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 $\to$ 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) $LockType \in \{\text{"}PUT\text{"}, \text{"}DELETE\text{"}, \text{"}LOCK\text{"}\} \land for\_update\_ts = 0 \equiv type = \text{"prewrite\_optimistic"}$
(2) $LockType \in \{\text{"}PUT\text{"}, \text{"}DELETE\text{"}\} \land for\_update\_ts > 0 \equiv type = \text{"prewrite\_pessimistic"}$
(3) $LockType = \text{"PESSIMISTIC"} \equiv type = \text{"lock\_key"}$

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

$ReqMessages \triangleq$
$[start\_ts : Ts,\ primary : KEY,\ type : \{\text{“lock\_key”}\},\ key : KEY,\ for\_update\_ts : Ts]$
$\cup\ [start\_ts : Ts,\ primary : KEY,\ type : \{\text{“prewrite\_optimistic”}\},\ key : KEY]$
$\cup\ [start\_ts : Ts,\ primary : KEY,\ type : \{\text{“prewrite\_pessimistic”}\},\ key : KEY]$
$\cup\ [start\_ts : Ts,\ primary : KEY,\ type : \{\text{“commit”}\},\ commit\_ts : Ts]$
$\cup\ [start\_ts : Ts,\ primary : KEY,\ type : \{\text{“resolve\_rollbacked”}\}]$
$\cup\ [start\_ts : Ts,\ primary : KEY,\ type : \{\text{“resolve\_committed”}\},\ commit\_ts : Ts]$
$\cup\ [start\_ts : Ts,\ primary : KEY,\ type : \{\text{“check\_txn\_status\_req”}\},\ rollback\_if\_not\_exist : \text{BOOLEAN}\ ,\ resolving\_pessimistic\_lock : \text{BOOLEAN}\ ]$

$RespMessages \triangleq$
$[start\_ts : Ts,\ type : \{\text{“prewrited”},\ \text{“locked\_key”}\},\ key : KEY]$
$\cup\ [start\_ts : Ts,\ type : \{\text{“lock\_failed”}\},\ key : KEY,\ latest\_commit\_ts : Ts]$
$\cup\ [start\_ts : Ts,\ type : \{\text{“committed”},\ \text{“commit\_aborted”},\ \text{“prewrite\_aborted”},\ \text{“lock\_key\_aborted”}\}]$
$\cup\ [start\_ts : Ts,\ type : \{\text{“check\_txn\_status\_resp”}\},\ status : \{\text{“Rollbacked”},\ \text{“PessimisticRollbacked”},\ \text{“Committed”},\ \text{“Uncommitted”},\ \text{“MinCommitTsPushed”},\ \text{“ErrTxnNotFound”},\ \text{“LockNotExistDoNothing”}\}]$

$TypeOK \triangleq$
$\land\ req\_msgs \in \text{SUBSET}\ ReqMessages$
$\land\ resp\_msgs \in \text{SUBSET}\ RespMessages$
$\land\ key\_data \in [KEY \to \text{SUBSET}\ Ts]$
$\land\ key\_lock\ \in [KEY \to \text{SUBSET}\ [ts : Ts,\ primary : KEY,\ type : \{\text{“prewrite\_optimistic”},\ \text{“prewrite\_pessimistic”},\ \text{“lock\_key”}\}]]$
At most one lock in $key\_lock[k]$
$\land\ \forall\, k \in KEY : Cardinality(key\_lock[k]) \leq 1$
$\land\ key\_write \in [KEY \to \text{SUBSET}\ ($
$[ts : Ts,\ start\_ts : Ts,\ type : \{\text{“write”}\}]$
$\cup\ [ts : Ts,\ start\_ts : Ts,\ type : \{\text{“rollback”}\},\ protected : \text{BOOLEAN}\ ])]$
$\land\ client\_state \in [CLIENT \to \{\text{“init”},\ \text{“locking”},\ \text{“prewriting”},\ \text{“committing”}\}]$
$\land\ client\_ts \in [CLIENT \to [start\_ts : Ts \cup \{NoneTs\},\ commit\_ts : Ts \cup \{NoneTs\},$

$for\_update\_ts : Ts \cup \{NoneTs\}]]$
$\land\ client\_key \in [CLIENT \to [locking : \text{SUBSET}\ KEY,\ prewriting : \text{SUBSET}\ KEY]]$
$\land\ next\_ts \in Ts$

---

Client Actions

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

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

$ClientRetryLockKey(c) \triangleq$
$\land\ client\_state[c] = \text{“locking”}$
$\land\ \exists\, resp \in resp\_msgs :$
$\land\ resp.type = \text{“lock\_failed”}$
$\land\ resp.start\_ts = client\_ts[c].start\_ts$
$\land\ resp.latest\_commit\_ts > client\_ts[c].for\_update\_ts$
$\land\ client\_ts' = [client\_ts\ \text{EXCEPT}\ ![c].for\_update\_ts = resp.latest\_commit\_ts]$
$\land\ SendReqs(\{[type \mapsto \text{“lock\_key”},$
$start\_ts \mapsto client\_ts'[c].start\_ts,$
$primary \mapsto CLIENT\_PRIMARY[c],$
$key \mapsto resp.key,$
$for\_update\_ts \mapsto client\_ts'[c].for\_update\_ts]\})$
$\land\ \text{UNCHANGED}\ \langle resp\_msgs,\ key\_vars,\ client\_state,\ client\_key,\ next\_ts\rangle$

$ClientPrewritePessimistic(c) \triangleq$
$\land\ client\_state[c] = \text{“locking”}$
$\land\ client\_key[c].locking = \{\}$

$\land\ client\_state' = [client\_state \text{ EXCEPT } ![c] = \text{“prewriting”}]$
$\land\ client\_key' = [client\_key \text{ EXCEPT } ![c].prewriting = CLIENT\_KEY[c]]$
$\land\ SendReqs(\{[type \mapsto \text{“prewrite\_pessimistic”},$
$start\_ts \mapsto client\_ts[c].start\_ts,$
$primary \mapsto CLIENT\_PRIMARY[c],$
$key \mapsto k] : k \in CLIENT\_KEY[c]\})$
$\land$ UNCHANGED $\langle resp\_msgs,\ key\_vars,\ client\_ts,\ next\_ts\rangle$

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

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

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

---

Server Actions

Write the write column and unlock the lock iff the lock exists.
$commit(pk,\ start\_ts,\ commit\_ts) \triangleq$

```
  ∃ l ∈ key_lock[pk] :
    ∧ l.ts = start_ts
    ∧ key_lock' = [key_lock EXCEPT ![pk] = {}]
    ∧ key_write' = [key_write EXCEPT ![pk] = @ ∪ {[ts ↦ commit_ts,
                                                   type ↦ "write",
                                                   start_ts ↦ start_ts]}]

 Rollback the transaction that starts at start_ts on key k.
rollback(k, start_ts) ≜
  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 ≜ ∨ ∃ l ∈ key_lock[k] :
                    ∧ l.ts = start_ts
                    ∧ l.primary = k
                    ∧ l.type ∈ {"lock_key", "prewrite_pessimistic"}
                ∨ ∃ l ∈ key_lock[k] : l.ts ≠ start_ts
                ∨ key_lock[k] = {}
  IN
     If a lock exists and has the same ts, unlock it.
    ∧ IF ∃ l ∈ key_lock[k] : l.ts = start_ts
       THEN key_lock' = [key_lock EXCEPT ![k] = {}]
       ELSE UNCHANGED key_lock
    ∧ key_data' = [key_data EXCEPT ![k] = @ \ {start_ts}]
    ∧ IF
          ∧ ¬∃ w ∈ key_write[k] : w.ts = start_ts
      THEN
          key_write' = [key_write EXCEPT
            ![k] =
              collapse rollback
             (@ \ {w ∈ @ : w.type = "rollback" ∧ ¬w.protected ∧ w.ts < start_ts})
              write rollback record
             ∪ {[ts ↦ start_ts,
                 start_ts ↦ start_ts,
                 type ↦ "rollback",
                 protected ↦ protected]}]
      ELSE
        UNCHANGED ⟨key_write⟩

ServerLockKey ≜
  ∃ req ∈ req_msgs :
    ∧ req.type = "lock_key"
    ∧ LET
```

```
    k ≜ req.key
    start_ts ≜ 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.
    ∧ key_lock[k] = {}
    ∧ LET
        latest_write ≜ {w ∈ key_write[k] : ∀ w2 ∈ key_write[k] : w.ts ≥ w2.ts}
        all_commits ≜ {w ∈ key_write[k] : w.type = "write"}
        latest_commit ≜ {w ∈ all_commits : ∀ w2 ∈ all_commits : w.ts ≥ 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,
                                                   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⟩

ServerPrewritePessimistic ≜
  ∃ req ∈ req_msgs :
    ∧ req.type = "prewrite_pessimistic"
    ∧ LET
```

```
        k ≜ req.key
        start_ts ≜ req.start_ts
      IN
        Pessimistic prewrite is allowed only if pessimistic lock is
        acquired, otherwise abort the transaction.
        ∧ IF ∃ l ∈ key_lock[k] : l.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,
                                                   type ↦ "prewrite_optimistic"]}]
            ∧ 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⟩

ServerCommit ≜
  ∃ req ∈ req_msgs :
    ∧ req.type = "commit"
    ∧ LET
        pk ≜ req.primary
```

```
    start_ts ≜ req.start_ts
  IN
    IF ∃ w ∈ key_write[pk] : w.start_ts = start_ts ∧ w.type = "write"
    THEN
      Key has already been committed. Do nothing.
      ∧ SendResp([start_ts ↦ start_ts, type ↦ "committed"])
      ∧ UNCHANGED ⟨req_msgs, client_vars, key_vars, next_ts⟩
    ELSE
      IF ∃ l ∈ key_lock[pk] : l.ts = start_ts
      THEN
        Commit the key only if the prewrite lock exists.
        ∧ commit(pk, start_ts, req.commit_ts)
        ∧ SendResp([start_ts ↦ start_ts, type ↦ "committed"])
        ∧ UNCHANGED ⟨req_msgs, client_vars, key_data, next_ts⟩
      ELSE
        Otherwise, abort the transaction.
        ∧ SendResp([start_ts ↦ start_ts, type ↦ "commit_aborted"])
        ∧ UNCHANGED ⟨req_msgs, client_vars, key_vars, next_ts⟩

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 ≜
  ∃ k ∈ KEY :
    ∃ l ∈ key_lock[k] :
      the resolving_pessimistic_lock field is set to * TRUE *
      CASE l.type ∈ {"lock_key"} →
        ∧  SendReqs({[type ↦ "check_txn_status_req",
                      start_ts ↦ l.ts,
                      primary ↦ l.primary,
                      rollback_if_not_exist ↦ TRUE,
                      resolving_pessimistic_lock ↦ TRUE
                      ]})
        ∧  UNCHANGED ⟨resp_msgs, client_vars, key_vars, next_ts⟩
        ∨  ∧ SendReqs({[type ↦ "check_txn_status_req",
                       start_ts ↦ l.ts,
                       primary ↦ l.primary,
                       rollback_if_not_exist ↦ FALSE,
                       resolving_pessimistic_lock ↦ TRUE
                       ]})
           ∧ UNCHANGED ⟨resp_msgs, client_vars, key_vars, next_ts⟩

      the resolving_pessimistic_lock field is set to * FALSE *
      □l.type ∈ {"prewrite_optimistic", "prewrite_pessimistic"} →
```

```
        ∧ SendReqs({[type ↦ "check_txn_status_req",
                     start_ts ↦ l.ts,
                     primary ↦ l.primary,
                     rollback_if_not_exist ↦ TRUE,
                     resolving_pessimistic_lock ↦ FALSE
                     ]})
        ∧ UNCHANGED ⟨resp_msgs, client_vars, key_vars, next_ts⟩
        ∨ ∧ SendReqs({[type ↦ "check_txn_status_req",
                     start_ts ↦ l.ts,
                     primary ↦ l.primary,
                     rollback_if_not_exist ↦ FALSE,
                     resolving_pessimistic_lock ↦ FALSE
                     ]})
          ∧ UNCHANGED ⟨resp_msgs, client_vars, key_vars, next_ts⟩
```

*Clean up stale locks by checking the status of the primary key.Commit the secondary keys if primary key is committed; otherwise rollback the transaction by rolling − back the primary key, and then also rollback the secondarys.*

```
ServerCheckTxnStatus ≜
  ∃ req ∈ req_msgs :
    ∧ req.type = "check_txn_status_req"
    ∧ LET
        pk ≜ req.primary
        start_ts ≜ req.start_ts
        pk_lock ≜ key_lock[pk]
        committed ≜ {w ∈ key_write[pk] : w.start_ts = start_ts ∧ w.type = "write"}
        rollbacked ≜ {r ∈ key_write[pk] : r.start_ts = start_ts ∧ r.type = "rollback"}
      IN
        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,
                       status ↦ "Committed"])
          ∧ UNCHANGED ⟨client_vars, key_vars, next_ts⟩
        ELSE
        IF rollbacked ≠ {}
        THEN
          ∧ rollback(pk, start_ts)
          ∧ SendReqs({[type ↦ "resolve_rollbacked",
                       start_ts ↦ start_ts,
```

$primary \mapsto pk]\})$
$\land\ SendResp([type \mapsto$ "check_txn_status_resp",
$start\_ts \mapsto start\_ts,$
$status \mapsto$ "Rollbacked"$])$
$\land$ UNCHANGED $\langle client\_vars, next\_ts\rangle$

*No commit or rollback record*
ELSE
IF $\exists\, lock \in pk\_lock : lock.ts = start\_ts$
*Has a matching(lock_ts or start_ts)lock*
THEN
$\lor$
*TTL expire*
IF $\exists\, lock \in pk\_lock :$
$lock.type =$ "lock_key"
$\land\ req.resolving\_pessimistic\_lock =$ TRUE
THEN
$\land\ key\_lock' = [key\_lock$ EXCEPT $![pk] = \{\}]$
$\land\ SendResp([type \mapsto$ "check_txn_status_resp",
$start\_ts \mapsto start\_ts,$
$status \mapsto$ "PessimisticRollbacked"$])$
$\land$ UNCHANGED $\langle req\_msgs, key\_data, key\_write, client\_vars, next\_ts\rangle$
ELSE
$\land\ rollback(pk, start\_ts)$
$\land\ SendReqs(\{[type \mapsto$ "resolve_rollbacked",
$start\_ts \mapsto start\_ts,$
$primary \mapsto pk]\})$
$\land\ SendResp([type \mapsto$ "check_txn_status_resp",
$start\_ts \mapsto start\_ts,$
$status \mapsto$ "PessimisticRollbacked"$])$
$\land$ UNCHANGED $\langle client\_vars, next\_ts\rangle$
$\lor$
*uncommitted*
$\land\ SendResp([type \mapsto$ "check_txn_status_resp",
$start\_ts \mapsto start\_ts,$
$status \mapsto$ "Uncommitted"$])$
$\land$ UNCHANGED $\langle req\_msgs, key\_vars, client\_vars, next\_ts\rangle$
ELSE
*LockNotExist*
IF $\neg req.rollback\_if\_not\_exist$
THEN
$\land\ SendResp([type \mapsto$ "check_txn_status_resp",
$start\_ts \mapsto start\_ts,$
$status \mapsto$ "ErrTxnNotFound"$])$
$\land$ UNCHANGED $\langle req\_msgs, key\_vars, client\_vars, next\_ts\rangle$

```
        ELSE
      IF req.resolving_pessimistic_lock
        THEN
          ∧ SendResp([type ↦ "check_txn_status_resp",
                      start_ts ↦ start_ts,
                      status ↦ "LockNotExistDoNothing"])
          ∧ UNCHANGED ⟨req_msgs, key_vars, client_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,
                      status ↦ "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)
            ∧ UNCHANGED ⟨msg_vars, client_vars, key_data, next_ts⟩

ServerResolveRollbacked ≜
  ∃ req ∈ req_msgs :
    ∧ req.type = "resolve_rollbacked"
    ∧ LET
        start_ts ≜ req.start_ts
      IN
        ∃ k ∈ KEY :
          ∃ l ∈ key_lock[k] :
            ∧ l.primary = req.primary
            ∧ l.ts = start_ts
            ∧ rollback(k, start_ts)
            ∧ UNCHANGED ⟨msg_vars, client_vars, next_ts⟩
```

*Specification*

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

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

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

---

*Consistency Invariants*

*Check whether there is a* “write” *record in* $key\_write[k]$ *corresponding to* $start\_ts$.
$keyCommitted(k, start\_ts) \triangleq$
$\exists\, w \in key\_write[k] :$
$\land\ w.start\_ts = start\_ts$
$\land\ w.type = \text{“write”}$

*A transaction can* t be both committed and aborted.

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

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.

$CommitConsistency \triangleq$
$\forall\, resp \in resp\_msgs :$
$(resp.type = \text{“committed”}) \Rightarrow$
$\exists\, c \in CLIENT :$
$\land\, client\_ts[c].start\_ts = resp.start\_ts$
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.
$\land \forall\, k \in CLIENT\_KEY[c] :$
$(\neg\exists\, l \in key\_lock[k] : l.ts = resp.start\_ts) =$
$keyCommitted(k,\, resp.start\_ts)$

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

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

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*.

$WriteConsistency \triangleq$
$\forall\, k \in KEY :$
$\forall\, w \in key\_write[k] :$
$\lor\ \land\, w.type = \text{“write”}$
$\land\, w.ts > w.start\_ts$
$\land\, w.start\_ts \in key\_data[k]$
$\lor\ \land\, w.type = \text{“rollback”}$
$\land\, w.ts = w.start\_ts$

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

$UniqueLockOrWrite \triangleq$
$\forall\, k \in KEY :$

$\forall\, l \in key\_lock[k] :$
$\quad \forall\, w \in key\_write[k] :$
$\quad\quad w.start\_ts \neq l.ts$

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

$UniqueWrite \triangleq$
$\forall\, k \in KEY :$
$\quad \forall\, w,\, w2 \in key\_write[k] :$
$\quad\quad (w.start\_ts = w2.start\_ts) \Rightarrow (w = w2)$

---

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.

$MsgMonotonicity \triangleq$
$\quad \wedge \Box[\forall\, req \in req\_msgs : req \in req\_msgs']_{vars}$
$\quad \wedge \Box[\forall\, resp \in resp\_msgs : resp \in resp\_msgs']_{vars}$

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

$MsgTsConsistency \triangleq$
$\quad \wedge \forall\, req \in req\_msgs :$
$\quad\quad \wedge req.start\_ts \leq next\_ts$
$\quad\quad \wedge req.type \in \{\text{“commit”},\ \text{“resolve\_committed”}\} \Rightarrow$
$\quad\quad\quad req.commit\_ts \leq next\_ts$
$\quad \wedge \forall\, resp \in resp\_msgs : resp.start\_ts \leq next\_ts$

*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*

$SnapshotIsolation \triangleq \wedge\, CommitConsistency$
$\quad \wedge\, AbortConsistency$
$\quad \wedge\, NextTsMonotonicity$
$\quad \wedge\, MsgMonotonicity$
$\quad \wedge\, MsgTsConsistency$

THEOREM $Safety \triangleq$

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