![c] = "prewriting"]
  ∧ client_key′ = [client_key EXCEPT ![c].prewriting = CLIENT_KEY[c]]
```

$\land\ SendReqs(\{[type \mapsto \text{“prewrite\_optimistic”},$
$\quad start\_ts \mapsto client\_ts[c].start\_ts,$
$\quad primary \mapsto CLIENT\_PRIMARY[c],$
$\quad key \mapsto k] : k \in CLIENT\_KEY[c]\})$
$\land\ \text{UNCHANGED}\ \langle resp\_msgs,\ client\_ts,\ key\_vars,\ next\_ts\rangle$

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

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

Server Actions

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

$unlock\_key(k) \triangleq$
$\land\ key\_lock' = [key\_lock\ \text{EXCEPT}\ ![k] = \{\}]$

$commit(pk,\ start\_ts,\ commit\_ts) \triangleq$
$\exists\, l \in key\_lock[pk] :$
$\quad \land\ l.ts = start\_ts$
$\quad \land\ unlock\_key(pk)$
$\quad \land\ key\_write' = [key\_write\ \text{EXCEPT}\ ![pk] = @ \cup \{[ts \mapsto commit\_ts,$
$\quad\quad type \mapsto \text{“write”},$
$\quad\quad start\_ts \mapsto start\_ts]\}]$

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.

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

IN

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

$\land$ IF $\exists\, l \in key\_lock[k] : l.ts = start\_ts$
  THEN $unlock\_key(k)$
  ELSE UNCHANGED key_lock
$\land\; key\_data' = [key\_data$ EXCEPT $![k] = @ \setminus \{start\_ts\}]$
$\land$ IF
  $\land\; \neg\exists\, w \in key\_write[k] : w.ts = start\_ts$
  THEN
    $key\_write' = [key\_write$ EXCEPT
    $![k] =$

collapse rollback

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

write rollback record

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

$ServerLockKey \triangleq$
$\exists\, req \in req\_msgs :$
$\land\; req.type = \text{"lock\_key"}$
$\land$ LET
  $k \triangleq req.key$
  $start\_ts \triangleq req.start\_ts$
  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 ∃ w ∈ key_write[k] : w.start_ts = start_ts ∧ w.type = "rollback"
           THEN
              If corresponding rollback record is found, which
              indicates that the transcation is rollbacked, abort the
              transaction.
             ∧ SendResp([start_ts ↦ start_ts, type ↦ "lock_key_aborted"])
             ∧ UNCHANGED ⟨req_msgs, client_vars, key_vars, next_ts⟩
           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.
             ∨ ∧ ¬∃ w ∈ latest_commit : w.ts > req.for_update_ts
               ∧ key_lock' = [key_lock EXCEPT ![k] = {[ts ↦ start_ts,
                                                      primary ↦ req.primary,
                                                      min_commit_ts ↦ NoneTs,
                                                      type ↦ "lock_key"]}]
               ∧ SendResp([start_ts ↦ start_ts, type ↦ "locked_key", key ↦ k])
               ∧ UNCHANGED ⟨req_msgs, client_vars, key_data, key_write, next_ts⟩
              Otherwise, reject the request and let client to retry
              with new for_update_ts.
             ∨ ∃ w ∈ latest_commit :
                ∧ w.ts > req.for_update_ts
                ∧ SendResp([start_ts ↦ start_ts,
                            type ↦ "lock_failed",
                            key ↦ k,
                            latest_commit_ts ↦ w.ts])
                ∧ UNCHANGED ⟨req_msgs, client_vars, key_vars, next_ts⟩

ServerReadKey ≜
  ∃ req ∈ req_msgs :
    ∧ req.type = "get"
    ∧ LET
        k ≜ req.key
        start_ts ≜ req.start_ts
      IN
       ∧ IF ¬∃ l ∈ key_lock : l.type = "prewrite_optimistic"
          THEN
            ∧ SendResp([start_ts ↦ start_ts, type ↦ "get_resp", key ↦ k, value ↦ Ts, met_optimistic_lock ↦
            ∧ UNCHANGED ⟨req_msgs, client_vars, key_vars⟩
          ELSE
            ∧ SendResp([start_ts ↦ start_ts, type ↦ "get_resp", key ↦ k, value ↦ NoneTs, met_optimistic_lo
            ∧ UNCHANGED ⟨req_msgs, client_vars, key_vars⟩
```

```
ServerPrewritePessimistic ≜
  ∃ req ∈ req_msgs :
    ∧ req.type = "prewrite_pessimistic"
    ∧ LET
        k ≜ req.key
        start_ts ≜ req.start_ts
      IN
        Pessimistic prewrite is allowed if pressimistic lock is
        acquired, or, there's no lock, and no write record whose
        commit_ts ≥ start_ts otherwise abort the transaction.
        ∧ IF ∃ l ∈ key_lock[k] : l.ts = start_ts
               ∨ ¬∃ w ∈ key_write[k] : w.ts ≥ start_ts
            THEN
              ∧ key_lock' = [key_lock EXCEPT ![k] = {[ts ↦ start_ts,
                                                      primary ↦ req.primary,
                                                      type ↦ "prewrite_pessimistic"]}]
              ∧ key_data' = [key_data EXCEPT ![k] = @ ∪ {start_ts}]
              ∧ SendResp([start_ts ↦ start_ts, type ↦ "prewrited", key ↦ k])
              ∧ UNCHANGED ⟨req_msgs, client_vars, key_write, next_ts⟩
            ELSE
              ∧ SendResp([start_ts ↦ start_ts, type ↦ "prewrite_aborted"])
              ∧ UNCHANGED ⟨req_msgs, client_vars, key_vars, next_ts⟩

ServerPrewriteOptimistic ≜
  ∃ req ∈ req_msgs :
    ∧ req.type = "prewrite_optimistic"
    ∧ LET
        k ≜ req.key
        start_ts ≜ req.start_ts
      IN
        ∧ IF ∃ w ∈ key_write[k] : w.ts ≥ start_ts
            THEN
              ∧ SendResp([start_ts ↦ start_ts, type ↦ "prewrite_aborted"])
              ∧ UNCHANGED ⟨req_msgs, client_vars, key_vars, next_ts⟩
            ELSE
              Optimistic prewrite is allowed only if no stale lock exists. If
              there is one, wait until ServerCleanupStaleLock to clean it up.
              ∧ ∨ key_lock[k] = {}
                ∨ ∃ l ∈ key_lock[k] : l.ts = start_ts
              ∧ key_lock' = [key_lock EXCEPT ![k] = {[ts ↦ start_ts,
                                                      primary ↦ req.primary,
                                                      min_commit_ts ↦ NoneTs,
                                                      type ↦ "prewrite_optimistic"]}]
              ∧ key_data' = [key_data EXCEPT ![k] = @ ∪ {start_ts}]
              ∧ SendResp([start_ts ↦ start_ts, type ↦ "prewrited", key ↦ k])
```

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

$ServerCommit \triangleq$
$\exists\, req \in req\_msgs :$
$\wedge\ req.type = \text{“commit”}$
$\wedge$ LET
$pk \triangleq req.primary$
$start\_ts \triangleq req.start\_ts$
IN
IF $\exists\, w \in key\_write[pk] : w.start\_ts = start\_ts \wedge w.type = \text{“write”}$
THEN
Key has already been committed. Do nothing.
$\wedge\ SendResp([start\_ts \mapsto start\_ts, type \mapsto \text{“committed”}])$
$\wedge$ 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.
$\wedge\ commit(pk, start\_ts, req.commit\_ts)$
$\wedge\ SendResp([start\_ts \mapsto start\_ts, type \mapsto \text{“committed”}])$
$\wedge$ UNCHANGED $\langle req\_msgs, client\_vars, key\_data, next\_ts \rangle$
ELSE
Otherwise, abort the transaction.
$\wedge\ SendResp([start\_ts \mapsto start\_ts, type \mapsto \text{“commit\_aborted”}])$
$\wedge$ 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.

$ServerCleanupStaleLock \triangleq$
$\exists\, k \in KEY :$
$\exists\, l \in key\_lock[k] :$
$\wedge\ SendReqs(\{[type \mapsto \text{“check\_txn\_status”},$
$start\_ts \mapsto l.ts,$
$caller\_start\_ts \mapsto next\_ts,$
$primary \mapsto l.primary,$
$resolving\_pessimistic\_lock \mapsto l.type = \text{“lock\_key”}]\})$
$\wedge\ next\_ts' = next\_ts + 1$
$\wedge$ UNCHANGED $\langle resp\_msgs, client\_vars, key\_vars \rangle$

Clean up the stale transaction by checking the status of the primary key.

In practice, the transaction will be rolled back if TTL on the lock is expired. But because it is hard to model the TTL in TLA+ spec, the TTL is considered constantly expired when the action is taken.

Moreover, *TiKV* will send a response for *TxnStatus to the client, and then depending on the TxnStatus, the client will send resolve_rollback or resolve_commit to the secondary keys to clean up stale locks.In the TLA + spec, the response to check_txn_status is omitted and TiKV will directly send resolve_rollback or resolve_commit message to secondary keys, because the action of client sending resolve message by proxying the TxnStatus from TiKV does not change the state of the client, therefore is equal to directly sending resolve message by TiKV*

$ServerCheckTxnStatus \triangleq$

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

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

$\quad \land\, \text{LET}$

$\qquad pk \triangleq req.primary$

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

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

$\qquad caller\_start\_ts \triangleq req.caller\_start\_ts$

$\quad\; \text{IN}$

$\qquad \text{IF } \exists\, lock \in key\_lock[pk] : lock.ts = start\_ts$

Found the matching lock.

$\qquad \text{THEN}$

$\qquad \lor$

$\qquad\; \text{IF}$

Pessimistic lock will be unlocked directly without rollback record.

$\qquad\quad \exists\, lock \in key\_lock[pk] :$

$\qquad\qquad \land\, lock.ts = start\_ts$

$\qquad\qquad \land\, lock.type = \text{"lock\_key"}$

$\qquad\qquad \land\, req.resolving\_pessimistic\_lock = \text{TRUE}$

$\qquad\quad \text{THEN}$

$\qquad\quad \land\, unlock\_key(pk)$

$\qquad\quad \land\, SendResp(\{[type \mapsto \text{"check\_txn\_status\_resp"}, start\_ts \mapsto start\_ts, action \mapsto \text{"pessimistic\_rollback"}]\})$

$\qquad\quad \land\, \text{UNCHANGED } \langle msg\_vars, key\_data, key\_write, client\_vars, next\_ts \rangle$

$\qquad\quad \text{ELSE}$

$\qquad\quad \land\, rollback(pk, start\_ts)$

$\qquad\quad \land\, SendReqs(\{[type \mapsto \text{"resolve\_rollbacked"}, start\_ts \mapsto start\_ts, primary \mapsto pk]\})$

$\qquad\quad \land\, SendResp([type \mapsto \text{"check\_txn\_status\_resp"}, start\_ts \mapsto start\_ts, action \mapsto \text{"rollbacked"}])$

$\qquad\quad \land\, \text{UNCHANGED } \langle client\_vars, next\_ts \rangle$

$\qquad \lor$

Push min_commit_ts

$\qquad\; \exists\, lock \in key\_lock[pk] :$

```
        ∧ key_lock′ = [key_lock EXCEPT ![pk] = {[ts ↦ lock.ts,
                                                 type ↦ lock.type,
                                                 primary ↦ lock.primary,
                                                 min_commit_ts ↦ caller_start_ts]}]
        ∧ SendResp([type ↦ "check_txn_status_resp",
                    start_ts ↦ start_ts,
                    action ↦ "min_commit_ts_pushed"])
        ∧ UNCHANGED ⟨req_msgs, key_data, key_write, client_vars, next_ts⟩
      Lock not found or start_ts on the lock mismatches.
      ELSE
       IF committed ≠ {} THEN
        ∧ SendReqs({[type ↦ "resolve_committed",
                     start_ts ↦ start_ts,
                     primary ↦ pk,
                     commit_ts ↦ w.ts] : w ∈ committed})
        ∧ SendResp([type ↦ "check_txn_status_resp",
                    start_ts ↦ start_ts,
                    action ↦ "committed"])
        ∧ UNCHANGED ⟨client_vars, key_vars, next_ts⟩
       ELSE IF req.resolving_pessimistic_lock = TRUE THEN
        ∧    SendResp({[type ↦ "check_txn_status_resp",
                        start_ts ↦ start_ts,
                        action ↦ "lock_not_exist_do_nothing"]})
        ∧    UNCHANGED ⟨req_msgs, client_vars, key_vars, next_ts⟩
       ELSE
        ∧ rollback(pk, start_ts)
        ∧ SendReqs({[type ↦ "resolve_rollbacked",
                     start_ts ↦ start_ts,
                     primary ↦ pk]})
        ∧ SendResp([type ↦ "check_txn_status_resp",
                    start_ts ↦ start_ts,
                    action ↦ "rollbacked"])
        ∧ UNCHANGED ⟨client_vars, next_ts⟩

ServerResolveCommitted ≜
  ∃ req ∈ req_msgs :
    ∧ req.type = "resolve_committed"
    ∧ LET
        start_ts ≜ req.start_ts
      IN
       ∃ k ∈ KEY :
         ∃ l ∈ key_lock[k] :
           ∧ l.primary = req.primary
           ∧ l.ts = start_ts
           ∧ commit(k, start_ts, req.commit_ts)
```

$\wedge$ UNCHANGED $\langle msg\_vars,\ client\_vars,\ key\_data,\ next\_ts \rangle$

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

Specification

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

$Next \triangleq$
$\vee\ \exists\, c \in OPTIMISTIC\_CLIENT :$
$\vee\ ClientReadKey(c)$
$\vee\ ClientCheckTxnStatus(c)$
$\vee\ ClientPrewriteOptimistic(c)$
$\vee\ ClientPrewrited(c)$
$\vee\ ClientCommit(c)$
$\vee\ \exists\, c \in PESSIMISTIC\_CLIENT :$
$\vee\ ClientReadKey(c)$
$\vee\ ClientCheckTxnStatus(c)$
$\vee\ ClientLockKey(c)$
$\vee\ ClientLockedKey(c)$
$\vee\ ClientRetryLockKey(c)$
$\vee\ ClientPrewritePessimistic(c)$
$\vee\ ClientPrewrited(c)$

$\quad\quad \lor\ ClientCommit(c)$
$\quad \lor\ ServerLockKey$
$\quad \lor\ ServerPrewritePessimistic$
$\quad \lor\ ServerPrewriteOptimistic$
$\quad \lor\ ServerCommit$
$\quad \lor\ ServerCleanupStaleLock$
$\quad \lor\ ServerCheckTxnStatus$
$\quad \lor\ ServerResolveCommitted$
$\quad \lor\ ServerResolveRollbacked$

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

Consistency Invariants

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

$keyCommitted(k, start\_ts) \triangleq$
$\quad \exists\, w \in key\_write[k] :$
$\quad\quad \land\ w.start\_ts = start\_ts$
$\quad\quad \land\ w.type = \text{"write"}$

A transaction can t be both committed and aborted.

$UniqueCommitOrAbort \triangleq$
$\quad \forall\, resp, resp2 \in resp\_msgs :$
$\quad\quad (resp.type = \text{"committed"}) \land (resp2.type = \text{"commit\_aborted"}) \Rightarrow$
$\quad\quad\quad resp.start\_ts \neq resp2.start\_ts$

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.

$CommitConsistency \triangleq$
$\quad \forall\, resp \in resp\_msgs :$
$\quad\quad (resp.type = \text{"committed"}) \Rightarrow$
$\quad\quad\quad \exists\, c \in CLIENT :$
$\quad\quad\quad\quad \land\ client\_ts[c].start\_ts = resp.start\_ts$

Primary key must be committed

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

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

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

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

$AbortConsistency \triangleq$

$\forall resp \in resp\_msgs :$
$\quad (resp.type = \text{“commit\_aborted”}) \Rightarrow$
$\quad\quad \forall c \in CLIENT :$
$\quad\quad\quad (client\_ts[c].start\_ts = resp.start\_ts) \Rightarrow$
$\quad\quad\quad\quad \neg keyCommitted(CLIENT\_PRIMARY[c], resp.start\_ts)$

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*.

$$
\begin{aligned}
&MsgTsConsistency \triangleq \\
&\quad \land \forall req \in req\_msgs : \\
&\quad\quad\quad \land req.start\_ts \leq next\_ts \\
&\quad\quad\quad \land req.type \in \{\text{“commit”}, \text{“resolve\_committed”}\} \Rightarrow
\end{aligned}
$$

$req.commit\_ts \leq next\_ts$

$\land \forall resp \in resp\_msgs : resp.start\_ts \leq next\_ts$

$ReadSnapshotIsolation \triangleq$

$\land \forall resp \in resp\_msgs :$

$\quad \land resp.type =$ "get_resp"

$\quad \land$ LET

$\qquad start\_ts \triangleq resp.start\_ts$

$\qquad key \triangleq resp.key$

As mentioned before, the value is just a timestamp

$\qquad value \triangleq resp.value$

$\qquad met\_optimistic\_lock \triangleq resp.met\_optimistic\_lock$

IN

$\land \exists c \in CLIENT :$

$\quad \land client\_ts[c].start\_ts = start\_ts$

$\quad \land$ LET

$\qquad commit\_ts \triangleq client\_ts[c].commit\_ts$

IN

IF $commit\_ts \in Ts$ THEN

$\quad \land \neg \exists w \in key\_write[key] :$

$\qquad start\_ts \leq w.ts \land w.ts \leq commit\_ts$

ELSE

$\quad \land$ TRUE

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*

TODO: Explain the *ReadSnapshotIsolation*

$SnapshotIsolation \triangleq \land CommitConsistency$
$\qquad \land AbortConsistency$
$\qquad \land NextTsMonotonicity$
$\qquad \land MsgMonotonicity$
$\qquad \land MsgTsConsistency$
$\qquad \land ReadSnapshotIsolation$

THEOREM $Safety \triangleq$

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