

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) $LockType \in \{“PUT”, “DELETE”, “LOCK”\} \wedge for_update_ts = 0 \equiv type = “prewrite_optimistic”$
- (2) $LockType \in \{“PUT”, “DELETE”\} \wedge for_update_ts > 0 \equiv type = “prewrite_pessimistic”$
- (3) $LockType = “PESSIMISTIC” \equiv type = “lock_key”$

VARIABLES *key_lock*

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

VARIABLES *key_write*

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

VARIABLES *client_state*

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

VARIABLES *client_ts*

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

VARIABLES *client_key*

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

VARIABLES *next_ts*

$msg_vars \triangleq \langle req_msgs, resp_msgs \rangle$
 $client_vars \triangleq \langle client_state, client_ts, client_key \rangle$
 $key_vars \triangleq \langle key_data, key_lock, key_write \rangle$
 $vars \triangleq \langle msg_vars, client_vars, key_vars, next_ts \rangle$

$SendReqs(msgs) \triangleq req_msgs' = req_msgs \cup msgs$
 $SendResp(msg) \triangleq resp_msgs' = resp_msgs \cup \{msg\}$

Type Definitions

$ReqMessages \triangleq$

- $[start_ts : Ts, primary : KEY, type : \{ "lock_key" \}, key : KEY,$
- $for_update_ts : Ts]$
- $\cup [start_ts : Ts, primary : KEY, type : \{ "prewrite_optimistic" \}, key : KEY]$
- $\cup [start_ts : Ts, primary : KEY, type : \{ "prewrite_pessimistic" \}, key : KEY]$
- $\cup [start_ts : Ts, primary : KEY, type : \{ "commit" \}, commit_ts : Ts]$
- $\cup [start_ts : Ts, primary : KEY, type : \{ "resolve_rolledback" \}]$
- $\cup [start_ts : Ts, primary : KEY, type : \{ "resolve_committed" \}, commit_ts : Ts]$
- $\cup [start_ts : Ts, primary : KEY, type : \{ "check_txn_status" \}, resolving_pessimistic_lock : BOOLEAN]$

$RespMessages \triangleq$

- $[start_ts : Ts, type : \{ "prewrited", "locked_key" \}, key : KEY]$
- $\cup [start_ts : Ts, type : \{ "lock_failed" \}, key : KEY, latest_commit_ts : Ts]$
- $\cup [start_ts : Ts, type : \{ "committed",$
- $"commit_aborted",$
- $"prewrite_aborted",$
- $"lock_key_aborted" \}]$

$TypeOK \triangleq$

- $\wedge req_msgs \in SUBSET ReqMessages$
 - $\wedge resp_msgs \in SUBSET RespMessages$
 - $\wedge key_data \in [KEY \rightarrow SUBSET Ts]$
 - $\wedge key_lock \in [KEY \rightarrow SUBSET [ts : Ts,$
 - $primary : KEY,$
 - $type : \{ "prewrite_optimistic",$
 - $"prewrite_pessimistic",$
 - $"lock_key" \}]]$
 - At most one lock in $key_lock[k]$
 - $\wedge \forall k \in KEY : Cardinality(key_lock[k]) \leq 1$
 - $\wedge key_write \in [KEY \rightarrow SUBSET ($
 - $[ts : Ts, start_ts : Ts, type : \{ "write" \}]$
 - $\cup [ts : Ts, start_ts : Ts, type : \{ "rollback" \}, protected : BOOLEAN]]$
 - $\wedge client_state \in [CLIENT \rightarrow \{ "init", "locking", "prewriting", "committing" \}]$
 - $\wedge client_ts \in [CLIENT \rightarrow [start_ts : Ts \cup \{ NoneTs \},$
 - $commit_ts : Ts \cup \{ NoneTs \},$
 - $for_update_ts : Ts \cup \{ NoneTs \}]]$
 - $\wedge client_key \in [CLIENT \rightarrow [locking : SUBSET KEY, prewriting : SUBSET KEY]]$
 - $\wedge next_ts \in Ts$
-

Client Actions

$ClientLockKey(c) \triangleq$

- $\wedge client_state[c] = "init"$
- $\wedge client_state' = [client_state \text{ EXCEPT } ![c] = "locking"]$

$$\begin{aligned}
& \wedge \text{client_ts}' = [\text{client_ts} \text{ EXCEPT } ![c].\text{start_ts} = \text{next_ts}, ![c].\text{for_update_ts} = \text{next_ts}] \\
& \wedge \text{next_ts}' = \text{next_ts} + 1 \\
& \text{Assume we need to acquire pessimistic locks for all keys} \\
& \wedge \text{client_key}' = [\text{client_key} \text{ EXCEPT } ![c].\text{locking} = \text{CLIENT_KEY}[c]] \\
& \wedge \text{SendReqs}(\{[type \mapsto \text{"lock_key"}, \\
& \quad \text{start_ts} \mapsto \text{client_ts}'[c].\text{start_ts}, \\
& \quad \text{primary} \mapsto \text{CLIENT_PRIMARY}[c], \\
& \quad \text{key} \mapsto k, \\
& \quad \text{for_update_ts} \mapsto \text{client_ts}'[c].\text{for_update_ts}] : k \in \text{CLIENT_KEY}[c]\}) \\
& \wedge \text{UNCHANGED } \langle \text{resp_msgs}, \text{key_vars} \rangle \\
\\
\text{ClientLockedKey}(c) & \triangleq \\
& \wedge \text{client_state}[c] = \text{"locking"} \\
& \wedge \exists \text{resp} \in \text{resp_msgs} : \\
& \quad \wedge \text{resp.type} = \text{"locked_key"} \\
& \quad \wedge \text{resp.start_ts} = \text{client_ts}[c].\text{start_ts} \\
& \quad \wedge \text{resp.key} \in \text{client_key}[c].\text{locking} \\
& \quad \wedge \text{client_key}' = [\text{client_key} \text{ EXCEPT } ![c].\text{locking} = @ \setminus \{\text{resp.key}\}] \\
& \quad \wedge \text{UNCHANGED } \langle \text{msg_vars}, \text{key_vars}, \text{client_ts}, \text{client_state}, \text{next_ts} \rangle \\
\\
\text{ClientRetryLockKey}(c) & \triangleq \\
& \wedge \text{client_state}[c] = \text{"locking"} \\
& \wedge \exists \text{resp} \in \text{resp_msgs} : \\
& \quad \wedge \text{resp.type} = \text{"lock_failed"} \\
& \quad \wedge \text{resp.start_ts} = \text{client_ts}[c].\text{start_ts} \\
& \quad \wedge \text{resp.latest_commit_ts} > \text{client_ts}[c].\text{for_update_ts} \\
& \quad \wedge \text{client_ts}' = [\text{client_ts} \text{ EXCEPT } ![c].\text{for_update_ts} = \text{resp.latest_commit_ts}] \\
& \quad \wedge \text{SendReqs}(\{[type \mapsto \text{"lock_key"}, \\
& \quad \quad \text{start_ts} \mapsto \text{client_ts}'[c].\text{start_ts}, \\
& \quad \quad \text{primary} \mapsto \text{CLIENT_PRIMARY}[c], \\
& \quad \quad \text{key} \mapsto \text{resp.key}, \\
& \quad \quad \text{for_update_ts} \mapsto \text{client_ts}'[c].\text{for_update_ts}]\}) \\
& \quad \wedge \text{UNCHANGED } \langle \text{resp_msgs}, \text{key_vars}, \text{client_state}, \text{client_key}, \text{next_ts} \rangle \\
\\
\text{ClientPrewritePessimistic}(c) & \triangleq \\
& \wedge \text{client_state}[c] = \text{"locking"} \\
& \wedge \text{client_key}[c].\text{locking} = \{\} \\
& \wedge \text{client_state}' = [\text{client_state} \text{ EXCEPT } ![c] = \text{"prewriting"}] \\
& \wedge \text{client_key}' = [\text{client_key} \text{ EXCEPT } ![c].\text{prewriting} = \text{CLIENT_KEY}[c]] \\
& \wedge \text{SendReqs}(\{[type \mapsto \text{"prewrite_pessimistic"}, \\
& \quad \text{start_ts} \mapsto \text{client_ts}[c].\text{start_ts}, \\
& \quad \text{primary} \mapsto \text{CLIENT_PRIMARY}[c], \\
& \quad \text{key} \mapsto k] : k \in \text{CLIENT_KEY}[c]\}) \\
& \wedge \text{UNCHANGED } \langle \text{resp_msgs}, \text{key_vars}, \text{client_ts}, \text{next_ts} \rangle \\
\\
\text{ClientPrewriteOptimistic}(c) & \triangleq
\end{aligned}$$

$$\begin{aligned}
& \wedge \text{client_state}[c] = \text{"init"} \\
& \wedge \text{client_state}' = [\text{client_state} \text{ EXCEPT } ![c] = \text{"prewriting"}] \\
& \wedge \text{client_ts}' = [\text{client_ts} \text{ EXCEPT } ![c].\text{start_ts} = \text{next_ts}] \\
& \wedge \text{next_ts}' = \text{next_ts} + 1 \\
& \wedge \text{client_key}' = [\text{client_key} \text{ EXCEPT } ![c].\text{prewriting} = \text{CLIENT_KEY}[c]] \\
& \wedge \text{SendReqs}(\{[type \mapsto \text{"prewrite_optimistic"}, \\
& \quad \text{start_ts} \mapsto \text{client_ts}'[c].\text{start_ts}, \\
& \quad \text{primary} \mapsto \text{CLIENT_PRIMARY}[c], \\
& \quad \text{key} \mapsto k] : k \in \text{CLIENT_KEY}[c]\}) \\
& \wedge \text{UNCHANGED } \langle \text{resp_msgs}, \text{key_vars} \rangle \\
\\
\text{ClientPrewritten}(c) & \triangleq \\
& \wedge \text{client_state}[c] = \text{"prewriting"} \\
& \wedge \text{client_key}[c].\text{locking} = \{\} \\
& \wedge \exists \text{resp} \in \text{resp_msgs} : \\
& \quad \wedge \text{resp.type} = \text{"prewrited"} \\
& \quad \wedge \text{resp.start_ts} = \text{client_ts}[c].\text{start_ts} \\
& \quad \wedge \text{resp.key} \in \text{client_key}[c].\text{prewriting} \\
& \quad \wedge \text{client_key}' = [\text{client_key} \text{ EXCEPT } ![c].\text{prewriting} = @ \setminus \{\text{resp.key}\}] \\
& \quad \wedge \text{UNCHANGED } \langle \text{msg_vars}, \text{key_vars}, \text{client_ts}, \text{client_state}, \text{next_ts} \rangle \\
\\
\text{ClientCommit}(c) & \triangleq \\
& \wedge \text{client_state}[c] = \text{"prewriting"} \\
& \wedge \text{client_key}[c].\text{prewriting} = \{\} \\
& \wedge \text{client_state}' = [\text{client_state} \text{ EXCEPT } ![c] = \text{"committing"}] \\
& \wedge \text{client_ts}' = [\text{client_ts} \text{ EXCEPT } ![c].\text{commit_ts} = \text{next_ts}] \\
& \wedge \text{next_ts}' = \text{next_ts} + 1 \\
& \wedge \text{SendReqs}(\{[type \mapsto \text{"commit"}, \\
& \quad \text{start_ts} \mapsto \text{client_ts}'[c].\text{start_ts}, \\
& \quad \text{primary} \mapsto \text{CLIENT_PRIMARY}[c], \\
& \quad \text{commit_ts} \mapsto \text{client_ts}'[c].\text{commit_ts}]\}) \\
& \wedge \text{UNCHANGED } \langle \text{resp_msgs}, \text{key_vars}, \text{client_key} \rangle
\end{aligned}$$

Server Actions

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

$$\begin{aligned}
\text{commit}(pk, \text{start_ts}, \text{commit_ts}) & \triangleq \\
& \exists l \in \text{key_lock}[pk] : \\
& \quad \wedge l.\text{ts} = \text{start_ts} \\
& \quad \wedge \text{key_lock}' = [\text{key_lock} \text{ EXCEPT } ![pk] = \{\}] \\
& \quad \wedge \text{key_write}' = [\text{key_write} \text{ EXCEPT } ![pk] = @ \cup \{[ts \mapsto \text{commit_ts}, \\
& \quad \quad \quad \text{type} \mapsto \text{"write"}, \\
& \quad \quad \quad \text{start_ts} \mapsto \text{start_ts}]\}]
\end{aligned}$$

Rollback the transaction that starts at start_ts on key k .

$$\text{rollback}(k, \text{start_ts}) \triangleq$$

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.

$$protected \triangleq \vee \exists l \in key_lock[k] : \\ \wedge l.ts = start_ts \\ \wedge l.primary = k \\ \wedge l.type \in \{ "lock_key", "prewrite_pessimistic" \} \\ \vee \exists l \in key_lock[k] : l.ts \neq start_ts \\ \vee key_lock[k] = \{ \}$$

IN

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

\wedge IF $\exists l \in key_lock[k] : l.ts = start_ts$
 THEN $key_lock' = [key_lock \text{ EXCEPT } ![k] = \{ \}]$
 ELSE UNCHANGED key_lock
 $\wedge key_data' = [key_data \text{ EXCEPT } ![k] = @ \setminus \{ start_ts \}]$
 \wedge IF
 $\wedge \neg \exists w \in key_write[k] : w.ts = start_ts$
 THEN
 $key_write' = [key_write \text{ EXCEPT } \\ ![k] = \\ \text{collapse rollback} \\ (@ \setminus \{ w \in @ : w.type = "rollback" \wedge \neg w.protected \wedge w.ts < start_ts \}) \\ \text{write rollback record} \\ \cup \{ [ts \mapsto start_ts, \\ start_ts \mapsto start_ts, \\ type \mapsto "rollback", \\ protected \mapsto protected] \}]$
 ELSE
 UNCHANGED $\langle key_write \rangle$

$ServerLockKey \triangleq$
 $\exists req \in req_msgs :$
 $\wedge req.type = "lock_key"$
 \wedge 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.
 $\wedge key_lock[k] = \{ \}$
 \wedge LET
 $latest_write \triangleq \{ w \in key_write[k] : \forall w2 \in key_write[k] : w.ts \geq w2.ts \}$

$all_commits \triangleq \{w \in key_write[k] : w.type = \text{"write"}\}$
 $latest_commit \triangleq \{w \in all_commits : \forall w2 \in all_commits : w.ts \geq w2.ts\}$

IN
 IF $\exists w \in key_write[k] : w.start_ts = start_ts \wedge w.type = \text{"rollback"}$
 THEN
 If corresponding rollback record is found, which indicates that the transaction is rolled back, abort the transaction.
 $\wedge SendResp([start_ts \mapsto start_ts, type \mapsto \text{"lock_key_aborted"}])$
 $\wedge UNCHANGED \langle req_msgs, client_vars, key_vars, next_ts \rangle$
 ELSE
 Acquire pessimistic lock only if for_update_ts of req is greater or equal to the latest "write" record.
 Because if the latest record is "write", it means that a new version is committed after for_update_ts , which violates Read Committed guarantee.
 $\vee \wedge \neg \exists w \in latest_commit : w.ts > req.for_update_ts$
 $\wedge key_lock' = [key_lock \text{ EXCEPT } ![k] = \{[ts \mapsto start_ts,$
 $primary \mapsto req.primary,$
 $type \mapsto \text{"lock_key"}]\}]$
 $\wedge SendResp([start_ts \mapsto start_ts, type \mapsto \text{"locked_key"}, key \mapsto k])$
 $\wedge UNCHANGED \langle req_msgs, client_vars, key_data, key_write, next_ts \rangle$
 Otherwise, reject the request and let client to retry with new for_update_ts .
 $\vee \exists w \in latest_commit :$
 $\wedge w.ts > req.for_update_ts$
 $\wedge SendResp([start_ts \mapsto start_ts,$
 $type \mapsto \text{"lock_failed"},$
 $key \mapsto k,$
 $latest_commit_ts \mapsto w.ts])$
 $\wedge UNCHANGED \langle req_msgs, client_vars, key_vars, next_ts \rangle$

$ServerPrewritePessimistic \triangleq$
 $\exists req \in req_msgs :$
 $\wedge req.type = \text{"prewrite_pessimistic"}$
 $\wedge LET$
 $k \triangleq req.key$
 $start_ts \triangleq req.start_ts$
 IN
 Pessimistic prewrite is allowed only if pessimistic lock is acquired, otherwise abort the transaction.
 $\wedge IF \exists l \in key_lock[k] : l.ts = start_ts$
 THEN
 $\wedge key_lock' = [key_lock \text{ EXCEPT } ![k] = \{[ts \mapsto start_ts,$
 $primary \mapsto req.primary,$

$$\begin{aligned}
& \text{type} \mapsto \text{"prewrite_pessimistic"} \}}] \\
& \wedge \text{key_data}' = [\text{key_data} \text{ EXCEPT } ![k] = @ \cup \{ \text{start_ts} \}] \\
& \wedge \text{SendResp}([\text{start_ts} \mapsto \text{start_ts}, \text{type} \mapsto \text{"prewrited"}, \text{key} \mapsto k]) \\
& \wedge \text{UNCHANGED } \langle \text{req_msgs}, \text{client_vars}, \text{key_write}, \text{next_ts} \rangle \\
& \text{ELSE} \\
& \wedge \text{SendResp}([\text{start_ts} \mapsto \text{start_ts}, \text{type} \mapsto \text{"prewrite_aborted"}]) \\
& \wedge \text{UNCHANGED } \langle \text{req_msgs}, \text{client_vars}, \text{key_vars}, \text{next_ts} \rangle \\
\text{ServerPrewriteOptimistic} & \triangleq \\
& \exists \text{req} \in \text{req_msgs} : \\
& \wedge \text{req.type} = \text{"prewrite_optimistic"} \\
& \wedge \text{LET} \\
& \quad k \triangleq \text{req.key} \\
& \quad \text{start_ts} \triangleq \text{req.start_ts} \\
& \text{IN} \\
& \wedge \text{IF } \exists w \in \text{key_write}[k] : w.ts \geq \text{start_ts} \\
& \quad \text{THEN} \\
& \quad \wedge \text{SendResp}([\text{start_ts} \mapsto \text{start_ts}, \text{type} \mapsto \text{"prewrite_aborted"}]) \\
& \quad \wedge \text{UNCHANGED } \langle \text{req_msgs}, \text{client_vars}, \text{key_vars}, \text{next_ts} \rangle \\
& \quad \text{ELSE} \\
& \quad \text{Optimistic prewrite is allowed only if no stale lock exists. If} \\
& \quad \text{there is one, wait until } \text{ServerCleanupStaleLock} \text{ to clean it up.} \\
& \quad \wedge \vee \text{key_lock}[k] = \{ \} \\
& \quad \quad \vee \exists l \in \text{key_lock}[k] : l.ts = \text{start_ts} \\
& \quad \wedge \text{key_lock}' = [\text{key_lock} \text{ EXCEPT } ![k] = \{ [ts \mapsto \text{start_ts}, \\
& \quad \quad \quad \text{primary} \mapsto \text{req.primary}, \\
& \quad \quad \quad \text{type} \mapsto \text{"prewrite_optimistic"}] \}}] \\
& \quad \wedge \text{key_data}' = [\text{key_data} \text{ EXCEPT } ![k] = @ \cup \{ \text{start_ts} \}] \\
& \quad \wedge \text{SendResp}([\text{start_ts} \mapsto \text{start_ts}, \text{type} \mapsto \text{"prewrited"}, \text{key} \mapsto k]) \\
& \quad \wedge \text{UNCHANGED } \langle \text{req_msgs}, \text{client_vars}, \text{key_write}, \text{next_ts} \rangle \\
\text{ServerCommit} & \triangleq \\
& \exists \text{req} \in \text{req_msgs} : \\
& \wedge \text{req.type} = \text{"commit"} \\
& \wedge \text{LET} \\
& \quad pk \triangleq \text{req.primary} \\
& \quad \text{start_ts} \triangleq \text{req.start_ts} \\
& \text{IN} \\
& \text{IF } \exists w \in \text{key_write}[pk] : w.start_ts = \text{start_ts} \wedge w.type = \text{"write"} \\
& \quad \text{THEN} \\
& \quad \text{Key has already been committed. Do nothing.} \\
& \quad \wedge \text{SendResp}([\text{start_ts} \mapsto \text{start_ts}, \text{type} \mapsto \text{"committed"}]) \\
& \quad \wedge \text{UNCHANGED } \langle \text{req_msgs}, \text{client_vars}, \text{key_vars}, \text{next_ts} \rangle \\
& \quad \text{ELSE} \\
& \quad \text{IF } \exists l \in \text{key_lock}[pk] : l.ts = \text{start_ts}
\end{aligned}$$

THEN

Commit the key only if the prewrite lock exists.
 $\wedge \text{commit}(pk, start_ts, req.commit_ts)$
 $\wedge \text{SendResp}([start_ts \mapsto start_ts, type \mapsto \text{"committed"}])$
 $\wedge \text{UNCHANGED } \langle req_msgs, client_vars, key_data, next_ts \rangle$

ELSE

Otherwise, abort the transaction.
 $\wedge \text{SendResp}([start_ts \mapsto start_ts, type \mapsto \text{"commit_aborted"}])$
 $\wedge \text{UNCHANGED } \langle req_msgs, client_vars, key_vars, next_ts \rangle$

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.

$\text{ServerCleanupStaleLock} \triangleq$

$\exists k \in KEY :$

$\exists l \in key_lock[k] :$

$\wedge \text{SendReqs}(\{[type \mapsto \text{"check_txn_status"},$
 $start_ts \mapsto l.ts,$
 $primary \mapsto l.primary,$
 $resolving_pessimistic_lock \mapsto l.type = \text{"lock_key"}]$
 $\})$

$\wedge \text{UNCHANGED } \langle resp_msgs, client_vars, key_vars, next_ts \rangle$

Clean up stale locks by checking the status of the primary key. It can be divided into two cases: pk_lock exists or not. Note that it is hard to reflect the TTL-related things into the TLA+ spec, so the TTL expired actions is done without time checkings.

$\text{ServerCheckTxnStatus} \triangleq$

$\exists req \in req_msgs :$

$\wedge req.type = \text{"check_txn_status"}$

$\wedge \text{LET}$

$pk \triangleq req.primary$

$start_ts \triangleq req.start_ts$

$committed \triangleq \{w \in key_write[pk] : w.start_ts = start_ts \wedge w.type = \text{"write"}\}$

IN

IF $\exists lock \in key_lock[pk] : lock.ts = start_ts$

Found the matching lock whose TTL is expired.

THEN

IF

Pessimistic lock will be unlocked directly without rollback record.

$\exists lock \in key_lock[pk] :$

$\wedge lock.ts = start_ts$

$\wedge lock.type = \text{"lock_key"}$

$\wedge req.resolving_pessimistic_lock = \text{TRUE}$

```

THEN
   $\wedge key\_lock' = [key\_lock \text{ EXCEPT } ![pk] = \{\}]$ 
   $\wedge \text{UNCHANGED } \langle msg\_vars, key\_data, key\_write, client\_vars, next\_ts \rangle$ 
ELSE
   $\wedge \text{rollback}(pk, start\_ts)$ 
   $\wedge \text{SendReqs}(\{[type \mapsto \text{"resolve\_rollbacked"},$ 
     $start\_ts \mapsto start\_ts,$ 
     $primary \mapsto pk]\})$ 
   $\wedge \text{UNCHANGED } \langle resp\_msgs, client\_vars, next\_ts \rangle$ 
  Lock not found or  $start\_ts$  on the lock mismatches.
ELSE
  IF  $committed \neq \{\}$  THEN
     $\wedge \text{SendReqs}(\{[type \mapsto \text{"resolve\_committed"},$ 
       $start\_ts \mapsto start\_ts,$ 
       $primary \mapsto pk,$ 
       $commit\_ts \mapsto w.ts] : w \in committed\})$ 
     $\wedge \text{UNCHANGED } \langle resp\_msgs, client\_vars, key\_vars, next\_ts \rangle$ 
  ELSE IF  $req.resolving\_pessimistic\_lock = \text{TRUE}$  THEN
     $\wedge \text{UNCHANGED } \langle vars \rangle$ 
  ELSE
     $\wedge \text{rollback}(pk, start\_ts)$ 
     $\wedge \text{SendReqs}(\{[type \mapsto \text{"resolve\_rollbacked"},$ 
       $start\_ts \mapsto start\_ts,$ 
       $primary \mapsto pk]\})$ 
     $\wedge \text{UNCHANGED } \langle resp\_msgs, client\_vars, next\_ts \rangle$ 

```

$ServerResolveCommitted \triangleq$

```

 $\exists req \in req\_msgs :$ 
   $\wedge req.type = \text{"resolve\_committed"}$ 
   $\wedge \text{LET}$ 
     $start\_ts \triangleq req.start\_ts$ 
  IN
     $\exists k \in KEY :$ 
       $\exists l \in key\_lock[k] :$ 
         $\wedge l.primary = req.primary$ 
         $\wedge l.ts = start\_ts$ 
         $\wedge commit(k, start\_ts, req.commit\_ts)$ 
         $\wedge \text{UNCHANGED } \langle msg\_vars, client\_vars, key\_data, next\_ts \rangle$ 

```

$ServerResolveRollbacked \triangleq$

```

 $\exists req \in req\_msgs :$ 
   $\wedge req.type = \text{"resolve\_rollbacked"}$ 
   $\wedge \text{LET}$ 
     $start\_ts \triangleq req.start\_ts$ 
  IN

```

$$\begin{aligned}
& \exists k \in KEY : \\
& \quad \exists l \in key_lock[k] : \\
& \quad \quad \wedge l.primary = req.primary \\
& \quad \quad \wedge l.ts = start_ts \\
& \quad \quad \wedge rollback(k, start_ts) \\
& \quad \quad \wedge UNCHANGED \langle msg_vars, client_vars, next_ts \rangle
\end{aligned}$$

Specification

$$\begin{aligned}
Init & \triangleq \\
& \wedge next_ts = 1 \\
& \wedge req_msgs = \{\} \\
& \wedge resp_msgs = \{\} \\
& \wedge client_state = [c \in CLIENT \mapsto \text{"init"}] \\
& \wedge client_key = [c \in CLIENT \mapsto [locking \mapsto \{\}, prewriting \mapsto \{\}]] \\
& \wedge client_ts = [c \in CLIENT \mapsto [start_ts \mapsto NoneTs, \\
& \quad \quad \quad commit_ts \mapsto NoneTs, \\
& \quad \quad \quad for_update_ts \mapsto NoneTs]] \\
& \wedge key_lock = [k \in KEY \mapsto \{\}] \\
& \wedge key_data = [k \in KEY \mapsto \{\}] \\
& \wedge key_write = [k \in KEY \mapsto \{\}]
\end{aligned}$$

$$\begin{aligned}
Next & \triangleq \\
& \vee \exists c \in OPTIMISTIC_CLIENT : \\
& \quad \vee ClientPrewriteOptimistic(c) \\
& \quad \vee ClientPrewritten(c) \\
& \quad \vee ClientCommit(c) \\
& \vee \exists c \in PESSIMISTIC_CLIENT : \\
& \quad \vee ClientLockKey(c) \\
& \quad \vee ClientLockedKey(c) \\
& \quad \vee ClientRetryLockKey(c) \\
& \quad \vee ClientPrewritePessimistic(c) \\
& \quad \vee ClientPrewritten(c) \\
& \quad \vee ClientCommit(c) \\
& \vee ServerLockKey \\
& \vee ServerPrewritePessimistic \\
& \vee ServerPrewriteOptimistic \\
& \vee ServerCommit \\
& \vee ServerCleanupStaleLock \\
& \vee ServerCheckTxnStatus \\
& \vee ServerResolveCommitted \\
& \vee ServerResolveRollbacked
\end{aligned}$$

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

Consistency Invariants

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

$$\begin{aligned} keyCommitted(k, start_ts) &\triangleq \\ \exists w \in key_write[k] : & \\ \quad \wedge w.start_ts = start_ts & \\ \quad \wedge w.type = \text{“write”} & \end{aligned}$$

A transaction can’t be both committed and aborted.

$$\begin{aligned} UniqueCommitOrAbort &\triangleq \\ \forall resp, resp2 \in resp_msgs : & \\ (resp.type = \text{“committed”}) \wedge (resp2.type = \text{“commit_aborted”}) \Rightarrow & \\ resp.start_ts \neq resp2.start_ts & \end{aligned}$$

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.

$$\begin{aligned} CommitConsistency &\triangleq \\ \forall resp \in resp_msgs : & \\ (resp.type = \text{“committed”}) \Rightarrow & \\ \exists c \in CLIENT : & \\ \quad \wedge client_ts[c].start_ts = resp.start_ts & \\ \quad \text{Primary key must be committed} & \\ \quad \wedge keyCommitted(CLIENT_PRIMARY[c], resp.start_ts) & \\ \quad \text{Secondary key must be either committed or locked by the} & \\ \quad \text{start_ts of the transaction.} & \\ \quad \wedge \forall k \in CLIENT_KEY[c] : & \\ \quad (\neg \exists l \in key_lock[k] : l.ts = resp.start_ts) = & \\ \quad keyCommitted(k, resp.start_ts) & \end{aligned}$$

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

$$\begin{aligned} AbortConsistency &\triangleq \\ \forall resp \in resp_msgs : & \\ (resp.type = \text{“commit_aborted”}) \Rightarrow & \\ \forall c \in CLIENT : & \\ \quad (client_ts[c].start_ts = resp.start_ts) \Rightarrow & \\ \quad \neg keyCommitted(CLIENT_PRIMARY[c], resp.start_ts) & \end{aligned}$$

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 $start_ts$.

$$\begin{aligned} WriteConsistency &\triangleq \\ \forall k \in KEY : & \\ \quad \forall w \in key_write[k] : & \\ \quad \quad \vee \wedge w.type = \text{“write”} & \\ \quad \quad \wedge w.ts > w.start_ts & \end{aligned}$$

$$\begin{aligned} & \wedge w.start_ts \in key_data[k] \\ \vee & \wedge w.type = \text{"rollback"} \\ & \wedge w.ts = w.start_ts \end{aligned}$$

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

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

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

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

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.

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

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

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

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

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

PROOF BY *CommitConsistency*, *MsgMonotonicity*

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

PROOF BY *NextTsConsistency*, *MsgTsConsistency*

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

PROOF BY *AbortConsistency*, *MsgMonotonicity*

$$\begin{aligned} \text{SnapshotIsolation} &\triangleq \wedge \text{CommitConsistency} \\ &\quad \wedge \text{AbortConsistency} \\ &\quad \wedge \text{NextTsMonotonicity} \\ &\quad \wedge \text{MsgMonotonicity} \\ &\quad \wedge \text{MsgTsConsistency} \end{aligned}$$

THEOREM *Safety* \triangleq

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