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 $\rightarrow$ 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) $LockType \in \{\text{“}PUT\text{”}, \text{“}DELETE\text{”}, \text{“}LOCK\text{”}\} \land for\_update\_ts = 0 \equiv$ $type =$ “prewrite_optimistic”
(2) $LockType \in \{\text{“}PUT\text{”}, \text{“}DELETE\text{”}\} \land for\_update\_ts > 0 \equiv$ $type =$ “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

$$
\begin{aligned}
ReqMessages \triangleq\ & \\
& [start\_ts : Ts,\ primary : KEY,\ type : \{\text{"lock\_key"}\},\ key : KEY, \\
& \quad for\_update\_ts : Ts] \\
\cup\ & [start\_ts : Ts,\ primary : KEY,\ type : \{\text{"prewrite\_optimistic"}\},\ key : KEY] \\
\cup\ & [start\_ts : Ts,\ primary : KEY,\ type : \{\text{"prewrite\_pessimistic"}\},\ key : KEY] \\
\cup\ & [start\_ts : Ts,\ primary : KEY,\ type : \{\text{"commit"}\},\ commit\_ts : Ts] \\
\cup\ & [start\_ts : Ts,\ primary : KEY,\ type : \{\text{"resolve\_rollbacked"}\}] \\
\cup\ & [start\_ts : Ts,\ primary : KEY,\ type : \{\text{"resolve\_committed"}\},\ commit\_ts : Ts] \\
\cup\ & [start\_ts : Ts,\ primary : KEY,\ type : \{\text{"check\_txn\_status"}\},\ resolving\_pessimistic\_lock : \text{BOOLEAN}]
\end{aligned}
$$

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

$$
\begin{aligned}
TypeOK \triangleq\ & \land req\_msgs \in \text{SUBSET } ReqMessages \\
& \land resp\_msgs \in \text{SUBSET } RespMessages \\
& \land key\_data \in [KEY \rightarrow \text{SUBSET } Ts] \\
& \land key\_lock \in [KEY \rightarrow \text{SUBSET } [ts : Ts, \\
& \qquad primary : KEY, \\
& \qquad type : \{\text{"prewrite\_optimistic"}, \\
& \qquad\qquad \text{"prewrite\_pessimistic"}, \\
& \qquad\qquad \text{"lock\_key"}\}]]
\end{aligned}
$$

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

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

Client Actions

$$
\begin{aligned}
& ClientLockKey(c) \triangleq \\
& \quad \land client\_state[c] = \text{"init"} \\
& \quad \land client\_state' = [client\_state \text{ EXCEPT } ![c] = \text{"locking"}]
\end{aligned}
$$

$\land\ client\_ts' = [client\_ts \text{ EXCEPT } ![c].start\_ts = next\_ts,\ ![c].for\_update\_ts = next\_ts]$
$\land\ next\_ts' = next\_ts + 1$

Assume we need to acquire pessimistic locks for all keys

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

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

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

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

$ClientPrewriteOptimistic(c) \triangleq$

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

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

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

Server Actions

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

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

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.

$$
\begin{aligned}
&\mathit{protected} \triangleq \lor \exists\, l \in \mathit{key\_lock}[k] : \\
&\qquad\qquad \land\, l.\mathit{ts} = \mathit{start\_ts} \\
&\qquad\qquad \land\, l.\mathit{primary} = k \\
&\qquad\qquad \land\, l.\mathit{type} \in \{\text{“lock\_key”}, \text{“prewrite\_pessimistic”}\} \\
&\qquad \lor \exists\, l \in \mathit{key\_lock}[k] : l.\mathit{ts} \neq \mathit{start\_ts} \\
&\qquad \lor\, \mathit{key\_lock}[k] = \{\}
\end{aligned}
$$

IN

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

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

collapse rollback

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

write rollback record

$$
\begin{aligned}
&\cup \{[\mathit{ts} \mapsto \mathit{start\_ts}, \\
&\quad \mathit{start\_ts} \mapsto \mathit{start\_ts}, \\
&\quad \mathit{type} \mapsto \text{“rollback”}, \\
&\quad \mathit{protected} \mapsto \mathit{protected}]\}] \\
&\text{ELSE} \\
&\quad \text{UNCHANGED } \langle \mathit{key\_write} \rangle
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{ServerLockKey} \triangleq \\
&\ \exists\, \mathit{req} \in \mathit{req\_msgs} : \\
&\quad \land\ \mathit{req}.\mathit{type} = \text{“lock\_key”} \\
&\quad \land\ \text{LET} \\
&\qquad k \triangleq \mathit{req}.\mathit{key} \\
&\qquad \mathit{start\_ts} \triangleq \mathit{req}.\mathit{start\_ts} \\
&\quad\ \ \text{IN}
\end{aligned}
$$

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

$$
\begin{aligned}
&\land\ \mathit{key\_lock}[k] = \{\} \\
&\land\ \text{LET} \\
&\qquad \mathit{latest\_write} \triangleq \{w \in \mathit{key\_write}[k] : \forall\, w2 \in \mathit{key\_write}[k] : w.\mathit{ts} \geq w2.\mathit{ts}\}
\end{aligned}
$$

$all\_commits \triangleq \{w \in key\_write[k] : w.type = \text{"write"}\}$
$latest\_commit \triangleq \{w \in all\_commits : \forall\, w2 \in all\_commits : w.ts \geq w2.ts\}$

$\textsc{in}$

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

$\textsc{then}$

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

$\land\ SendResp([start\_ts \mapsto start\_ts,\ type \mapsto \text{"lock\_key\_aborted"}])$
$\land\ \textsc{unchanged}\ \langle req\_msgs,\ client\_vars,\ key\_vars,\ next\_ts\rangle$

$\textsc{else}$

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.

$\lor\ \land \neg\exists\, w \in latest\_commit : w.ts > req.for\_update\_ts$
$\quad \land\ key\_lock' = [key\_lock\ \textsc{except}\ ![k] = \{[ts \mapsto start\_ts,$
$\qquad\qquad primary \mapsto req.primary,$
$\qquad\qquad type \mapsto \text{"lock\_key"}]\}]$
$\quad \land\ SendResp([start\_ts \mapsto start\_ts,\ type \mapsto \text{"locked\_key"},\ key \mapsto k])$
$\quad \land\ \textsc{unchanged}\ \langle req\_msgs,\ client\_vars,\ key\_data,\ key\_write,\ next\_ts\rangle$

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

$\lor\ \exists\, w \in latest\_commit :$
$\quad \land\ w.ts > req.for\_update\_ts$
$\quad \land\ SendResp([start\_ts \mapsto start\_ts,$
$\qquad\qquad type \mapsto \text{"lock\_failed"},$
$\qquad\qquad key \mapsto k,$
$\qquad\qquad latest\_commit\_ts \mapsto w.ts])$
$\quad \land\ \textsc{unchanged}\ \langle req\_msgs,\ client\_vars,\ key\_vars,\ next\_ts\rangle$

$ServerPrewritePessimistic \triangleq$

$\exists\, req \in req\_msgs :$
$\quad \land\ req.type = \text{"prewrite\_pessimistic"}$
$\quad \land\ \textsc{let}$
$\qquad k \triangleq req.key$
$\qquad start\_ts \triangleq req.start\_ts$
$\quad \textsc{in}$

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

$\quad \land\ \textsc{if}\ \exists\, l \in key\_lock[k] : l.ts = start\_ts$
$\qquad \textsc{then}$
$\qquad \land\ key\_lock' = [key\_lock\ \textsc{except}\ ![k] = \{[ts \mapsto start\_ts,$
$\qquad\qquad primary \mapsto req.primary,$

$type \mapsto \text{"prewrite\_pessimistic"}]\}]$

$\land\ key\_data' = [key\_data\ \textsc{except}\ ![k] = @ \cup \{start\_ts\}]$

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

$\land\ \textsc{unchanged}\ \langle req\_msgs,\ client\_vars,\ key\_write,\ next\_ts \rangle$

$\textsc{else}$

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

$\land\ \textsc{unchanged}\ \langle req\_msgs,\ client\_vars,\ key\_vars,\ next\_ts \rangle$

$ServerPrewriteOptimistic \triangleq$

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

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

$\land\ \textsc{let}$

$k \triangleq req.key$

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

$\textsc{in}$

$\land\ \textsc{if}\ \exists\, w \in key\_write[k] : w.ts \geq start\_ts$

$\textsc{then}$

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

$\land\ \textsc{unchanged}\ \langle req\_msgs,\ client\_vars,\ key\_vars,\ next\_ts \rangle$

$\textsc{else}$

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

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

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

$\land\ key\_lock' = [key\_lock\ \textsc{except}\ ![k] = \{[ts \mapsto start\_ts,$

$primary \mapsto req.primary,$

$type \mapsto \text{"prewrite\_optimistic"}]\}]$

$\land\ key\_data' = [key\_data\ \textsc{except}\ ![k] = @ \cup \{start\_ts\}]$

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

$\land\ \textsc{unchanged}\ \langle req\_msgs,\ client\_vars,\ key\_write,\ next\_ts \rangle$

$ServerCommit \triangleq$

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

$\land\ req.type = \text{"commit"}$

$\land\ \textsc{let}$

$pk \triangleq req.primary$

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

$\textsc{in}$

$\textsc{if}\ \exists\, w \in key\_write[pk] : w.start\_ts = start\_ts \land w.type = \text{"write"}$

$\textsc{then}$

Key has already been committed. Do nothing.

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

$\land\ \textsc{unchanged}\ \langle req\_msgs,\ client\_vars,\ key\_vars,\ next\_ts \rangle$

$\textsc{else}$

$\textsc{if}\ \exists\, l \in key\_lock[pk] : l.ts = start\_ts$

THEN

Commit the key only if the prewrite lock exists.

$\land\ commit(pk,\ start\_ts,\ req.commit\_ts)$
$\land\ SendResp([start\_ts \mapsto start\_ts,\ type \mapsto \text{"committed"}])$
$\land\ \textsc{unchanged}\ \langle req\_msgs,\ client\_vars,\ key\_data,\ next\_ts \rangle$

ELSE

Otherwise, abort the transaction.

$\land\ SendResp([start\_ts \mapsto start\_ts,\ type \mapsto \text{"commit\_aborted"}])$
$\land\ \textsc{unchanged}\ \langle req\_msgs,\ client\_vars,\ key\_vars,\ next\_ts \rangle$

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.

$ServerCleanupStaleLock \triangleq$
$\quad \exists\, k \in KEY :$
$\quad\quad \exists\, l \in key\_lock[k] :$
$\quad\quad\quad \land\ SendReqs(\{[type \mapsto \text{"check\_txn\_status"},$
$\quad\quad\quad\quad start\_ts \mapsto l.ts,$
$\quad\quad\quad\quad primary \mapsto l.primary,$
$\quad\quad\quad\quad resolving\_pessimistic\_lock \mapsto l.type = \text{"lock\_key"}$
$\quad\quad\quad\quad]\})$
$\quad\quad\quad \land\ \textsc{unchanged}\ \langle resp\_msgs,\ client\_vars,\ key\_vars,\ next\_ts \rangle$

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.

$ServerCheckTxnStatus \triangleq$
$\quad \exists\, req \in req\_msgs :$
$\quad\quad \land\ req.type = \text{"check\_txn\_status"}$
$\quad\quad \land\ \textsc{let}$
$\quad\quad\quad pk \triangleq req.primary$
$\quad\quad\quad start\_ts \triangleq req.start\_ts$
$\quad\quad\quad pk\_lock \triangleq key\_lock[pk]$
$\quad\quad\quad committed \triangleq \{w \in key\_write[pk] : w.start\_ts = start\_ts \land w.type = \text{"write"}\}$
$\quad\quad\quad rollbacked \triangleq \{r \in key\_write[pk] : r.start\_ts = start\_ts \land r.type = \text{"rollback"}\}$
$\quad\quad \textsc{in}$
$\quad\quad\quad \textsc{if}\ \exists\, lock \in pk\_lock : lock.ts = start\_ts\ \textsc{then}$

Has a $matching(lock\_ts\ or\ start\_ts)$ lock

$\lor$

TTL expire

$\quad\quad\quad\quad \textsc{if}\ \exists\, lock \in pk\_lock :$
$\quad\quad\quad\quad\quad lock.type = \text{"lock\_key"}$
$\quad\quad\quad\quad\quad \land\ req.resolving\_pessimistic\_lock = \textsc{true}$

```
          THEN
            ∧ key_lock' = [key_lock EXCEPT ![pk] = {}]
            ∧ UNCHANGED ⟨req_msgs, resp_msgs, key_data, key_write, client_vars, next_ts⟩
          ELSE
            ∧ rollback(pk, start_ts)
            ∧ SendReqs({[type ↦ "resolve_rollbacked",
                         start_ts ↦ start_ts,
                         primary ↦ pk]})
            ∧ UNCHANGED ⟨resp_msgs, client_vars, next_ts⟩
        ELSE
           LockNotExist
         IF committed ≠ {} THEN
         ∧ SendReqs({[type ↦ "resolve_committed",
                      start_ts ↦ start_ts,
                      primary ↦ pk,
                      commit_ts ↦ w.ts] : w ∈ committed})
         ∧ UNCHANGED ⟨resp_msgs, client_vars, key_vars, next_ts⟩
         ELSE
         ∧ rollbacked ≠ {}
         ∧ rollback(pk, start_ts)
         ∧ SendReqs({[type ↦ "resolve_rollbacked",
                      start_ts ↦ start_ts,
                      primary ↦ pk]})
         ∧ UNCHANGED ⟨resp_msgs, 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 :
```

$\exists\, l \in key\_lock[k] :$
$\quad \land\ l.primary = req.primary$
$\quad \land\ l.ts = start\_ts$
$\quad \land\ rollback(k,\ start\_ts)$
$\quad \land\ \text{UNCHANGED}\ \langle msg\_vars,\ client\_vars,\ next\_ts \rangle$

Specification

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

$$
\begin{aligned}
&Next \triangleq \\
&\quad \lor\ \exists\, c \in OPTIMISTIC\_CLIENT : \\
&\qquad \lor\ ClientPrewriteOptimistic(c) \\
&\qquad \lor\ ClientPrewrited(c) \\
&\qquad \lor\ ClientCommit(c) \\
&\quad \lor\ \exists\, c \in PESSIMISTIC\_CLIENT : \\
&\qquad \lor\ ClientLockKey(c) \\
&\qquad \lor\ ClientLockedKey(c) \\
&\qquad \lor\ ClientRetryLockKey(c) \\
&\qquad \lor\ ClientPrewritePessimistic(c) \\
&\qquad \lor\ ClientPrewrited(c) \\
&\qquad \lor\ ClientCommit(c) \\
&\quad \lor\ ServerLockKey \\
&\quad \lor\ ServerPrewritePessimistic \\
&\quad \lor\ ServerPrewriteOptimistic \\
&\quad \lor\ ServerCommit \\
&\quad \lor\ ServerCleanupStaleLock \\
&\quad \lor\ ServerCheckTxnStatus \\
&\quad \lor\ ServerResolveCommitted \\
&\quad \lor\ ServerResolveRollbacked
\end{aligned}
$$

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

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.

$$
\begin{aligned}
&CommitConsistency \triangleq \\
&\quad \forall\, resp \in resp\_msgs : \\
&\qquad (resp.type = \text{"committed"}) \Rightarrow \\
&\qquad\quad \exists\, c \in CLIENT : \\
&\qquad\qquad \land\ client\_ts[c].start\_ts = resp.start\_ts \\
&\qquad\qquad \text{Primary key must be committed} \\
&\qquad\qquad \land\ keyCommitted(CLIENT\_PRIMARY[c], resp.start\_ts) \\
&\qquad\qquad \text{Secondary key must be either committed or locked by the } start\_ts \text{ of the transaction.} \\
&\qquad\qquad \land\ \forall\, k \in CLIENT\_KEY[c] : \\
&\qquad\qquad\quad (\neg\exists\, l \in key\_lock[k] : l.ts = resp.start\_ts) = \\
&\qquad\qquad\qquad keyCommitted(k, resp.start\_ts)
\end{aligned}
$$

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

$$
\begin{aligned}
&AbortConsistency \triangleq \\
&\quad \forall\, resp \in resp\_msgs : \\
&\qquad (resp.type = \text{"commit\_aborted"}) \Rightarrow \\
&\qquad\quad \forall\, c \in CLIENT : \\
&\qquad\qquad (client\_ts[c].start\_ts = resp.start\_ts) \Rightarrow \\
&\qquad\qquad\quad \neg keyCommitted(CLIENT\_PRIMARY[c], resp.start\_ts)
\end{aligned}
$$

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 $start\_ts$.

$$
\begin{aligned}
&WriteConsistency \triangleq \\
&\quad \forall\, k \in KEY : \\
&\qquad \forall\, w \in key\_write[k] : \\
&\qquad\quad \lor\ \land\ w.type = \text{"write"} \\
&\qquad\qquad \land\ w.ts > w.start\_ts
\end{aligned}
$$

$\land$ $w.start\_ts \in key\_data[k]$
$\lor$ $\land$ $w.type =$ "rollback"
$\land$ $w.ts = w.start\_ts$

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*

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

THEOREM *Safety* $\triangleq$

$$Spec \Rightarrow \Box(\land\ TypeOK \\ \land\ UniqueCommitOrAbort \\ \land\ CommitConsistency \\ \land\ AbortConsistency \\ \land\ WriteConsistency \\ \land\ UniqueLockOrWrite \\ \land\ UniqueWrite \\ \land\ SnapshotIsolation)$$