Transaction Commit Specifications

Leslie Lamport

Microsoft Research

23 February 2016 at 18:22 PST

The specifications and most of the explanatory text are extracts from *Consensus on Transaction Commit* by Jim Gray and Leslie Lamport, which appeared in *ACM Transactions on Database Systems 31*, 1 (2006), 133-160. It is available at

`http://research.microsoft.com/apps/pubs/default.aspx?id=64636`

Contents

1 Introduction

A distributed transaction consists of a number of operations, performed at multiple sites, terminated by a request to commit or abort the transaction. The sites then use a transaction commit protocol to decide whether the transaction is committed or aborted. The transaction can be committed only if all sites are willing to commit it. Achieving this all-or-nothing atomicity property in a distributed system is not trivial. The requirements for transaction commit are stated precisely in Section 2.

The classic transaction commit protocol is Two-Phase Commit, described in Section 3. It uses a single coordinator to reach agreement. The failure of that coordinator can cause the protocol to block, with no process knowing the outcome, until the coordinator is repaired. In Section 4, we use the Paxos consensus algorithm to obtain a transaction commit protocol that uses multiple coordinators; it makes progress if a majority of the coordinators are working.

We assume that algorithms are executed by a collection of processes that communicate using messages. Each process executes at a node in a network. Different processes may execute on the same node. Our failure model assumes that nodes, and hence their processes, can fail; messages can be lost or duplicated, but not corrupted. Any process executing at a failed node simply stops performing actions; it does not perform incorrect actions and does not forget its state.

In general, there are two kinds of correctness properties that an algorithm must satisfy: safety and liveness. Intuitively, a safety property describes what is allowed to happen, and a liveness property describes what must happen. Liveness is here discussed only informally. Our formal specifications ignore liveness and consider only safety. Our algorithms are asynchronous in the sense that their safety properties do not depend on timely execution by processes or on bounded message delay.

The main body of this paper informally describes transaction commit and our two protocols. The Appendix contains formal TLA$^+$ [**?**] specifications of their safety properties—that is, specifications omitting assumptions and requirements involving progress or real-time constraints.

2 Transaction Commit

In a distributed system, a transaction is performed by a collection of processes called resource managers (RMs), each executing on a different node.

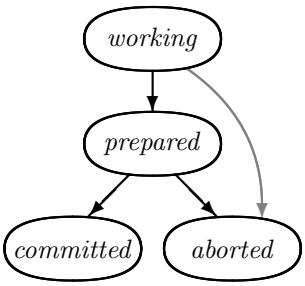

Figure 1: The state-transition diagram for a resource manager. It begins in the *working* state, in which it may decide that it wants to abort or commit. It aborts by simply entering the *aborted* state. If it decides to commit, it enters the *prepared* state. From this state, it can commit only if all other resource managers also decided to commit.

The transaction ends when one of the resource managers issues a request either to commit or to abort the transaction. For the transaction to be committed, each participating RM must be willing to commit it. Otherwise, the transaction must be aborted. Prior to the commit request, any RM may spontaneously decide to abort its part of the transaction. The fundamental requirement is that all RMs must agree on whether the transaction is committed or aborted.[1] We assume a fixed set of participating RMs determined in advance.

We abstract the requirements of a transaction commit protocol as follows. (The requirements are summarized in the state-transition diagram of Figure 1.) We assume a set of RM processes, each beginning in a *working* state. The goal of the protocol is for the RMs all to reach a *committed* state or all to reach an *aborted* state. Two safety requirements of the protocol are:

Stability Once an RM has entered the *committed* or *aborted* state, it remains in that state forever.

Consistency It is impossible for one RM to be in the *committed* state and another to be in the *aborted* state.

These two properties imply that, once an RM enters the *committed* state, no other RM can enter the *aborted* state, and vice versa.

Each RM also has a *prepared* state. We require that

- An RM can enter the *committed* state only after all RMs have been in the *prepared* state.

These requirements imply that the transaction can commit, meaning that all RMs reach the *committed* state, only by the following sequence of events:

[1] In some descriptions of transaction commit, there is a client process that ends the transaction and must also learn if it is committed. We consider such a client to be one of the RMs.

- All the RMs enter the *prepared* state, in any order.
- All the RMs enter the *committed* state, in any order.

The protocol allows the following event that prevents the transaction from committing:

- Any RM in the *working* state can enter the *aborted* state.

The stability and consistency conditions imply that this spontaneous abort event cannot occur if some RM has entered the *committed* state. In practice, an RM will abort when it learns that a failure has occurred that prevents the transaction from committing. However, our abstract representation of the problem permits an RM spontaneously to enter the *aborted* state.

The goal of the algorithm is for all RMs to reach the *committed* or *aborted* state, but this cannot be achieved in a non-trivial way if RMs can fail or become isolated through communication failure. (A trivial solution is one in which all RMs always abort.) Moreover, a classic theorem of Fischer, Lynch, and Paterson implies that a deterministic, purely asynchronous algorithm cannot satisfy the stability and consistency conditions and still guarantee progress in the presence of even a single fault.

We can more precisely specify a transaction commit protocol by specifying its set of legal behaviors, where a behavior is a sequence of system states. We specify the safety properties with an initial predicate and a next-state relation that describes all possible steps (state transitions). The initial predicate asserts that all RMs are in the *working* state. To define the next-state relation, we first define two state predicates:

canCommit True iff all RMs are in the *prepared* or *committed* state.

notCommitted True iff no RM is in the *committed* state.

The next-state relation asserts that each step consists of one of the following two actions performed by a single RM:

Prepare The RM can change from the *working* state to the *prepared* state.

Decide If the RM is in the *prepared* state and *canCommit* is true, then it can transition to the *committed* state; and if the RM is in either the *working* or *prepared* state and *notCommitted* is true, then it can transition to the *aborted* state.

The TLA$^+$ specification is in Section A.1 on page 10.

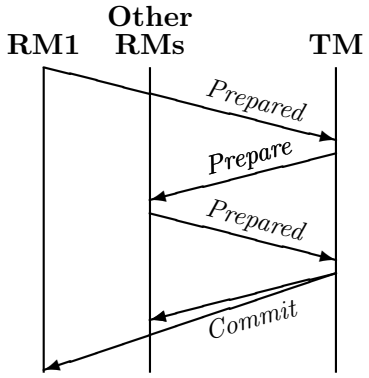

Figure 2: The message flow for Two-Phase Commit in the normal failure-free case, where RM1 is the first RM to enter the *prepared* state.

3 Two-Phase Commit

3.1 The Protocol

The Two-Phase Commit protocol is an implementation of transaction commit that uses a *transaction manager* (TM) process to coordinate the decision-making procedure. The RMs have the same states in this protocol as in the specification of transaction commit. The TM has the following states: *init* (its initial state), *preparing*, *committed*, and *aborted*.

The Two-Phase Commit protocol starts when an RM enters the *prepared* state and sends a *Prepared* message to the TM. Upon receipt of the *Prepared* message, the TM enters the *preparing* state and sends a *Prepare* message to every other RM. Upon receipt of the *Prepare* message, an RM that is still in the *working* state can enter the *prepared* state and send a *Prepared* message to the TM. When it has received a *Prepared* message from all RMs, the TM can enter the *committed* state and send *Commit* messages to all the other processes. The RMs can enter the *committed* state upon receipt of the *Commit* message from the TM. The message flow for the Two-Phase Commit protocol is shown in Figure 2.

Figure 2 shows one distinguished RM spontaneously preparing. In fact, any RM can spontaneously go from the *working* to *prepared* state and send a *prepared* message at any time. The TM's *prepare* message can be viewed as an optional suggestion that now would be a good time to do so. Other events, including real-time deadlines, might cause working RMs to prepare. This observation is the basis for variants of the Two-Phase Commit protocol that use fewer messages.

An RM can spontaneously enter the *aborted* state if it is in the *working* state; and the TM can spontaneously enter the *aborted* state unless it is in the *committed* state. When the TM aborts, it sends an *abort* message to all

RMs. Upon receipt of such a message, an RM enters the *aborted* state. In an implementation, spontaneous aborting can be triggered by a timeout.[2]

Two-Phase Commit is described in many texts; we will not bother to provide a more precise informal description or prove its correctness. The protocol is specified formally in Section A.2 of the Appendix, along with a theorem asserting that it implements the specification of transaction commit. This theorem has been checked by the TLC model checker for large enough configurations (numbers of RMs) so it is unlikely to be incorrect.

3.2 The Problem with Two-Phase Commit

In a transaction commit protocol, if one or more RMs fail, the transaction is usually aborted. For example, in the Two-Phase Commit protocol, if the TM does not receive a *Prepared* message from some RM soon enough after sending the *Prepare* message, then it will abort the transaction by sending *Abort* messages to the other RMs. However, the failure of the TM can cause the protocol to block until the TM is repaired. In particular, if the TM fails after every RM has sent a *Prepared* message, then the other RMs have no way of knowing whether the TM committed or aborted the transaction.

A non-blocking commit protocol is one in which the failure of a single process does not prevent the other processes from deciding if the transaction is committed or aborted. Several such protocols have been proposed, and a few have been implemented. They have usually attempted to "fix" the Two-Phase Commit protocol by choosing another TM if the first TM fails. However, we know of none that provides a complete algorithm proven to satisfy a clearly stated correctness condition. For example, the discussion of non-blocking commit in the classic text of Bernstein, Hadzilacos, and Goodman fails to explain what a process should do if it receives messages from two different processes, both claiming to be the current TM. Guaranteeing that this situation cannot arise is a problem that is as difficult as implementing a transaction commit protocol.

4 Paxos Commit

4.1 The Paxos Consensus Algorithm

The distributed computing community has studied the more general problem of *consensus*, which requires that a collection of processes agree on some

[2] In practice, an RM may notify the TM when it spontaneously aborts; we ignore this optimization.

value. Many solutions to this problem have been proposed, under various failure assumptions. These algorithms have precise fault models and rigorous proofs of correctness.

In the consensus problem, a collection of processes called *acceptors* cooperate to choose a value. The basic safety requirement is that only a single value be chosen. To rule out trivial solutions, there is an additional requirement that the chosen value must be one proposed by a client. The liveness requirement asserts that, if a large enough subnetwork of the acceptors' nodes is nonfaulty for a long enough time, then some value is eventually chosen.

The Paxos algorithm is a popular asynchronous consensus algorithm. It assumes some method of choosing a coordinator process, called the *leader*. Clients send proposed values to the leader. In normal operation, there is a unique leader. A new leader is selected only when the current one fails. However, a unique leader is needed only to ensure progress; safety is guaranteed even if there is no leader or there are multiple leaders. The Paxos algorithm satisfies the liveness requirement for consensus if the leader-selection algorithm ensures that a unique nonfaulty leader is chosen whenever a large enough subnetwork of the acceptors' nodes is nonfaulty for a long enough time.

The algorithm uses a set of ballot numbers, which we take to be nonnegative integers. Each ballot number "belongs to" a unique possible leader.

There is a predetermined choice of initial leader. In the normal, failure-free case, when the leader receives a proposed value, it sends a phase 2a message to all acceptors containing this value and ballot number 0. (The missing phase 1 is explained below.) Each acceptor receives this message, and replies with a phase 2b message for ballot number 0. When the leader receives these phase 2b messages from a majority of acceptors, it sends a phase 3 message announcing that the value is chosen.

The initial leader may not succeed in getting a value chosen in ballot 0, perhaps because it fails. In that case, one or more additional ballots are executed. The following is the general algorithm for executing a ballot; it is performed whenever a new leader is selected.

Phase 1a The leader chooses a ballot number *bal* that belongs to it and that it thinks is larger than any ballot number for which phase 1 has been performed. The leader sends a phase 1a message for ballot number *bal* to every acceptor.

Phase 1b When an acceptor receives the phase 1a message for ballot number *bal*, if it has not already performed any action for a bal-

lot numbered *bal* or higher, it responds with a phase 1b message containing its current state, which consists of

- The largest ballot number for which it received a phase 1a message, and
- The phase 2b message with the highest ballot number it has sent, if any.

Phase 2a When the leader has received a phase 1b message for ballot number *bal* from a majority of the acceptors, it can learn one of two possibilities:

Free The algorithm has not yet chosen a value.

Forced The algorithm might already have chosen a particular value v.

In the free case, the leader can try to get any value accepted; it usually picks the first value proposed by a client. In the forced case, it must try to get the value v chosen. It tries to get a value chosen by sending a phase 2a message with that value and with ballot number *bal* to every acceptor.

Phase 2b When an acceptor receives a phase 2a message for a value v and ballot number *bal*, if it has not already received a phase 1a or 2a message for a larger ballot number, it *accepts* that message and sends a phase 2b message for v and *bal* to the leader.

Phase 3 When the leader has received phase 2b messages for value v and ballot *bal* from a majority of the acceptors, it knows that the value v has been chosen and communicates that fact to all interested processes with a phase 3 message.

In the normal fault-free case, described above, the algorithm starts with the initial leader having already performed phase 1 for ballot number 0. Since there are no ballot numbers less than 0, acceptors cannot have done anything for such ballot numbers and hence have nothing to report in phase 1 for ballot number 0.

A complete specification of the Paxos algorithm must describe how a leader interprets phase 1b messages to determine if, and for what value, it is in the forced state. The reader can find such a description in the literature. It also appears in the definition of the $Phase2a$ action in the formal specification of our Paxos Commit algorithm that appears in Section A.3 of the Appendix.

The Paxos algorithm guarantees that at most one value is chosen despite any non-malicious failure of any part of the system—that is, as long

as processes do not make errors in executing the algorithm and the communication network does not undetectably corrupt messages. It guarantees progress if a unique leader is selected and if the network of nodes executing that leader and some majority of acceptors is nonfaulty for a long enough period of time.

In practice, it is not difficult to construct an algorithm that, except during rare periods of network instability, selects a suitable unique leader among a majority of nonfaulty acceptors. Transient failure of the leader-selection algorithm is harmless, violating neither safety nor eventual progress.

4.2 The Paxos Commit Algorithm

In the Two-Phase Commit protocol, the TM decides whether to abort or commit, records that decision in stable storage, and informs the RMs of its decision. We could make that fault-tolerant by simply using a consensus algorithm to choose the *committed*/*aborted* decision. However, in the normal case, the leader must learn that each RM has prepared before it can try to get the value *committed* chosen. Having the RMs tell the leader that they have prepared requires at least one message delay. Our *Paxos Commit* algorithm eliminates that message delay as follows.

Paxos Commit uses a separate instance of the Paxos consensus algorithm to obtain agreement on the decision each RM makes of whether to prepare or abort—a decision we represent by the values *Prepared* and *Aborted*. So, there is one instance of Paxos for each RM. The transaction is committed iff each RM's instance of the consensus algorithm chooses *Prepared*; otherwise the transaction is aborted.

The same set of acceptors and the same leader is used for each instance of Paxos. We assume that the RMs know the acceptors in advance. In ordinary Paxos, a ballot 0 phase 2a message can have any value v. While the leader usually sends such a message, the Paxos algorithm works just as well if, instead of the leader, some other single process sends that message. In Paxos Commit, each RM announces its prepare/abort decision by sending, in its instance of Paxos, a ballot 0 phase 2a message with the value *Prepared* or *Aborted*.

Execution of Paxos Commit normally starts when some RM decides to prepare and sends a *BeginCommit* message to the leader. The leader then sends a *Prepare* message to all the other RMs. If an RM decides that it wants to prepare, it sends a phase 2a message with value *Prepared* and ballot number 0 in its instance of the Paxos algorithm. Otherwise, it sends a phase 2a message with the value *Aborted* and ballot number 0. For each

instance, an acceptor sends its phase 2b message to the leader. The leader knows the outcome of this instance if it receives phase 2b messages for ballot number 0 from a majority of the acceptors, whereupon it can send its phase 3 message announcing the outcome to the RMs. The transaction is committed iff every RM's instance of the Paxos algorithm chooses *Prepared*; otherwise the transaction is aborted.

The instances of the Paxos algorithm for one or more RMs may not reach a decision with ballot number 0. In that case, the leader (alerted by a timeout) assumes that each of those RMs has failed and executes phase 1a for a larger ballot number in each of their instances of Paxos. If, in phase 2a, the leader learns that its choice is free (so that instance of Paxos has not yet chosen a value), then it tries to get *Aborted* chosen in phase 2b.

An examination of the Paxos algorithm—in particular, of how the decision is reached in phase 2a—shows that the value *Prepared* can be chosen in the instance for resource manager *rm* only if *rm* sends a phase 2a message for ballot number 0 with value *Prepared*. If *rm* instead sends a phase 2a message for ballot 0 with value *Aborted*, then its instance of the Paxos algorithm can choose only *Aborted*, which implies that the transaction must be aborted. In this case, Paxos Commit can short-circuit and inform all processes that the transaction has aborted. This short-circuiting is possible only for phase 2a messages with ballot number 0. It is possible for an instance of the Paxos algorithm to choose the value *Prepared* even though a leader has sent a phase 2a message (for a ballot number greater than 0) with value *Aborted*.

The safety part of the algorithm—that is, the algorithm with no progress requirements—is specified formally in Section A.3 of the Appendix, along with a theorem asserting that it implements transaction commit. The correctness of this theorem has been checked by the TLC model checker on configurations that are too small to detect subtle errors, but are probably large enough to find simple "coding" errors. Rigorous proofs of the Paxos algorithm convince us that it harbors no subtle errors, and correctness of the Paxos Commit algorithm seems to be a simple corollary of the correctness of Paxos.

A The TLA$^+$ Specifications

A.1 The Specification of a Transaction Commit Protocol

MODULE *TCommit*

CONSTANT RM The set of participating resource managers
VARIABLE $rmState$ $rmState[rm]$ is the state of resource manager rm.

$TCTypeOK \triangleq$

The type-correctness invariant

$rmState \in [RM \to \{\text{“working”}, \text{“prepared”}, \text{“committed”}, \text{“aborted”}\}]$

$TCInit \triangleq \quad rmState = [rm \in RM \mapsto \text{“working”}]$

The initial predicate.

$canCommit \triangleq \forall\, rm \in RM : rmState[rm] \in \{\text{“prepared”}, \text{“committed”}\}$

True iff all RMs are in the “prepared” or “committed” state.

$notCommitted \triangleq \forall\, rm \in RM : rmState[rm] \neq \text{“committed”}$

True iff no resource manager has decided to commit.

We now define the actions that may be performed by the RMs, and then define the complete next-state action of the specification to be the disjunction of the possible RM actions.

$$\begin{aligned} Prepare(rm) \triangleq\ & \land\ rmState[rm] = \text{“working”} \\ & \land\ rmState' = [rmState \text{ EXCEPT } ![rm] = \text{“prepared”}] \end{aligned}$$

$$\begin{aligned} Decide(rm) \triangleq\ & \lor\ \land\ rmState[rm] = \text{“prepared”} \\ & \quad\ \land\ canCommit \\ & \quad\ \land\ rmState' = [rmState \text{ EXCEPT } ![rm] = \text{“committed”}] \\ & \lor\ \land\ rmState[rm] \in \{\text{“working”}, \text{“prepared”}\} \\ & \quad\ \land\ notCommitted \\ & \quad\ \land\ rmState' = [rmState \text{ EXCEPT } ![rm] = \text{“aborted”}] \end{aligned}$$

$TCNext \triangleq \exists\, rm \in RM : Prepare(rm) \lor Decide(rm)$

The next-state action.

$TCSpec \triangleq TCInit \land \Box[TCNext]_{rmState}$

The complete specification of the protocol.

We now assert invariance properties of the specification.

$TCConsistent \triangleq$

A state predicate asserting that two RMs have not arrived at conflicting decisions.

$$\forall\, rm1,\, rm2 \in RM : \neg \wedge\, rmState[rm1] = \text{“aborted”} \\ \wedge\, rmState[rm2] = \text{“committed”}$$

THEOREM $TCSpec \Rightarrow \Box(\, TCTypeOK \wedge TCConsistent)$

Asserts that *TCTypeOK* and *TCInvariant* are invariants of the protocol.

A.2 The Specification of the Two-Phase Commit Protocol

MODULE *TwoPhase*

This specification describes the Two-Phase Commit protocol, in which a transaction manager (TM) coordinates the resource managers (RMs) to implement the Transaction Commit specification of module *TCommit*. In this specification, RMs spontaneously issue *Prepared* messages. We ignore the *Prepare* messages that the TM can send to the RMs.

For simplicity, we also eliminate *Abort* messages sent by an RM when it decides to abort. Such a message would cause the TM to abort the transaction, an event represented here by the TM spontaneously deciding to abort.

This specification describes only the safety properties of the protocol–that is, what is allowed to happen. What must happen would be described by liveness properties, which we do not specify.

CONSTANT RM The set of resource managers

VARIABLES

$rmState$,	$rmState[rm]$ is the state of resource manager RM.
$tmState$,	The state of the transaction manager.
$tmPrepared$,	The set of RMs from which the TM has received "*Prepared*" messages.

msgs

In the protocol, processes communicate with one another by sending messages. Since we are specifying only safety, a process is not required to receive a message, so there is no need to model message loss. (There's no difference between a process not being able to receive a message because the message was lost and a process simply ignoring the message.) We therefore represent message passing with a variable *msgs* whose value is the set of all messages that have been sent. Messages are never removed from *msgs*. An action that, in an implementation, would be enabled by the receipt of a certain message is here enabled by the existence of that message in *msgs*. (Receipt of the same message twice is therefore allowed; but in this particular protocol, receiving a message for the second time has no effect.)

$Message \triangleq$

The set of all possible messages. Messages of type "Prepared" are sent from the RM indicated by the message's rm field to the TM. Messages of type "Commit" and "Abort" are broadcast by the TM, to be received by all RMs. The set $msgs$ contains just a single copy of such a message.

$[type : \{\text{"Prepared"}\},\ rm : RM] \ \cup \ [type : \{\text{"Commit"},\ \text{"Abort"}\}]$

$TPTypeOK \ \triangleq$

The type-correctness invariant

$\land\ rmState \in [RM \to \{\text{"working"},\ \text{"prepared"},\ \text{"committed"},\ \text{"aborted"}\}]$
$\land\ tmState \in \{\text{"init"},\ \text{"committed"},\ \text{"aborted"}\}$
$\land\ tmPrepared \subseteq RM$
$\land\ msgs \subseteq Message$

$TPInit \ \triangleq$

The initial predicate.

$\land\ rmState = [rm \in RM \mapsto \text{"working"}]$
$\land\ tmState = \text{"init"}$
$\land\ tmPrepared \ = \{\}$
$\land\ msgs = \{\}$

We now define the actions that may be performed by the processes, first the TM's actions, then the RMs' actions.

$TMRcvPrepared(rm) \ \triangleq$

The TM receives a "Prepared" message from resource manager rm.

$\land\ tmState = \text{"init"}$
$\land\ [type \mapsto \text{"Prepared"},\ rm \mapsto rm] \in msgs$
$\land\ tmPrepared' = tmPrepared \cup \{rm\}$
$\land$ UNCHANGED $\langle rmState,\ tmState,\ msgs\rangle$

$TMCommit \ \triangleq$

The TM commits the transaction; enabled iff the TM is in its initial state and every RM has sent a "Prepared" message.

$\land\ tmState = \text{"init"}$
$\land\ tmPrepared = RM$
$\land\ tmState' = \text{"committed"}$
$\land\ msgs' = msgs \cup \{[type \mapsto \text{"Commit"}]\}$
$\land$ UNCHANGED $\langle rmState,\ tmPrepared\rangle$

$TMAbort \ \triangleq$

The TM spontaneously aborts the transaction.

$\land\ tmState = \text{"init"}$

$\land\ tmState' = \text{“aborted”}$
$\land\ msgs' = msgs \cup \{[type \mapsto \text{“Abort”}]\}$
$\land$ UNCHANGED $\langle rmState,\ tmPrepared\rangle$

$RMPrepare(rm)\ \triangleq$

Resource manager rm prepares.

$\land\ rmState[rm] = \text{“working”}$
$\land\ rmState' = [rmState \text{ EXCEPT } ![rm] = \text{“prepared”}]$
$\land\ msgs' = msgs \cup \{[type \mapsto \text{“Prepared”},\ rm \mapsto rm]\}$
$\land$ UNCHANGED $\langle tmState,\ tmPrepared\rangle$

$RMChooseToAbort(rm)\ \triangleq$

Resource manager rm spontaneously decides to abort. As noted above, rm does not send any message in our simplified spec.

$\land\ rmState[rm] = \text{“working”}$
$\land\ rmState' = [rmState \text{ EXCEPT } ![rm] = \text{“aborted”}]$
$\land$ UNCHANGED $\langle tmState,\ tmPrepared,\ msgs\rangle$

$RMRcvCommitMsg(rm)\ \triangleq$

Resource manager rm is told by the TM to commit.

$\land\ [type \mapsto \text{“Commit”}] \in msgs$
$\land\ rmState' = [rmState \text{ EXCEPT } ![rm] = \text{“committed”}]$
$\land$ UNCHANGED $\langle tmState,\ tmPrepared,\ msgs\rangle$

$RMRcvAbortMsg(rm)\ \triangleq$

Resource manager rm is told by the TM to abort.

$\land\ [type \mapsto \text{“Abort”}] \in msgs$
$\land\ rmState' = [rmState \text{ EXCEPT } ![rm] = \text{“aborted”}]$
$\land$ UNCHANGED $\langle tmState,\ tmPrepared,\ msgs\rangle$

$TPNext\ \triangleq$
$\lor\ TMCommit \lor TMAbort$
$\lor \exists\, rm \in RM :$
$\quad TMRcvPrepared(rm) \lor RMPrepare(rm) \lor RMChooseToAbort(rm)$
$\quad\ \lor RMRcvCommitMsg(rm) \lor RMRcvAbortMsg(rm)$

$TPSpec\ \triangleq\ TPInit \land \Box[TPNext]_{\langle rmState,\, tmState,\, tmPrepared,\, msgs\rangle}$

The complete spec of the Two-Phase Commit protocol.

THEOREM $TPSpec \Rightarrow \Box\, TPTypeOK$

This theorem asserts that the type-correctness predicate TPTypeOK is an invariant of the specification.

We now assert that the Two-Phase Commit protocol implements the Transaction Commit protocol of module *TCommit*. The following statement defines $TC\,!\,TCSpec$ to be formula *TSpec* of module *TCommit*. (The TLA$^+$ INSTANCE statement is used to rename the operators defined in module *TCommit* avoids any name conflicts that might exist with operators in the current module.)

$TC \triangleq$ INSTANCE *TCommit*

THEOREM $TPSpec \Rightarrow TC\,!\,TCSpec$

This theorem asserts that the specification TPSpec of the Two-Phase Commit protocol implements the specification TCSpec of the Transaction Commit protocol.

The two theorems in this module have been checked with TLC for six RMs, a configuration with 50816 reachable states, in a little over a minute on a 1 GHz PC.

A.3 The Paxos Commit Algorithm

MODULE *PaxosCommit*

This module specifies the Paxos Commit algorithm. We specify only safety properties, not liveness properties. We simplify the specification in the following ways.

- As in the specification of module *TwoPhase*, and for the same reasons, we let the variable *msgs* be the set of all messages that have ever been sent. If a message is sent to a set of recipients, only one copy of the message appears in *msgs*.
- We do not explicitly model the receipt of messages. If an operation can be performed when a process has received a certain set of messages, then the operation is represented by an action that is enabled when those messages are in the set *msgs* of sent messages. (We are specifying only safety properties, which assert what events can occur, and the operation can occur if the messages that enable it have been sent.)
- We do not model leader selection. We define actions that the current leader may perform, but do not specify who performs them.

As in the specification of Two-Phase commit in module *TwoPhase*, we have RMs spontaneously issue Prepared messages and we ignore *Prepare* messages.

EXTENDS *Integers*

$Maximum(S) \triangleq$

If S is a set of numbers, then this define $Maximum(S)$ to be the maximum of those numbers, or -1 if S is empty.

IF $S = \{\}$ THEN -1
ELSE CHOOSE $n \in S : \forall\, m \in S : n \geq m$

CONSTANT RM, The set of resource managers.
$Acceptor$, The set of acceptors.

Majority, The set of majorities of acceptors
Ballot The set of ballot numbers

ASSUME We assume these properties of the declared constants.

$\land\ Ballot \subseteq Nat$
$\land\ 0 \in Ballot$
$\land\ Majority \subseteq \text{SUBSET}\ Acceptor$
$\land\ \forall\, MS1,\ MS2 \in Majority : MS1 \cap MS2 \neq \{\}$

All we assume about the set *Majority* of majorities is that any two majorities have non-empty intersection.

$Message \triangleq$

The set of all possible messages. There are messages of type "Commit" and "Abort" to announce the decision, as well as messages for each phase of each instance of *ins* of the Paxos consensus algorithm. The *acc* field indicates the sender of a message from an acceptor to the leader; messages from a leader are broadcast to all acceptors.

$[type : \{\text{"phase1a"}\},\ ins : RM,\ bal : Ballot \setminus \{0\}]$
$\cup$
$[type : \{\text{"phase1b"}\},\ ins : RM,\ mbal : Ballot,\ bal : Ballot \cup \{-1\},$
$\ val : \{\text{"prepared"},\ \text{"aborted"},\ \text{"none"}\},\ acc : Acceptor]$
$\cup$
$[type : \{\text{"phase2a"}\},\ ins : RM,\ bal : Ballot,\ val : \{\text{"prepared"},\ \text{"aborted"}\}]$
$\cup$
$[type : \{\text{"phase2b"}\},\ acc : Acceptor,\ ins : RM,\ bal : Ballot,$
$\ val : \{\text{"prepared"},\ \text{"aborted"}\}]$
$\cup$
$[type : \{\text{"Commit"},\ \text{"Abort"}\}]$

VARIABLES

rmState, *rmState*[*rm*] is the state of resource manager *rm*.
aState, *aState*[*ins*][*ac*] is the state of acceptor *ac* for instance *ins* of the Paxos algorithm
msgs The set of all messages ever sent.

$PCTypeOK \triangleq$

The type-correctness invariant. Each acceptor maintains the values *mbal*, *bal*, and *val* for each instance of the Paxos consensus algorithm.

$\land\ rmState \in [RM \to \{\text{"working"},\ \text{"prepared"},\ \text{"committed"},\ \text{"aborted"}\}]$
$\land\ aState \in [RM \to [Acceptor \to$
$[mbal : Ballot,$
$\ bal : Ballot \cup \{-1\},$
$\ val : \{\text{"prepared"},\ \text{"aborted"},\ \text{"none"}\}]]]$

$\land\ msgs \in \text{SUBSET } Message$

$PCInit \triangleq$ The initial predicate.

$\land\ rmState = [rm \in RM \mapsto \text{“working”}]$

$\land\ aState = [ins \in RM \mapsto [ac \in Acceptor \mapsto [mbal \mapsto 0,\ bal \mapsto -1,\ val \mapsto \text{“none”}]]]$

$\land\ msgs = \{\}$

The Actions

$Send(m) \triangleq msgs' = msgs \cup \{m\}$

An action expression that describes the sending of message m.

RM Actions

$RMPrepare(rm) \triangleq$

Resource manager rm prepares by sending a phase 2a message for ballot number 0 with value “prepared”.

$\land\ rmState[rm] = \text{“working”}$

$\land\ rmState' = [rmState \text{ EXCEPT } ![rm] = \text{“prepared”}]$

$\land\ Send([type \mapsto \text{“phase2a”},\ ins \mapsto rm,\ bal \mapsto 0,\ val \mapsto \text{“prepared”}])$

$\land\ \text{UNCHANGED } aState$

$RMChooseToAbort(rm) \triangleq$

Resource manager rm spontaneously decides to abort. It may (but need not) send a phase 2a message for ballot number 0 with value “aborted”.

$\land\ rmState[rm] = \text{“working”}$

$\land\ rmState' = [rmState \text{ EXCEPT } ![rm] = \text{“aborted”}]$

$\land\ Send([type \mapsto \text{“phase2a”},\ ins \mapsto rm,\ bal \mapsto 0,\ val \mapsto \text{“aborted”}])$

$\land\ \text{UNCHANGED } aState$

$RMRcvCommitMsg(rm) \triangleq$

Resource manager rm is told by the leader to commit. When this action is enabled, $rmState[rm]$ must equal either “prepared” or “committed”. In the latter case, the action leaves the state unchanged (it is a “stuttering step”).

$\land\ [type \mapsto \text{“Commit”}] \in msgs$

$\land\ rmState' = [rmState \text{ EXCEPT } ![rm] = \text{“committed”}]$

$\land\ \text{UNCHANGED } \langle aState,\ msgs \rangle$

$RMRcvAbortMsg(rm) \triangleq$

Resource manager rm is told by the leader to abort. It could be in any state except “committed”.

$\land\, [type \mapsto \text{“Abort”}] \in msgs$
$\land\, rmState' = [rmState \text{ EXCEPT } ![rm] = \text{“aborted”}]$
$\land$ UNCHANGED $\langle aState,\, msgs\rangle$

Leader Actions

The following actions are performed by any process that believes itself to be the current leader. Since leader selection is not assumed to be reliable, multiple processes could simultaneously consider themselves to be the leader.

$Phase1a(bal,\, rm) \triangleq$

If the leader times out without learning that a decision has been reached on resource manager *rm*'s prepare/abort decision, it can perform this action to initiate a new ballot *bal*. (Sending duplicate phase 1a messages is harmless.)

$\land\, Send([type \mapsto \text{“phase1a”},\, ins \mapsto rm,\, bal \mapsto bal])$
$\land$ UNCHANGED $\langle rmState,\, aState\rangle$

$Phase2a(bal,\, rm) \triangleq$

The action in which a leader sends a phase 2a message with ballot $bal > 0$ in instance *rm*, if it has received phase 1b messages for ballot number *bal* from a majority of acceptors. If the leader received a phase 1b message from some acceptor that had sent a phase 2b message for this instance, then $maxbal \geq 0$ and the value *val* the leader sends is determined by the phase 1b messages. (If $val =$ “prepared”, then *rm* must have prepared.) Otherwise, $maxbal = -1$ and the leader sends the value “aborted”.

The first conjunct asserts that the action is disabled if any commit leader has already sent a phase 2a message with ballot number *bal*. In practice, this is implemented by having ballot numbers partitioned among potential leaders, and having a leader record in stable storage the largest ballot number for which it sent a phase 2a message.

$\land\, \neg\exists\, m \in msgs :\ \land\, m.type = \text{“phase2a”}$
$\qquad\qquad\qquad\qquad\ \land\, m.bal = bal$
$\qquad\qquad\qquad\qquad\ \land\, m.ins = rm$
$\land\, \exists\, MS \in Majority :$
$\quad$ LET $mset \triangleq \{m \in msgs :\ \land\, m.type = \text{“phase1b”}$
$\qquad\qquad\qquad\qquad\qquad\qquad\ \land\, m.ins = rm$
$\qquad\qquad\qquad\qquad\qquad\qquad\ \land\, m.mbal = bal$
$\qquad\qquad\qquad\qquad\qquad\qquad\ \land\, m.acc \in MS\}$
$\qquad maxbal \triangleq Maximum(\{m.bal : m \in mset\})$
$\qquad val \triangleq$ IF $maxbal = -1$
$\qquad\qquad\qquad$ THEN “aborted”
$\qquad\qquad\qquad$ ELSE $(\text{CHOOSE } m \in mset : m.bal = maxbal).val$
$\quad$ IN $\land\, \forall\, ac \in MS : \exists\, m \in mset : m.acc = ac$
$\qquad\ \land\, Send([type \mapsto \text{“phase2a”},\, ins \mapsto rm,\, bal \mapsto bal,\, val \mapsto val])$
$\land$ UNCHANGED $\langle rmState,\, aState\rangle$

$Decide \triangleq$

A leader can decide that Paxos Commit has reached a result and send a message announcing the result if it has received the necessary phase 2b messages.

$\land$ LET $Decided(rm,\ v) \triangleq$

True iff instance rm of the Paxos consensus algorithm has chosen the value v.

$\exists\, b \in Ballot,\ MS \in Majority :$
$\forall\, ac \in MS : [type \mapsto$ "phase2b", $ins \mapsto rm,$
$bal \mapsto b,\ val \mapsto v,\ acc \mapsto ac] \in msgs$

IN $\lor\ \land \forall\, rm \in RM : Decided(rm,$ "prepared")
$\land\ Send([type \mapsto$ "Commit"])
$\lor\ \land \exists\, rm \in RM : Decided(rm,$ "aborted")
$\land\ Send([type \mapsto$ "Abort"])

$\land$ UNCHANGED $\langle rmState,\ aState\rangle$

Acceptor Actions

$Phase1b(acc) \triangleq$
$\exists\, m \in msgs :$
$\land\ m.type =$ "phase1a"
$\land\ aState[m.ins][acc].mbal < m.bal$
$\land\ aState' = [aState$ EXCEPT $![m.ins][acc].mbal = m.bal]$
$\land\ Send([type \mapsto$ "phase1b",
$ins \mapsto m.ins,$
$mbal \mapsto m.bal,$
$bal \mapsto aState[m.ins][acc].bal,$
$val \mapsto aState[m.ins][acc].val,$
$acc \mapsto acc])$
$\land$ UNCHANGED $rmState$

$Phase2b(acc) \triangleq$
$\land \exists\, m \in msgs :$
$\land\ m.type =$ "phase2a"
$\land\ aState[m.ins][acc].mbal \leq m.bal$
$\land\ aState' = [aState$ EXCEPT $![m.ins][acc].mbal = m.bal,$
$![m.ins][acc].bal = m.bal,$
$![m.ins][acc].val = m.val]$
$\land\ Send([type \mapsto$ "phase2b", $ins \mapsto m.ins,\ bal \mapsto m.bal,$
$val \mapsto m.val,\ acc \mapsto acc])$
$\land$ UNCHANGED $rmState$

$PCNext \triangleq$ The next-state action

$\lor \exists\, rm \in RM :\ \lor\ RMPrepare(rm)$
$\qquad\qquad\qquad\qquad\ \lor\ RMChooseToAbort(rm)$
$\qquad\qquad\qquad\qquad\ \lor\ RMRcvCommitMsg(rm)$
$\qquad\qquad\qquad\qquad\ \lor\ RMRcvAbortMsg(rm)$

$\lor \exists\, bal \in Ballot \setminus \{0\},\ rm \in RM : Phase1a(bal,\ rm) \lor Phase2a(bal,\ rm)$

$\lor\ Decide$

$\lor \exists\, acc \in Acceptor : Phase1b(acc) \lor Phase2b(acc)$

$PCSpec \triangleq PCInit \land \Box[PCNext]_{\langle rmState,\ aState,\ msgs\rangle}$

The complete spec of the Paxos Commit protocol.

THEOREM $PCSpec \Rightarrow PCTypeOK$

We now assert that the two-phase commit protocol implements the transaction commit protocol of module TCommit. The following statement defines $TC\,!\,TCSpec$ to be the formula *TCSpec* of module *TCommit*. (The TLA$^+$ INSTANCE statement must is used to rename the operators defined in module *TCommit* to avoid possible name conflicts with operators in the current module having the same name.)

$TC \triangleq$ INSTANCE $TCommit$

THEOREM $PCSpec \Rightarrow TC\,!\,TCSpec$