State transfer for TOB- and TOM- (a.k.a. TOA) based (full/partial) replication

Reminders/Premises

1. TOB and TOM based replication protocols can commit transactions in one phase (1P-TO) or two phases (2P-TO) depending on which consistency level is set and on whether ISPN is the only XA resource in the transaction or not. Figure 1 reports the cases in which 1P-TO vs and 2P-TO can be used.
2. With TOB/TOM we have a single thread that processes incoming messages delivered on the TO channel. We call it *TO-Thread.*
    1. In 1PC, the *TO-Thread* is also the thread that actually perform the write-back/commit/flush of the transaction.
    2. With 2PC, the *TO-Thread* simply schedules the validation of the messages to a pool of threads (these are internally synchronized via a latch-based scheme that ensures that txs are validated according to the TO order, see TO Based Documentation, the “how it works” chapter)
3. Analogously to the current state transfer protocol for 2PC-based replication, also with TOB/TOM committing transaction acquires the state transfer lock before TOB-sending the prepare command (which is actually a commit if we are using 1PC), and then releases it afterwards.

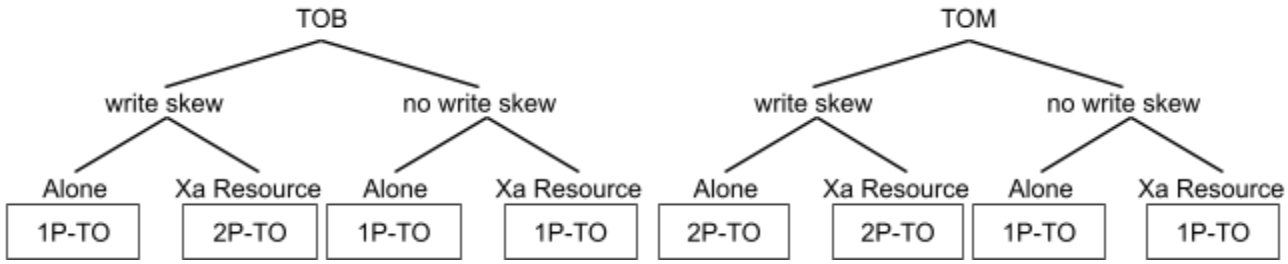

Figure 1. Scenarios in which 1P-TO and 2P-TO can be used.

The state transfer protocols

In the following we describe the proposed State transfer protocol for TOB/TOM depending on whether we have 1PC or 2PC.

1P-TO case

Figure 2 shows a diagram illustrating the proposed state transfer protocol for 1P-TO. Below we describe it and provide some hints on its correctness.

1. when a node joins, the coordinator sends the Prepare_View in total order to **everybody** (also with distribution).
2. when the Prepare_view is delivered at a node n, the *TO-Thread* acquires the StateTransferLock:
    a. this ensure that while the ST is in progress the transactions requesting to commit on n will block.
    b. on the other hand we have to deal with TOB requests that have been sent in current view (say view i) and will be delivered after the state transfer has completed (in view i+1 if the view change succeeds, or in view i if the view change aborts). This is dealt with in point 5**.**
4. the push state is performed (as in current code):
    a. what happens if a node receives the push state but has not yet received the Prepare_view message? We need to hold the push state till the Prepare_view is delivered (is this done already? How?)
5. when a node finishes to push its state, it acks the coordinator (as in current code):
    - Note that at this point JGroups will start delivering the messages sent in total order either in the previous view or in the new view. The *TO-Thread* needs to block until the new view is installed (point 5).
5. the coordinator sends the Commit_View (or rollback) (as in current code). From this point on the *TO-Thread* can re-start processing transactions, and we need to distinguish two cases (see point 4):
    a. the delivered msg was originated in the current view, we process it normally
    b. the delivered msg was originated in the previous view, so we have these cases:
        i. the msg was originated by a different node, we discard it.
        ii. the msg was originated by this node, we re-TO* send it.
        iii. ***optimization***: if TOB and view change only has processes leaving, consider it as originated in current view

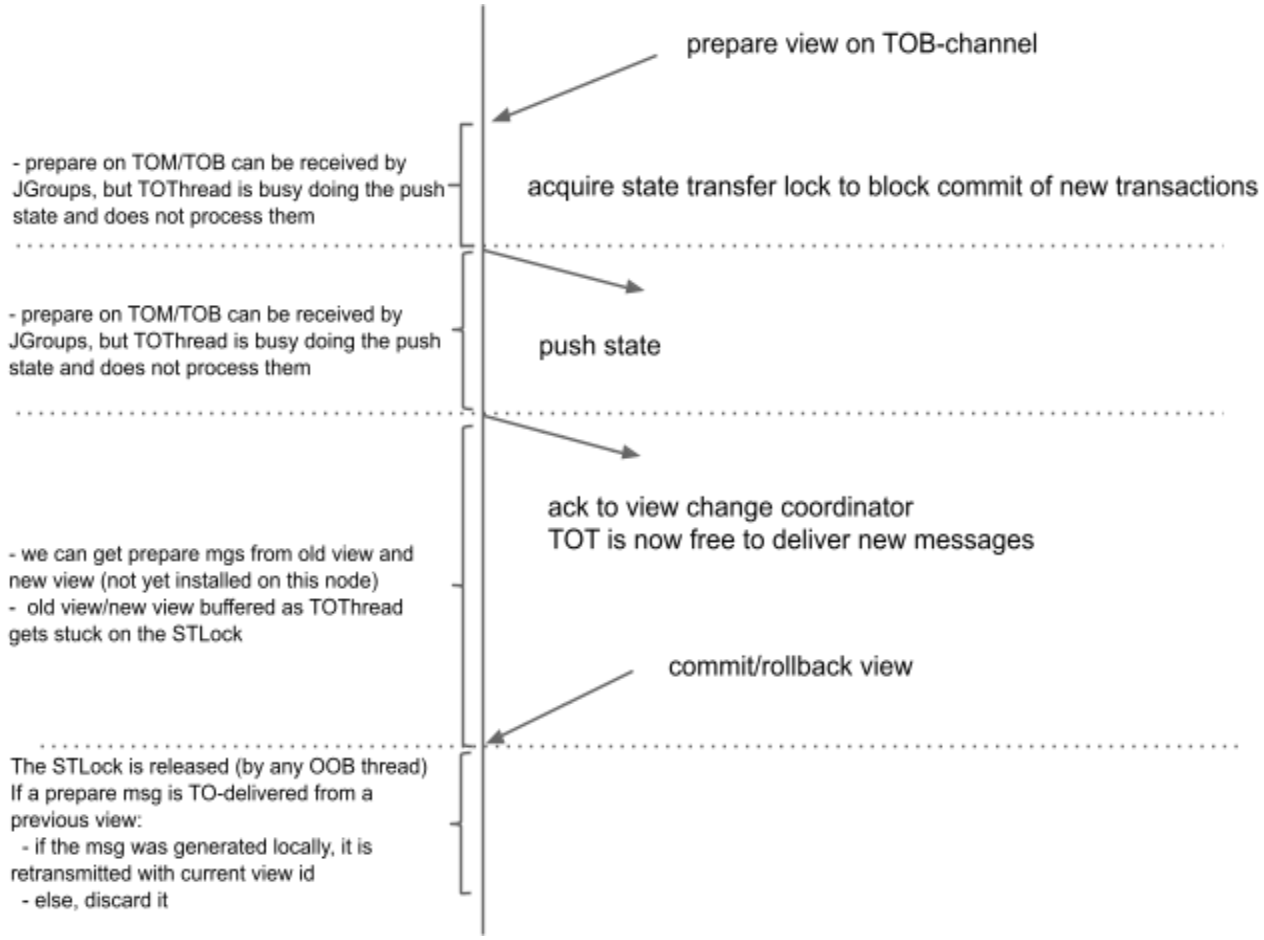

Figure 2: 1P-TO state transfer protocol

2P-TO case

The behavior of the state transfer protocol for 2P-TO is shown in Figure 3. Below we discuss why and how it differs from the protocol proopsed for the 1P-TO case.

With respect to 1P-TO in this case we need to deal with 2 extra problems due to the fact that, in this scenario, we have additional messages (commit and abort) sent during the second phase:

1. When the Prepare_view is delivered at a node n, there can be TO-delivered transactions which have not yet received their final outcome. We call these xacts *pending*. Note that, instead, with 1P-TO, a transaction's outcome was immediately determined when the corresponding prepare message was TO-delivered.
   We choose a simple/safe approach to cope with this issue ("*premature optimization is the root of all evil*" :-)), which consists in waiting for all pending transactions, so that the state transfer only starts when we have finalized these transactions. Note that it is possible to show that pending transactions are all eventually finalized (hence ensuring that this phase eventually terminates) because, as in 2.a of 1P-TO, when the Prepare_view is delivered the

StateTransferLock is acquired, which prevents new transactions to enter the commit phase.

2. While a node *n* waits for the Commit_view/Abort_view from the coordinator, it can receive prepare/commit/abort messages both from view i, and from view i+1. In fact, the Commit/Abort_View can be delivered on *n'* earlier than on node *n*, so *n'* can start processing transaction in view i+1 (or in the new instance of view *i*, if view i+1 is aborted) while *n* is still transiting from view i to i+1.
   We deal with these scenarios as follows:
   - Prepare messages from view *i*,*i+1* ⇒ hold them until the state transfer is finished (using state transfer lock), then apply the same rules described in point 5 of 1P-TO protocol
   - Abort/Commit messages from, view i, i+1 ⇒ these messages have been received before the corresponding prepare messages have been processed. This is a case we already handle in the TOB/TOM code by either i) aborting immediately the xact, or ii) waiting to flush the data till the corresponding prepare is received. We can do the same here.

For the sake of clarity below we report the entire 2P-TO state transfer protocol highlighting in green the parts that differ w.r.t 1P-TO

1. when a node joins, the coordinator sends the Prepare_View in total order to everybody (also with distribution).
2. when the Prepare_view is delivered at a node n, the *TO-Thread* acquires the StateTransferLock:
   a. this ensure that while the ST is in progress the transactions requesting to commit on n will block.
   b. wait until all pending transactions are finalized
   c. on the other hand we have to deal with TOB requests that have been sent in current view (say view i) and will be delivered after the state transfer has completed (in view i+1 if the view change succeeds, or in view i if the view change aborts). This is dealt with in point 5.
4. the push state is performed (as in current code):
   a. what happens if a node receives the push state but has not yet received the Prepare_view message? We need to hold the push state till the Prepare_view is delivered (is this done already? How?)
4. when a node finishes to push its state, it acks the coordinator and waits for a Commit_View/Rollback View(as in current code):
   - Note that at this point JGroups will start delivering the messages sent in total order either in the previous view or in the new view. The *TO-Thread* needs to block (not delivering further prepare messages) until the new view is installed (point 5).
   - Note that at this point, we can start receiving commit/abort messages. In this scenario, it is safe to deliver the commit/abort messages we receive (either in the new view or in the current one). This means that the TO-Thread has not yet delivered the corresponding prepare, and this scenario is already handled in the normal TOB/TOM

replication code.

5. the coordinator sends the Commit_View (or rollback) (as in current code). From this point on the *TO-Thread* can re-start processing transactions, and we need to distinguish two cases (see point 4):
    a. the delivered msg was originated in the current view, we process it normally
    b. the delivered msg was originated in the previous view, so we have these cases:
        i. the msg was originated by a different node, we discard it.
        ii. the msg was originated by this node, we re-TO* send it.
        iii. ***optimization***: if TOB and view change only has processes leaving, consider it as originated in current view

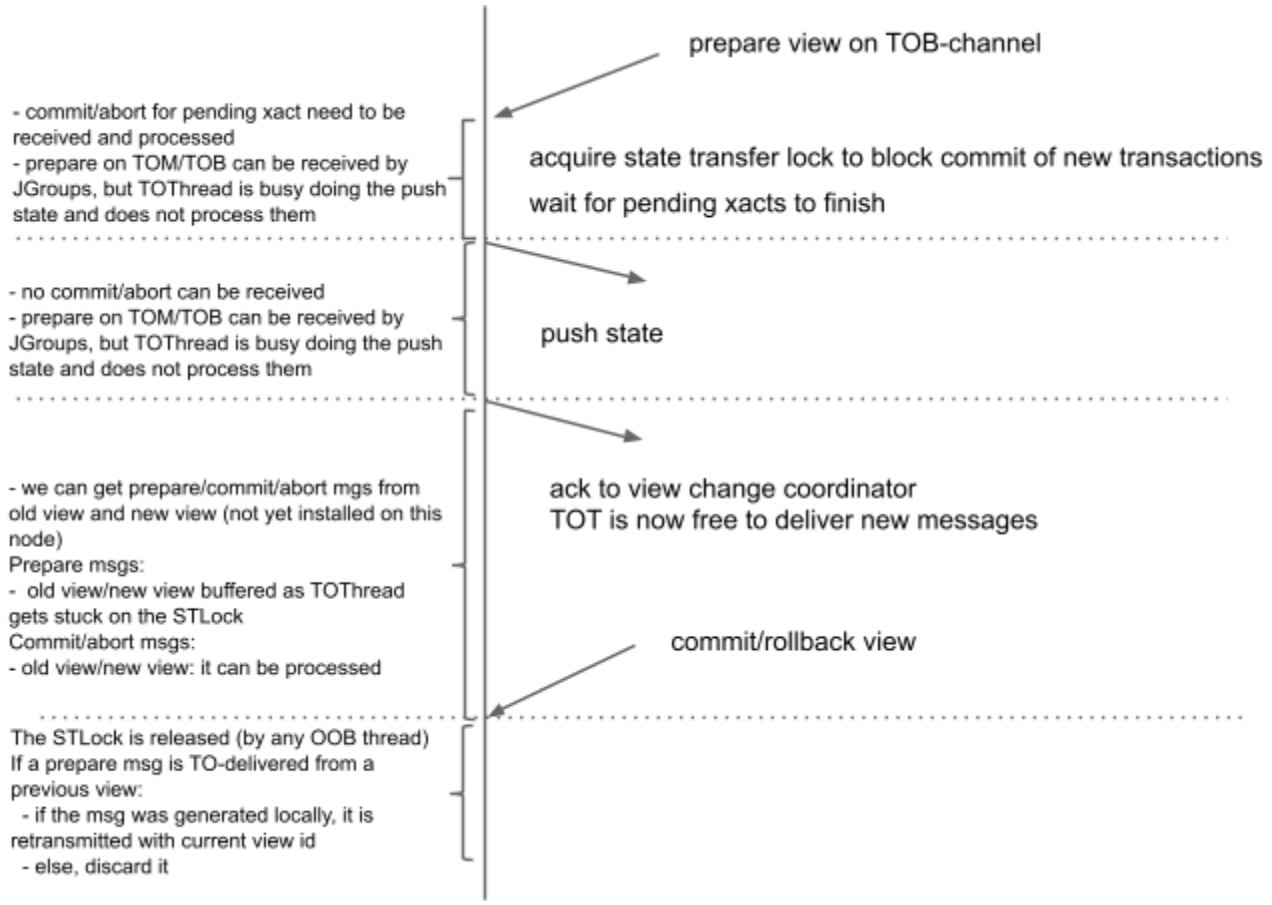

Figure 3: 2P-TO state transfer protocol