A Highly Available Log Service for Distributed Transaction Processing

Lásaro Camargos[‡⋆] Marcin Wieloch[⋆] Fernando Pedone[⋆]

Edmundo Madeira[‡]

[⋆]Faculty of Informatics
University of Lugano
6900 Lugano, Switzerland

[‡]Institute of Computing
University of Campinas
3084-971 Campinas-SP, Brazil

University of Lugano
Faculty of Informatics
Technical Report No. 2006/08
December 2006

Abstract

Processing distributed transactions hinges on enforcing atomicity and durability: resource managers participating in a transaction must agree on its outcome and the transaction updates must be permanent. We introduce the log service abstraction, which gathers resource managers' votes to commit or abort a transaction and their updates, and outputs the transaction's outcome. Updates are made durable and non-concurrent transactions are totally ordered by the service. The sequence of updates performed by a resource manager is available as a means to consistently recover resource managers without relying on their local state. As a consequence, a remote process, whose state will be recovered from the log service, can reincarnate a crashed resource manager. Moreover, the service ensures that only one process plays the role of a given resource manager at any time. We present two highly available implementations of this service and evaluate their performance running TPC-C and a microbenchmark on a distributed database.

Contents

1 Introduction

1.1 Atomic Commit and Paxos Commit

The problem of atomic termination of transactions is central to any distributed database system aiming at strong consistency. Terminating a distributed transaction is normally performed by an *atomic commitment protocol*, which gathers votes from all resource managers involved in the transaction and decides on the transaction's outcome based on these votes. If all votes are for commit, the transaction is committed; otherwise it is aborted. To avoid blocking when a resource manager is not available to vote, a weaker version of atomic commitment requires the outcome to be commit if none of the participants is suspected to have failed and all vote to commit the transaction. This weaker problem is called *non-blocking atomic commitment*.

In a recent paper, Gray and Lamport have introduced Paxos Commit, a non-blocking atomic commitment protocol based on the Paxos consensus algorithm [5]. In Paxos Commit each resource manager uses a consensus instance to vote. If a resource manager suspects that another one has crashed and cannot vote, the first one votes on behalf of the second resource manager, proposing in its consensus instance to abort the transaction. If more than one vote is issued for a resource manager (e.g., it is wrongly suspected), consensus ensures that participants will agree on the vote of each participant.[1] Paxos Commit is not only an elegant but also an efficient solution to non-blocking atomic commitment. It has the same latency as two-phase commit (2PC) without blocking if the transaction coordinator fails. It is cheaper than three-phase commit (3PC) and has a simpler termination protocol in case of failure of the coordinator [2]. Instead of keeping the participant votes at the coordinator, in Paxos Commit this information is stored at the *acceptors*, the processes responsible for consensus decisions.

1.2 The Highly Available Log Service

Paxos Commit suggests that information critical to transaction termination should reside neither at the coordinator, as in 2PC, nor at the resource managers, as in 3PC, but at a third party, the acceptors, whose availability can be easily parameterized. Inspired by Paxos Commit, we abstract transaction termination in terms of a *highly available log service*. This approach is justified by the following reasons:

- *Simplicity.* A non-blocking atomic commitment protocol based on a highly available log service is quite intuitive: Transaction participants submit their votes to the remote log; as soon as there is one commit vote for each participant or one abort vote stored in the log, the outcome of the transaction can be determined. Even if more than one vote is issued for a resource manager, should it be suspected to have failed, the ordering in which the votes are kept in the log ensures that the transaction outcome is deterministic.

- *Performance.* By defining transaction termination in terms of a log service, we allow other implementations to take advantage of our approach to non-blocking atomic commitment. In fact, we present in the paper an implementation of the log service that has better performance than a Paxos Commit-like implementation. We substantiate this claim with experimental results using the TPC-C benchmark and a micro-benchmark running in a cluster of nodes.

- *Fault-tolerance.* The log can be made highly available using standard replication techniques (e.g., state machine replication). Moreover, if a resource manager fails, then another instance of it can

[1] Notice that in Paxos not all consensus participants are required to propose a value.

be created on an operational node using the state stored in the log. Thus, the recovery of the node hosting the resource manager is not needed for the system to resume operation, increasing the availability of resource managers.

1.3 From specs to implementations

We discuss in detail two implementations of the log service, *uncoordinated*, inspired by Paxos Commit, and *coordinated*. Both implementations rely on a series of consensus executions. The main difference between them concerns the ordering of these executions, and therefore the ordering of committed transactions in the log. While there is a separate sequence of consensus instances per resource manager in the uncoordinated implementation, leading to a *partial order of committed transactions in the log*, there is a single sequence of consensus in the coordinated version, ensuring a *total order of committed transactions in the log*. In the coordinated implementation, instead of using consensus to vote, resource managers send their votes to the coordinator, which submits the votes in a consensus instance. The nature of each approach has performance implications on both the termination of transactions (Section 3.3) and the recovery of resource managers (Section 3.3).

1.4 Theory and practice of termination

The coordinated implementation requires one more communication step than the uncoordinated implementation, corresponding to the vote sent by the participants to the coordinator. Notice that the coordinator here does not play the same role as the transaction coordinator in 2PC or 3PC. The role of the transaction coordinator in the coordinated log implementation is to gather all votes and propose them in a consensus instance. Just like in Paxos, if the coordinator fails, it can be promptly replaced by another process.

In theory, the coordinated log implementation should take longer to execute than the uncoordinated log implementation. However, performance experiments revealed that, contrary to the expectations, the coordinated approach largely outperforms the uncoordinated technique in a clustered environment. This is due to the fact that by having a single coordinator, votes from all participants in an atomic commitment, and also from concurrent atomic commitment executions, can be batched by the coordinator and proposed in a single consensus instance, saving not only in the number of messages but also in the processing done by the acceptors. These savings largely make up for the additional message latency of the coordinated approach.

1.5 Recovering resource managers

The ordering of transactions in the log has an impact on the recovery of resource managers. Upon recovery, a resource manager should find out the outcome of transactions in whose termination it was involved before the failure. The question is: How many consensus instances should be inspected until a resource manager can rebuild a consistent image of the local database? The answer to this question can be simple if the log implementation is coordinated: It suffices for the resource manager to send a "flush" request to the coordinator and wait until this request is decided in some instance. Once the request is decided in instance i, it follows from the total order of log entries that any pending transactions concerning the resource manager must have been decided before instance i. The coordinator may have them cached locally, or learn their values by re-executing the consensus instances for the missing values.

Recovery with an uncoordinated log implementation is more complex and depends on the recovery strategy and the maximum number k of transactions allowed to execute concurrently at the resource manager:

- If resource managers are statically assigned to nodes, and a node crashes, the resource manager will only recover after the node is operational again. In this case, the resource manager proposes "flush" values using its local sequence of consensus until the value is decided in some instance i. Once this happens, it should find out the decision of instances $i+1, i+2, ..., i+k-1$ since there could have been up to k concurrent transactions in execution.

- If a resource manager is dynamically assigned to a node and this one is suspected to have crashed, another instance of it is reincarnated on an available node. In the best case, the procedure described above may work here too, but if the original node did not crash, there could be two operational processes incarnating the same resource manager simultaneously, and they would propose values in the same consensus stream, an undesired situation. Eventually the "flush" request will be delivered, and then $k-1$ more instances should be executed.

1.6 Our contributions

Section 2 reviews the atomic commit problem and specifies a property for consistent recovery. Section 3 introduces the log service specification and shows how it can be used by resource managers. Some implementation considerations are discussed in Section 4, and the coordinated and uncoordinated log implementations are presented in Sections 5 and 6, respectively. These approaches are compared in Section 7. Section 8 reviews related work and Section 9 concludes the paper.

2 Problem statement

Transactions are executed by a collection of processes called *resource managers (RMs)*. When a transaction ends, a special process called *transaction manager (TM)* starts an atomic commit (AC) protocol, in which every RM participating in the transaction votes for its commit or abort. Processes may fail by crashing and never behave maliciously (i.e., no Byzantine failures).

Atomic commit is defined by the following properties.

- **AC-Validity** If an RM decides to commit a transaction, then all RMs voted to commit the transaction.

- **AC-Agreement** It is impossible for one RM to commit a transaction and another one to abort the transaction.

- **AC-Non-Triviality** If all RMs vote to commit the transaction and no RM is suspected throughout the execution of the protocol, then the decision is commit.

- **AC-Termination** Non-faulty RMs eventually decide.

AC-Non-Triviality assumes that RMs can be "suspected" to have failed. The only assumption that we make about failure detection is that if an RM fails, then it will eventually be suspected by the other

processes.[2] Therefore, resource managers may be incorrectly suspected to have failed, in which case transactions would be unnecessarily aborted.

Atomic commit defines how distributed transactions terminate. A related problem is building a consistent state of a recovering resource manager. Intuitively, the problems are related because after recovering from a crash, an RM should find out which transactions were committed before the failure. Determining such transactions requires identifying previous instances of atomic commit executions.

The following property characterizes correct recovery from failures. R-Consistency can be ensured, for example, by replaying the write operations of committed transactions in the order in which these transactions committed.

- **R-Consistency** The database state of an RM after the recovery from a failure is the same as its committed state before the failure.

3 The Log Service

In the following we discuss how atomic commitment and recovery of resource managers can be implemented using an abstract log service. Before explaining the abstract algorithms, some formalism must be introduced.

3.1 Terminology and notation

The symbol " $\triangleq$ " should be read as "is defined as". For example,

$$\textsc{If}(a, b, c) \triangleq \textbf{if } a \textbf{ then } b \textbf{ else } c$$

means that $\textsc{If}(a, b, c)$ evaluates to b, if a is true, and it evaluates to c, otherwise; and

$$\textsc{AddValToB}(val) \triangleq \mathcal{B} \leftarrow \mathcal{B} + val$$

increments $\mathcal{B}$ by the value val. **let** and **in** specify the scope of a definition. For example,

$$\textbf{let } \textsc{Sum}(a, b) \triangleq a + b \textbf{ in } \textsc{Sum}(\textsc{Sum}(1, 2), \textsc{Sum}(3, 4))$$

defines $\textsc{Sum}(a, b)$ in the expression on the right of **in** only, evaluated to 10. For simplicity we assume that, with one exception (Algorithm 1 in Section 3.3), all definitions presented in the paper are performed atomically.

We use angle brackets to enclose a sequence of values as a tuple; $\langle\rangle$ is the tuple with no elements, and $\langle a, b, c\rangle$ is the tuple with elements a, b and c, in this order. The symbol "_" in a tuple matches any element. That is, $\langle a, b\rangle = \langle \_, b\rangle$ and $\langle \_, b\rangle = \langle c, b\rangle$ even if $c \neq a$. We denote by $\text{Len}(s)$ the number of elements of sequence s, and use the square brackets to index its elements, e.g., $\langle a, b, c\rangle[2]$ is the element b. $a <_s b$ (resp. $>_s$) holds if and only if a and b happen only once in s and $s[i] = a$ and $s[j] = b$ implies $i < j$ (resp. $i > j$). $s \bullet e$ is defined as the sequence s appended by element e (e.g., $\langle a, b\rangle \bullet c = \langle a, b, c\rangle$). If ss is a sequence of sets, then $ss \oplus e$ adds element e to the last set in ss, $ss[\text{Len}(ss)]$, (e.g., $\langle\{a, b\}, \{c\}\rangle \oplus d = \langle\{a, b\}, \{c, d\}\rangle$).

[2] This property is similar to the *completeness* property of Chandra and Toueg's failure detectors [3].

3.2 The Log Service specification

Algorithm 1 presents the abstract behavior of the log service (lines 1–40) and of transaction termination and resource manager recovery (lines 41–54, in page 6). In Sections 4–6 we show how these behaviors can be implemented in a shared-nothing asynchronous distributed system.

The algorithm uses five data-structures:

$\mathcal{V}$ The set of received votes, initially empty.

$\mathcal{T}$ A partially ordered set $(S, \preceq)$, where S is the set of committed transactions, and $\preceq$ is partial order relation over the elements of S. Non-concurrent transactions in S are totally ordered by $\preceq$, according to their commit order. For simplicity, we represent $\mathcal{T}$ as a sequence of sets of committed transactions such that, given two sets $\mathcal{T}[i]$ and $\mathcal{T}[j]$ in $\mathcal{T}$, $i < j$ implies that for all transactions t and u, $t \in \mathcal{T}[j]$ and $u \in \mathcal{T}[j]$, $t \preceq u$.

$\mathcal{C}$ **and** $LastC$ Helpers on building $\mathcal{T}$. $\mathcal{C}$ is the set of non-terminated transactions whose votes have been issued; $LastC$ is the subset of $\mathcal{C}$ with transactions concurrent to the last committed one. Both are initially empty.

$\mathcal{R}$ A mapping from resource managers to the processes that are currently "incarnating" them. Initially, all resource managers are mapped to no process (i.e., $\perp$ is an invalid process identifier).

OUTCOME(t) defines the result of transaction t. It is determined by checking whether at least one vote for t is ABORT, in which case the outcome is ABORT, or if all votes are COMMIT, in which case the outcome is COMMIT. If not all votes are known to the log service, but all the votes known so far are COMMIT, the outcome is UNDEFINED.

ISINVOLVED(t, rm) evaluates to TRUE if rm is a participant in t and to FALSE, otherwise, by checking if rm is in the participant list ($tset$) of any known vote for t. Because RMs may vote to ABORT a transaction at any time during its execution, including before learning about the other participants, ABORT votes may not contain the complete list of participants. Therefore, ISINVOLVED may not have conclusive information if all known votes for t are ABORT; in such a case the answer is UNKNOWN.

VOTE defines how votes are added to $\mathcal{V}$. A vote is added only if no previous vote for the same resource manager to the same transaction was added before. This ensures that if suspecting and suspected processes issue conflicting votes, only one vote per participant will be kept.

UPDATES(rm) evaluates to the sequence of sets of updates performed by resource manager rm, partially ordered accordingly to $\mathcal{T}$.

3.3 Termination and Recovery

Resource managers use the log service to terminate transactions and recover after crashes, as defined in Algorithm 1 (lines 41–54, in page 7). INCARNATE is used by process pid to incarnate resource manager rm, be it the first process to do it or a replacement incarnation. Firstly, the process sets $\mathcal{R}[rm]$ to pid and then evaluates UPDATES, described above, to get the updates executed by the previous rm incarnation. $updates$ is then scanned in order from the first to the last set, and all updates in one set are applied to the database before all the ones in the next set; no order among elements in the same set is required.

At the end of INCARNATE, if $\mathcal{R}[rm] = pid$, then pid has successfully incarnated rm and recovered the previous incarnation's state. pid will accept and process new transactions until it crashes or another process incarnates rm (i.e., $pid \neq \mathcal{R}[rm]$). If more than one process try to incarnate rm, only the last one to execute INCARNATE will succeed.

Algorithm 1 Log service specification

```
1:  Initially:
2:    V ← ∅                                          ◁ The history of votes.
3:    T ← ⟨⟩                                         ◁ Sequence of sets of committed transactions.
4:    C ← ∅                                          ◁ Set of concurrent transactions.
5:    LastC ← ∅                                      ◁ Set of concurrent transactions.
6:    ∀r ∈ RM, R[r] ← ⊥                              ◁ Processes incarnating RMs.

7:  OUTCOME(t) ≜
8:    if ∃⟨_, t, _, ABORT, _⟩ ∈ V                    ◁ Any ABORTs?
9:      ABORT
10:   else if∃⟨_, t, tset, _, _⟩ ∈ V : ∀p ∈ tset :
        ⟨p, t, tset, COMMIT, _⟩ ∈ V                  ◁ All COMMITs?
11:     COMMIT
12:   else
13:     UNDEFINED                                    ◁ Neither one nor the other

14: ISINVOLVED(t, rm) ≜
15:   if ∃⟨_, t, tset, _, _⟩ ∈ V : rm ∈ tset        ◁ Is rm in any list?
16:     TRUE
17:   else if∃⟨_, t, _, v, _⟩ ∈ V : v = COMMIT      ◁ Has t committed?
18:     FALSE
19:   else
20:     UNKNOWN                                      ◁ I don't know...

21: VOTE(⟨rm, t, tset, vote, update⟩) ≜
22:   if OUTCOME(t) = UNDEFINED                      ◁ If t has not terminated yet
23:     C ← C ∪ {t}                                  ◁ add it to C.
24:   if ¬∃⟨rm, t, _, _, _⟩ ∈ V                      ◁ Has rm voted yet?
25:     prevState ← OUTCOME(t)                       ◁ ABORT or UNDEFINED.
26:     V ← V ∪ {⟨rm, t, tset, vote, update⟩}        ◁ Keep this vote
27:     if (prevState = UNDEFINED) ∧ (OUTCOME(t) = COMMIT)
28:       if t ∈ LastC                               ◁ If t can be added to the last set...
29:         T ← T ⊕ t                                ◁ ...do it...
30:       else
31:         T ← T • {t}                              ◁ ...else add it to a new set
32:       LastC ← C                                  ◁ With a new LastC.
33:     C ← C\{t}

34: UPDATES(rm) ≜                                    ◁ Set incarnating process.
35:   let UPD(i) ≜
36:     if i = 0
37:       ⟨⟩
38:     else
39:       UPD(i − 1) • {upd : ⟨rm, t, _, COMMIT, upd⟩ ∈ V :
                                          t ∈ T[i] ∧ ISINVOLVED(rm, t)}
40:   in UPD(Len(T))                                 ◁ Return updates for rm
```

A resource manager rm uses TERMINATE to vote at the termination of transaction t, of which it is a participant. If rm is willing to commit the transaction, then $vote$ equals COMMIT and upd contains the updates performed by t. To abort the transaction $vote$ equals ABORT and upd is the empty set. After casting its vote, rm waits until it learns t's outcome. While waiting, rm monitors the other resource managers in $tset$, also involved in t. If rm suspects that some participant crashed, it votes ABORT on its behalf. After learning that t committed, rm will apply its updates (and possibly release the related locks, depending on the database). If rm learns that t aborted, it locally aborts the transaction. Updates are

Algorithm 1 (cont'd) Resource manager specification

```
41: INCARNATE(rm, pid) ≜
42:   R[rm] ← pid
43:   updates ← UPDATES(rm)                          ◁ Get committed state.
44:   for i = 1 to Len(updates)                      ◁ For each set of committed transaction...
45:     apply updates in updates[i]                  ◁ ...apply it to the database.

46: TERMINATE(rm, t, tset, vote, upd) ≜
47:   VOTE(⟨rm, t, tset, vote, upd⟩)
48:   while OUTCOME(t, rm) = UNDEFINED
49:     wait (OUTCOME(t, rm) ≠ UNDEFINED) ∨ (suspect r ∈ tset)
50:     if suspected r ∈ tset
51:       VOTE(⟨r, t, tset, Abort, ∅⟩)
52:   if OUTCOME(t, rm) = ABORT
53:     abort t in the database
54:   else
55:     apply upd to database
```

made durable by the log service. We assume that the TERMINATE definition is not atomic, so multiple resource managers can vote in parallel, as well as the same resource manager can terminate distinct transactions in parallel, if its scheduling model allows it.

3.4 Correctness

Algorithm 1 solves non-blocking atomic commitment. AC-Validity and AC-Agreement are ensured by having the votes added to $\mathcal{V}$, from which OUTCOME deterministically evaluates to the transaction's outcome. If no resource manager is suspected and all vote COMMIT for a given transaction, then only COMMIT votes will be added to $\mathcal{V}$, and COMMIT is the only possible outcome for such a transaction, satisfying the AC-Non-Triviality property. Finally, since crashed resource managers are eventually suspected and have votes issued on their behalf, the protocol eventually terminates, satisfying the AC-Termination property.

The state of resource managers is changed by applying updates from write transactions. If these updates are deterministic, then given two copies of a resource manager in the same initial state and applying the same sequence of updates leads to the same final state. Assuming that transactions terminating concurrently do not interfere with each other and that update operations are deterministic, the partial order on updates provided by the log service and the deterministic method to apply them when reincarnating a resource manager, specified in Algorithm 1, correctly recovers the state before the crash and satisfies the R-Consistency property.

Formal proofs that the log service provides these properties as well as that it is implemented by protocols in Section 5 and 6 are given at Section 9.

4 From abstract specifications to distributed implementations

In this section we define the system model and some building blocks used in our implementations.

4.1 Processes, communications and failures

We assume a shared-nothing asynchronous distributed system composed of a set $\{p_1, p_2, ..., p_n\}$ of processes. Processes may fail by crashing. Communication is by message passing using the primitives *send* and *receive*. These primitives guarantee quasi-reliable communication, that is, they ensure that if neither the sender nor the receiver of a message crashes, then the message is eventually delivered. This abstraction can be implemented on top of fair-lossy channels, for example, by periodically resubmitting messages until an acknowledgment is received by the sender. Quasi-reliable channels also ensure that messages are neither corrupted nor duplicated.

4.2 Leader-election oracle

Our coordinated implementation of the log service assumes a leader-election oracle. Participants use this oracle to determine the current coordinator. The leader-election oracle guarantees that eventually all participants will elect the same non-faulty process as the leader. Obviously, this can only be ensured if there is at least one process that eventually remains operational "forever"—in practical terms, long enough so that some useful computation can be done (e.g., deciding on a transaction's outcome).

4.3 Consensus

In the consensus problem, a set of processes tries to agree on a common value. As in [9], we define the consensus problem over three process roles: *proposers* can propose a value (i.e., the request to be executed by the service), *acceptors* interact to choose a proposed value, and *learners* must learn the decided value. A process can play any number of these roles, and it is dubbed non-faulty if it remains up long enough to perform the consensus algorithm. Let n be the number of acceptors in the system and $f < n/2$ the maximum number of faulty processes. A correct consensus algorithm satisfies three properties:

C-Nontriviality Only a proposed value may be learned.

C-Consistency Any two learned values must be equal.

C-Progress For any proposer p and learner l, if p, l and more than $n/2$ acceptors are non-faulty, and p proposes a value, then l must learn a value.

The algorithms in Sections 5 and 6 use many instance of consensus. Therein, the primitive *propose(*i, v*)* is used to propose a value v in the consensus instance i. The decision of instance i is the v in *decide(*i, v*)*.

5 Coordinated Implementation

The coordinated implementation is named after the coordinator, a process that serves as the log service's interface to resource managers. The coordinator receives the votes from resource managers, makes them durable, and informs the resource managers about the transaction's outcome. Votes are durable once they have been decided in some consensus instance; the coordinator proposes them in a sequence of instances, batching as many votes as possible to amortize the cost of termination.

For high availability, the service should replace the coordinator as soon as it is suspected to have failed. However, suspecting the coordinator too aggressively may incur unnecessary changes and different views among resource managers about which process is the coordinator. Resource managers use the leader-election oracle to elect the coordinator. The use of consensus ensures that safety is not violated

even if several processes become coordinators simultaneously. Once the leader-election oracle works properly, liveness will be ensured.

The implementation of the log service is decomposed in two parts. In Section 5.1 we present a set of stubs that run at the RMs and are used by Algorithm 1 to interact with the coordinator; in Section 5.2 we present the coordinator's protocol.

5.1 RM Stubs

The first part of Algorithm 2 initializes the data structures kept by the stubs: $outcome$ is the local view of OUTCOME, and $myInc$ and $rmPID$ implement $\mathcal{R}[rm]$ locally.

Algorithm 2 Stubs to implement Algorithm 1.

```
1: Initially:
2:   ∀t, outcome[t] ← UNDEFINED                    ◁ All transactions are undecided.
3:   myInc ← ⊥                                     ◁ Have not incarnated yet.
4:   rmPID ← ⊥                                     ◁ Have not incarnated yet.

5: INCARNATE(rm, pid)
6:   send ⟨RECOVER, pid, rm⟩ to current coordinator.
7:   wait until receive ⟨RECOVERED, rm, upd, inc⟩
8:   myInc ← inc
9:   rmPID ← pid                                   ◁ pid has incarnated rm.
10:  return upd

11: R[rm]
12:  return rmPID                                  ◁ Either my own pid or ⊥.

13: VOTE(⟨rm, t, tset, vote, upd⟩)
14:  if vote = ABORT
15:    outcome[t] ← ABORT                          ◁ Quickly abort.
16:  send ⟨VOTE, rm, myInc, t, tset, vote, upd⟩ to coordinator

17: OUTCOME(t)
18:  return outcome[t]

19: when receive ⟨TERMINATED, t, out⟩              ◁ Learn decision.
20:  outcome[t] ← out                              ◁ Learn t's outcome.

21: when receive ⟨INCARNATE, rm, newInc⟩           ◁ Was replaced.
22:  newInc > myInc
23:    rmPID ← ⊥                                   ◁ No longer incarnates rm.
```

The INCARNATE stub sends a message RECOVER to the coordinator requesting to exchange the incarnation of resource manager rm to process pid, and waits for confirmation that the change was performed. The confirmation carries the updates executed by previous incarnations and the incarnation number of rm. $\mathcal{R}[rm] = pid$ evaluates to TRUE until the stub learns a bigger incarnation number, indicating that another processes took over the rm. The case in which more than one process believe to hold the rights on the rm is handled by the coordinator. To account for coordinator crashes, every time a process executes INCARNATE, it does it with a different pid.

VOTE forwards the resource manager's vote to the coordinator. As an optimization, if the vote is for ABORT, the resource manager aborts the transaction locally, as ABORT is the unique possible outcome of such a transaction.

The **when** clauses at the end of the algorithm execute fairly and atomically once their conditions become TRUE.

5.2 Log Service

The coordinator runs Algorithm 3. $\mathcal{T}_c$ implements $\mathcal{T}$; besides the votes of committed transactions, it also keeps the incarnation requests. $\mathcal{V}_c$ implements $\mathcal{V}$ as in the specification. B is the set of votes received but not yet treated by the coordinator. The coordinator executes a series of consensus instances, and i is the identifier of the instance the coordinator is waiting to terminate. Finally, $recSet$ is the set of reincarnation requests awaiting to be decided.

$\mathcal{R}[rm]$ is not explicitly kept in any data structure, but lies implicit in the subsequence of INCARNATE votes in $\mathcal{T}_c$: $\mathcal{R}[rm]$ equals the process in the last INCARNATE vote in $\mathcal{T}_c$.

When VOTE messages are received by the coordinator, they are added to B, to be proposed in the next consensus instance. Once B is no longer empty, the coordinator proposes it on instance i. For simplicity, we assume that B always fits in a consensus proposal. In some implementations of consensus, proposals may have bounded size. In such cases, B would have to be split and proposed in several consensus instances.

The coordinator waits the decision of instance i and postpones the learning of decisions of later instances. Once the coordinator learns the decision D, it first removes it from B, and then processes each of its elements. If the element is a vote then the coordinator determines its final meaning before adding it to $\mathcal{V}_c$. A vote's final meaning is ABORT if its issuer no longer equals $\mathcal{R}[rm]$; otherwise, it is the issuer's proposed vote. This ensures that upon recovery the new rm learns all committed transactions.

$\langle$RECOVER, pid, $rm\rangle$ messages asking the coordinator to change the $\mathcal{R}[rm]$ to pid are handled in the last part of the algorithm. Once such a message is received, the coordinator adds an INCARNATE vote to B. $\langle$INCARNATE, pid, $rm\rangle$ requests are handled after all the normal votes. They are simply appended to $\mathcal{T}_c$, meaning that $\mathcal{R}[rm]$ has been set to pid until another $\langle$INCARNATE, pid', $rm\rangle$ is appended to $\mathcal{T}_c$. Once the coordinator learns that $\mathcal{R}[rm]$ changed to pid, it gathers the updates performed by rm. These updates are sent to process pid, the new incarnation of rm, and a warn about the change is sent to all resource managers.

6 Uncoordinated Implementation

The main idea behind the uncoordinated log service implementation is to save time by not going through a central process. Instead of forwarding messages to the coordinator and waiting for replies as in the coordinated version, participants access local data structures and execute consensus.

Algorithm 4 defines stubs to Algorithm 1. The resource manager rm keeps $\mathcal{V}_{rm}$, the set of votes it has received, $commitCounter$, the counter of committed transactions involving rm, and $rmPID$, used to locally evaluate $\mathcal{R}[rm]$. The system's multiprogramming level, that is, the maximum number of transactions that can be executed concurrently by a resource manager corresponds to k. Therefore, k is also the maximum number of started transactions yet to commit, and is paramount to the uncoordinated approach. In the coordinated approach, the coordinator changes the incarnation of a resource manager by having an INCARNATE transaction added to $\mathcal{T}_c$. Because $\mathcal{T}_c$ grows deterministically due to the ordering of consensus instances and their decisions, any process acting as the coordinator will see incarnation requests in the same order. In the uncoordinated approach, however, there is no total order of decisions, and a process can consider itself incarnated only after a consensus instance corroborates it,

Algorithm 3 Coordinator's protocol.

1: Initialization:
2: $\mathcal{T}_c \leftarrow \emptyset$ ◁ *The sequence of terminated transactions.*
3: $\mathcal{V}_c \leftarrow \emptyset$ ◁ *The set of durable votes.*
4: $B \leftarrow \emptyset$ ◁ *Votes to be broadcast.*
5: $i \leftarrow 0$ ◁ *Current consensus instance.*
6: $recSet \leftarrow 0$ ◁ *Awaiting reincarnation transactions.*

7: $\mathcal{R}[rm]$
8: $rrm \leftarrow r = \langle \text{INCARNATE}, pid, rm\rangle \in \mathcal{T}_c : \forall r' = \langle \text{INCARNATE}, pid', rm\rangle \in \mathcal{T}_c, r >_{\mathcal{T}_c} r'$
9: return $rrm[2]$

10: **when**receive $\langle \text{VOTE}, rm, pid, t, tset, vote, upd\rangle$
11: **if**$\neg\exists\langle rm, \_, t, \_, \_, \_\rangle \in B$
12: $B \leftarrow B \cup \{\langle rm, pid, t, tset, vote, upd\rangle\}$ ◁ *Only the first vote is considered.*

13: **when**$B \neq \emptyset$
14: propose(i, B) ◁ *Propose B on instance i.*

15: **when** decide(i, D) ◁ *Decided D on instance i.*
16: $B \leftarrow B \setminus D$ ◁ *Remove from next proposals.*
17: $C \leftarrow \emptyset$ ◁ *Temporary variable.*
18: **for all**$\langle rm, pid, t, tset, vote, upd\rangle \in D$
19: **if** $\neg\exists\langle rm, \_, t, \_, \_, \_\rangle \in \mathcal{V}_c$
20: $prevState \leftarrow \text{OUTCOME}(t)$
21: **if** $\mathcal{R}[rm] \neq pid$ ◁ *If rm has been reincarnated*
22: $\mathcal{V}_c \leftarrow \mathcal{V}_c \cup \langle rm, t, tset, \text{ABORT}, \emptyset\rangle$ ◁ *turn the vote into* ABORT
23: **else**
24: $\mathcal{V}_c \leftarrow \mathcal{V}_c \cup \{\langle rm, t, tset, vote, upd\rangle\}$ ◁ *otherwise add it to the set of votes.*
25: **if**$(prevState = \text{UNDEFINED}) \wedge (\text{OUTCOME}(t) = \text{COMMIT})$ ◁ *If vote lead to commit of t*
26: $C \leftarrow C \cup \{t\}$ ◁ *add it to C*
27: $\forall p \in tset$, send message $\langle \text{TERMINATED}, t, \text{OUTCOME}(t)\rangle$ to p ◁ *and warn participants.*
28: **else if** $vote = \text{ABORT}$ ◁ *If lead to abortion of t*
29: $\forall p \in tset$, send message $\langle \text{TERMINATED}, t, \text{ABORT}\rangle$ to p ◁ *warn all possible.*
30: $\mathcal{T}_c \leftarrow \mathcal{T}_c \bullet C$ ◁ *Store the partial order.*
31: **for all** $d = \langle \text{INCARNATE}, \_, \_\rangle \in D$ ◁ *Process* INCARNATE *votes.*
32: $\mathcal{T}_c \leftarrow \mathcal{T}_c \bullet d$
33: $i \leftarrow i + 1$ ◁ *Process the next batch.*

34: **when** receive $\langle \text{RECOVER}, pid, rm\rangle$ ◁ *Upon request to recover*
35: $t_{inc} \leftarrow \langle \text{INCARNATE}, pid, rm\rangle$ ◁ *create a "reincarnation transaction"*
36: $recSet \leftarrow recSet \bigcup \{t_{inc}\}$ ◁ *put it in the awaiting set*
37: $B \leftarrow B \bigcup t_{inc}$ ◁ *and make it terminate.*

38: **when** $\exists\ t_{inc} = \langle \text{INCARNATE}, pid, rm\rangle \in recSet : t_{inc} \in \mathcal{T}_c$ ◁ *Once a reincarnation transaction terminates*
39: $recSet \leftarrow recSet \setminus \{t_{inc}\}$
40: $\mathcal{V}^{rm} \leftarrow \{e = \langle rm, t, \_, \_, \_\rangle \in \mathcal{V}_c : (\text{OUTCOME}(t) = \text{COMMIT}) \wedge (t <_{\mathcal{T}_c} t_{inc})\}$ ◁ *determine the updates already performed*
41: $\mathcal{U}^{rm} \leftarrow \langle u : \langle rm, \_, \_, u\rangle \in \mathcal{V}^{rm}\rangle, \forall\langle rm, t_1, \_, u_1\rangle, \langle rm, t_2, \_, u_2\rangle \in \mathcal{V}^{rm}, u_1 <_{\mathcal{U}^{rm}} u_2 \Rightarrow t_1 <_{\mathcal{T}_c} t_2\}$ ◁ *ordered as in $\mathcal{T}$.*
42: $inc \leftarrow |\ \{r = \langle \text{INCARNATE}, \_, rm\rangle \in \mathcal{T} : r <_{\mathcal{T}} \langle \text{INCARNATE}, pid, rm\rangle\}\ |$ ◁ *Determine the incarnation number*
43: send $\langle \text{RECOVERED}, rm, \mathcal{U}^{rm}, inc\rangle$ to rm ◁ *and warn interested processes.*
44: send $\langle \text{INCARNATE}, rm, inc\rangle$ to all resource managers

i.e., an INCARNATE transaction is decided, and it is sure that all the consensus instances that the previous incarnation had possibly voted have been treated. There can be up to $k-1$ of such consensus instances.

6.1 Transaction Termination

The VOTE stub proposes rm's vote for transaction t on consensus instance $inst(rm, t)$. If the rm is voting on its own behalf, the current consensus instance is stored locally. To vote for other processes, the rm must know the instances they attributed to t—notice that the rm only cares about other resource managers if it is voting COMMIT. The rm can learn the consensus instance of other rm's from the transaction manager, when it receives the commit request. The transaction manager receives the consensus instance from the resource managers during the execution of the transactions. With the vote, the resource manager proposes also the number of transactions already committed, $commitCounter$. This information is used later during recovery, as we be explain below.

The decision of a consensus instance j is learned by rm if it is one of its instances, or one associated to a transaction in which rm is participant. This decision can be of three types: (i) a vote, in which case it is added to $\mathcal{V}_{rm}$; (ii) a change of incarnation of another resource manager, in which case it is added to $\mathcal{V}_{rm}$ as an ABORT vote; and (iii) a change of rm's incarnation to process pid', in which case the process $pid \neq pid'$ currently incarnating learns that its incarnation lasts only until instance $j+k$. The **when** clause that processes the decisions is only activated once the resource manager has completed recovery.

6.2 Recovering from Failures

The INCARNATE stub determines the updates performed by previous incarnations and in which instance the new incarnation starts. The procedure consists of three steps: in the first step, pid (i.e., the process incarnating rm) proposes $\langle \text{INCARNATE}, pid, rm \rangle$ in its first consensus instance and until such a value is decided in an instance i, at which point pid becomes the incarnation of rm for transactions associated with consensus instances $i+k$, until displaced by another process. The second step terminates the transactions associated with instances $i+1$ to $i+k-1$, belonging to the incarnations being replaced. The third step determines the outcome of all transactions associated with instances smaller than $i+k$.

The protocol terminates by gathering the updates of committed transactions in the sequence $\mathcal{U}_{rm}$. Elements in $\mathcal{U}_{rm}$ are ordered according to the commit counter in each vote, making it consistent with the commit order of non-concurrent transactions. Algorithm 4 does not keep the $\mathcal{T}$ data structure nor any abridged version of it; the order of committed transactions is forgotten by the resource managers once their updates are applied. Upon recovery, $\mathcal{U}_{rm}$ is created and kept just until the recovery is terminated.

To improve performance, implementations should use checkpoints and reduce the number of consensus instances in which resource managers must propose. Most of the remaining instances can run in parallel to reduce the recovery latency.

7 Evaluation

We performed the experiments in a cluster of nodes from the Emulab testbed [13]. Nodes were equipped with 64-bit Xeon 3GHz processors, and interconnected through a Gigabit Ethernet switch. Our communication library used datagrams of size up to 7500B.

In the graphs, points represent the average of values sampled after execution stabilization in each run of an experiment.

Algorithm 4 Algorithm 1 stubs and Algorithm 1 uncoordinated implementation, with multi-programming level equal to k (at resource manager rm).

1: Initialization
2: $\mathcal{V}_{rm} \leftarrow \emptyset$ ◁ *Votes I have seen.*
3: $commitCounter \leftarrow 0$ ◁ *How many transactions I committed.*
4: $rmPID \leftarrow \perp$ ◁ *Have not incarnated yet.*

5: $\mathcal{R}[rm]$
6: return $rmPID$ ◁ *Either my own pid or* $\perp$.

7: OUTCOME(t)
8: **if** $\exists\langle \_, t, \_, \text{ABORT}, \_\rangle \in \mathcal{V}_{rm}$ ◁ *Any* ABORT*s?*
9: ABORT
10: **else if** $\exists\langle \_, t, tset, \_, \_\rangle \in \mathcal{V}_{rm} : \forall s \in tset :$
$\langle s, t, tset, \text{COMMIT}, \_\rangle \in \mathcal{V}_{rm}$ ◁ *All* COMMIT*s?*
11: COMMIT
12: **else**
13: UNDEFINED ◁ *Neither one nor the other*

14: VOTE($\langle rm, t, tset, vote, update_{rm}\rangle$)
15: propose($inst(rm, t), \langle rm, t, tset, vote, upd, Len(\mathcal{T}_{rm})\rangle$) ◁ *Vote + number of committed.*

16: **when** decide(j, d) ◁ *[Decided on instance* j*.]*
17: **if** $d = \langle r, t, tset, vote, upd, cn\rangle), rm \in tset$
18: $\mathcal{V}_{rm} \leftarrow \mathcal{V}_{rm} \cup \{\langle r, t, tset, vote, upd\rangle\}$
19: **else if** $d = \langle \text{INCARNATE}, \_, r\rangle, r \neq rm$ ◁ *Someone else was substituted.*
20: $\mathcal{V}_{rm} \leftarrow \mathcal{V}_{rm} \cup \{\langle r, t, \{r, rm\}, \text{ABORT}, \emptyset\rangle\}, inst(rm, t) = j$ ◁ *Make* d *an* ABORT *vote.*
21: **else if** $d = \langle \text{INCARNATE}, pid', rm\rangle, pid \neq pid'$ ◁ rm *has been reincarnated,*
22: $rmPID \leftarrow \perp$ ◁ *so give it up.*

23: INCARNATE(rm)
24: $tranSet \leftarrow \emptyset$
25: $i \leftarrow 0$ ◁ *Assume that 0 is the first instance identifier.*
26: **while**TRUE
27: propose($i, \langle \text{INCARNATE}, pid, rm\rangle$) ◁ *Vote until incarnation changes,*
28: **wait until** decide(i, d)
29: **if** $d = \langle \text{INCARNATE}, pid, rm\rangle$
30: $pidRM \leftarrow pid$ ◁ *Reincarnated*
31: **break** ◁ *Stop the first iteration.*
32: **else**
33: $tranSet \leftarrow tranSet \bigcup \{d\}$
34: $i \leftarrow i + 1$
35: **for** $j \leftarrow 1..k - 1$ ◁ *and then, for* $k - 1$ *possibly open instances,*
36: propose($i + j, \langle rm, t, \{rm\}, \text{ABORT}, \emptyset, 0\rangle$) ◁ *vote to close them.*
37: **wait** until decide($i + j$, d)
38: **if** $d = \langle \text{INCARNATE}, \_, rm\rangle$ ◁ *Someone else is recovering.*
39: $rmPID \leftarrow \perp$ ◁ *Give up the resource manager*
40: return $\langle\rangle$ ◁ *and stop looking for updates.*
41: $tranSet \leftarrow tranSet \bigcup \{d\}$
42: **for each** $\langle rm, t, l, v, u, cn\rangle \in tranSet$ ◁ *Close all of those transactions:*
43: **if** $\neg\exists\langle p', t, l', \text{ABORT}, \emptyset, cn'\rangle \in V_{rm}$ ◁ *If* t *was not aborted,*
44: **for all** $p' \in l, \neg\exists\langle p', t, l, v', u', cn'\rangle \in V_{rm}$ ◁ *make sure to terminate it*
45: propose($inst(p', t), \langle p', t, l, \text{ABORT}, \emptyset, 0\rangle$) ◁ *voting for others if needed.*
46: **wait** until decide($inst(p', t)$, d)
47: $V_{rm} \leftarrow V_{rm} \bigcup \{d\}$
48: **if** $d = \langle p', t, \_, \text{ABORT}, \_, \_\rangle$ ◁ *Stop on* ABORT.
49: **break**
50: $U_{rm} \leftarrow \langle u_1, ..., u_m\rangle : e_k = \langle rm, t_k, l_k, v_k, u_k, cn_k\rangle \in V_{rm},$ ◁ *Now order updates of known transactions*
$\forall p \in l_k, \exists\langle p, t_k, l_k, \text{COMMIT}, u_p\rangle \in V_{rm},$ ◁ *that committed*
$\forall i < j, cn_i < cn_j$ ◁ *in commit order.*
51: return U_{rm}

7.1 Micro-benchmark

In the first experiment we used a micro-benchmark of non-conflicting transactions, each comprising one update operation being executed in parallel at 8 RMs, and resulting, at each of them, in 100 or 7000 B of data to be logged. Figure 1 shows that for both update sizes the coordinated implementation outperforms the uncoordinated one by an order of magnitude. In case of small updates the coordinated version must perform eight times less consensus instances and in addition the coordinator can batch votes from up to nine transactions executed concurrently, while our implementation of a single RM is not capable of this grouping. For big updates the coordinator has to run as many instances as the uncoordinated version does, but the difference might be explained by the consensus executions' time: The coordinator serializes its instances so they do not interfere with each other and can be timely finished, what is supported by our observation that only a few retries happened. In the uncoordinated implementation all the RMs try to execute their instances at the same time, and as the network contention leads to many unsuccessful termination attempts, the execution becomes driven by the timeouts used.

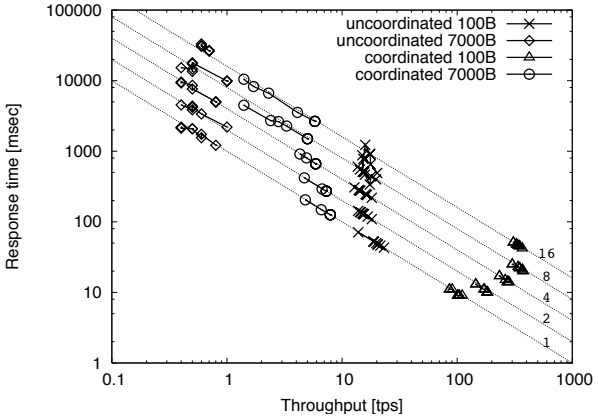

Figure 1: Throughput versus response time of the micro-benchmark. The number of clients is shown next to the lines.

7.2 The TPC-C benchmark

For TPC-C the read-only Items table was replicated on all RMs and other tables were range partitioned among RMs according to the warehouse id, so at most 15% transactions involved more than one RM. Update transactions were 92% of the workload and none of them produced data to be logged exceeding 1500B.

As Figure 2 shows, very small loads do not differ significantly in performance between configurations. By higher loads the coordinated version outperforms the uncoordinated one and the former scales much better with the number of transactions executed concurrently, which is determined by the number of client threads and RMs. Although in 85% of the cases a single transaction requires one consensus instance regardless of the termination protocol, the coordinated version can group at least five simultaneous requests, and even when a single RM could perform the batching on its own the coordinator still have higher potential for that as it can combine data from different RMs.

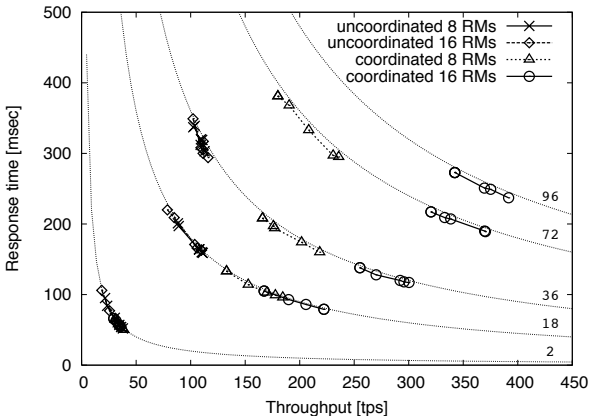

Figure 2: Throughput versus response time of TPC-C transactions. The number of clients is shown next to the curves.

8 Related Work

Stamus and Cristian [12] propose to aggregate log records generated by resource managers at the transaction managers and logged there. This approach resembles the log service we are proposing. In fact, the transaction manager acts as a log service to the transactions it managers. However, their approach uses a byzantine agreement abstraction with strong synchrony assumptions, and do not consider recovery of resource and transaction managers in nodes different from the initial ones.

The log service proposed by Daniels *et al.* [4] does consider recovery on different machines, but it considers one instance of the service per resource manager instead of a shared one, and therefore cannot provide the termination facilities of our approach.

The use of consensus or agreement protocols in atomic commitment protocols has been explored in some other works, but none of them considers the problem of interleaving commits and transaction ordering. Mohan et al. [11] used byzantine tolerant agreement abstractions to extend a two phase commit protocol with byzantine reliability. As in our protocols, the recovery of resource managers in their approach is based on the log provided by the agreement boxes, although not abstracted by a log service. However, they keep the "two-phases approach", even though one agreement phase should suffice to solve the problem.

Guerraoui *et al.* [6] presented a non-blocking protocol that terminates transactions in three communication steps, from termination request to transaction termination, in good runs (i.e., without failures and suspicions). In bad runs, the protocol resorts to consensus to agree on the outcome, exchanging at least one communication step for a consensus instance.

More recently, Lamport and Gray [5] presented the Paxos Commit protocol, a non-blocking generalization of two-phase commit using the Paxos consensus protocol. The biggest difference between this protocol and the previous one is that Paxos Commit uses consensus to decide on votes, not on the transaction's outcome. As Paxos does not require all participants to propose, a suspicious resource manager votes on the suspect's behalf as soon as suspecting it, and may have its conservative vote or the suspect's original vote decided in two communication steps, increasing the protocol's latency by the time to suspect a participant. Having several consensus instances in parallel, for possibly many transactions, has a bad impact on the overall performance. We looked for a way to isolate this impact and it was by mimicking Paxos' separation of concerns that the log service was devised.

9 Conclusion

In this paper we have introduced the specification of a log service for transaction processing systems. The log service provides atomicity and durability to transactions through a non-blocking termination protocol. The service totally orders non-concurrent transactions and, should a resource manager fail, the service can be used to recover the resource manager's state prior to the crash and start a copy of it on a different and functional node. The service safely copes with multiple copies of a resource manager.

We also presented two highly available implementations of our log service, namely coordinated and uncoordinated, and provided a comparative experimental performance evaluation. In the studied scenarios, a coordinated approach has led to a much higher transaction throughput and smaller response times.

References

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

[2] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[3] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Communications of the ACM*, 43(2):225–267, 1996.

[4] D. S. Daniels, A. Z. Spector, and D. S. Thompson. Distributed logging for transaction processing. In U. Dayal and I. Traiger, editors, *Proceedings of the ACM SIGMOD Annual Conference*, pages 82–96, San Francisco, CA, 1987. ACM, ACM Press.

[5] J. Gray and L. Lamport. Consensus on transaction commit. *ACM TODS*, 31(1):133–160, March 2006.

[6] R. Guerraoui, M. Larrea, and A. Schiper. Reducing the cost for non-blocking in atomic commitment. In *ICDCS '96: Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS '96)*, page 692, Washington, DC, USA, 1996. IEEE Computer Society.

[7] L. Lamport. Introduction to tla. Technical Report 1994-001, Palo Alto, CA, 1994.

[8] L. Lamport. Refinement in state-based formalisms, 1996.

[9] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[10] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Professional, July 2002.

[11] C. Mohan, R. Strong, and S. Finkelstein. Method for distributed transaction commit and recovery using byzantine agreement within clusters of processors. *SIGOPS Oper. Syst. Rev.*, 19(3):29–43, July 1985.

[12] J. W. Stamos and F. Cristian. Coordinator log transaction execution protocol. *Distributed and Parallel Databases*, 1(4):383–408, 1993.

[13] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, Dec. 2002. USENIX Association.

A Log Service

The specifications presented in Sections 3, 5, and 6 are simplified versions of our real specifications. For the unabridged versions, we used Lamport's Temporal Logic of Actions (TLA$^+$) specification language [10]. The language borrows most of its formalism from basic mathematics, and reading it should be straightforward except maybe for a few constructs. The cheat-sheet in [7] should be enough to clarify any other doubts.

A.1 Log Service Constants

This module specifies constants used by all the other specifications.

MODULE *LogServiceConstants*

This module specifies the *LogService*'s constants.

The service's constant parameters:

RM	the set of resource managers.
TID	the set of transaction ids.
Update	the type update.
ApplyUpdate	interface to database.
GetUpdate	update at r, by t.

CONSTANTS RM, TID, $Update$,
$ApplyUpdate(\_)$,
$GetUpdate(\_, \_)$

$NoRM \triangleq$ CHOOSE $r : r \notin RM$

$NoTID \triangleq$ CHOOSE $t : t \notin TID$

The environments's constant parameters:

PID	the set of all processes.

CONSTANTS PID

$NoPID \triangleq$ CHOOSE $p : p \notin PID$

A.2 Log Service Specification

MODULE *LogService*

This module specifies the *LogService*. That is, it specifies a centralized log service would look like, and how resource managers should use it.

EXTENDS $Naturals$, $FiniteSets$, $Sequences$,
$LogServiceConstants$

The environment's variables:

badProc	crashed processes
suspect	suspicions.

VARIABLES $badProc$, Environments's
$suspect$ Who suspects whom?

The Service's variables:

$vHist$	the history of votes.
$tHist$	the history of committed transactions
$LastConcSet$	LastConcSet concurrent to the last committed.
$terminatingAt$	transactions terminating at rm.
$terminatedAt$	transactions terminated at rm.

VARIABLES $vHist$, $tHist$, $rm2pid$, $LastConcSet$ Log service's

The Resource Managers' variables:

$terminatingAt$	transactions terminating at rm.
$terminatedAt$	transactions terminated at rm.

VARIABLES $terminatingAt$, $terminatedAt$, $pid2rm$ Resource Manager's

The Transaction Managers's variables:

$termReq$	transactions requested to terminate.
$part$	participants on each transaction.

VARIABLES $termReq$, $part$ Transaction Managers'

Defining some aliases.

$svars \triangleq \langle vHist,\ tHist,\ LastConcSet,\ rm2pid \rangle$
$rvars \triangleq \langle terminatingAt,\ terminatedAt,\ pid2rm \rangle$
$tvars \triangleq \langle part,\ termReq \rangle$
$evars \triangleq \langle badProc,\ suspect \rangle$
$avars \triangleq \langle svars,\ rvars,\ tvars,\ evars \rangle$

Types

$UNKNOWN \triangleq$ CHOOSE $v : v \notin \{$TRUE, FALSE$\}$

$Votes \triangleq [rm : RM,\ tr : TID,\ tset :$ SUBSET $RM,\ vt : \{$"Commit", "Abort"$\},\ upd : Update]$

Invariants

Type invariant.

$$
\begin{array}{ll}
TypeInvariant \triangleq & \\
\quad \land\ vHist & \in \text{SUBSET } Votes \\
\quad \land\ tHist & \in Seq(\text{SUBSET } TID) \\
\quad \land\ LastConcSet & \in \text{SUBSET } TID \\
\quad \land\ rm2pid & \in [RM \to PID \cup \{NoPID\}] \\
\quad \land\ terminatingAt & \in [PID \to \text{SUBSET } TID] \\
\quad \land\ terminatedAt & \in [PID \to \text{SUBSET } TID] \\
\quad \land\ pid2rm & \in [PID \to RM \cup \{NoRM\}] \\
\quad \land\ part & \in [TID \to \text{UNION } \{[S \to PID] : S \in \text{SUBSET } RM\}] \\
\quad \land\ termReq & \in \text{SUBSET } TID
\end{array}
$$

$\land\ badProc \quad \in \text{SUBSET } PID$
$\land\ suspect \quad \in [PID \to [PID \to \text{BOOLEAN }]]$

Initial State

$Init \triangleq$

$\land\ vHist$	$= \{\}$	No vote received.
$\land\ tHist$	$= \langle\rangle$	No transaction committed,
$\land\ LastConcSet$	$= \{\}$	
$\land\ rm2pid$	$= [r \in RM \mapsto NoPID\ ]$	No RM is incarnated.
$\land\ terminatingAt$	$= [p \in PID \mapsto \{\}]$	No transactions terminating at any RM.
$\land\ terminatedAt$	$= [p \in PID \mapsto \{\}]$	No transactions terminating at any RM.
$\land\ pid2rm$	$= [r \in PID \mapsto NoRM]$	No RM is incarnated.
$\land\ part$	$= [t \in TID \mapsto [e \in \{\} \mapsto \{\}]]$	No participant in any transaction.
$\land\ termReq$	$= \{\}$	No termination request issued.
$\land\ badProc$	$= \{\}$	No bad process.
$\land\ suspect$	$= [p \in PID \mapsto [q \in PID \mapsto \text{FALSE}]]$	

Operators

Operator $IsInvolved(t,\ rm)$ checks whether rm is involved in transaction t.

$$
\begin{array}{l}
IsInvolved(t,\ rm) \triangleq \text{IF } \exists\, v \in vHist : v.tr = t \land rm \in v.rm \\
\qquad \text{THEN TRUE} \\
\qquad \text{ELSE IF } \exists\, v \in vHist : v.tr = t \land v.vt = \text{“Commit”} \\
\qquad\qquad \text{THEN FALSE} \\
\qquad\qquad \text{ELSE } UNKNOWN
\end{array}
$$

Operator $Updates(rm)$ gets the updates performed by rm in committed transactions.

$$
\begin{array}{l}
Updates(rm) \triangleq \\
\quad \text{LET } UpdateOn(t) \triangleq (\text{CHOOSE } v \in vHist : v.tr = t \land v.rm = rm).upd \\
\\
\qquad upd[i \in 0\,..\,Len(tHist)] \triangleq \\
\qquad\quad \text{IF } i = 0 \\
\qquad\quad \text{THEN } \langle\rangle \\
\qquad\quad \text{ELSE } Append(upd[i-1], \{UpdateOn(t) : t \in \{e \in tHist[i] : \\
\qquad\qquad\qquad\qquad\qquad\qquad IsInvolved(rm,\ e) = \text{TRUE}\}\}) \\
\\
\quad \text{IN } \quad upd[Len(tHist)]
\end{array}
$$

Operator $Outcome(h,\ t)$ gives the termination status of transaction t, considering the history of votes h.

$$
\begin{array}{l}
Outcome(h,\ t) \triangleq \text{IF } \exists\, v \in h : v.tr = t \land v.vt = \text{“Abort”} \\
\qquad\qquad \text{THEN “Abort”} \\
\qquad\qquad \text{ELSE IF } \exists\, v \in h : \\
\qquad\qquad\qquad \land\ v.tr = t \\
\qquad\qquad\qquad \land\ \forall\, p\ \in v.tset : \\
\qquad\qquad\qquad\quad \exists\, vv \in h : \land\ vv.rm = p \\
\qquad\qquad\qquad\qquad\qquad\ \land\ vv.tr\ = t
\end{array}
$$

$\wedge\ vv.vt = \text{“Commit”}$
THEN “Commit”
ELSE “Undefined”

Actions

Environment

This action crashes a good process.

$$Crash \triangleq \wedge \exists\, p \in (PID \setminus badProc) : badProc' = badProc \cup \{p\}$$
$$\wedge \text{ UNCHANGED } \langle suspect \rangle$$

This action changes the suspicion status.

$$ChangeSuspicion \triangleq \wedge \exists\, p \in (PID \setminus badProc),\ q \in PID :$$
$$\wedge\ p \neq q$$
$$\wedge\ suspect' = [suspect \text{ EXCEPT } ![p][q] = \neg@]$$
$$\wedge \text{ UNCHANGED } \langle badProc \rangle$$

EnvActions:
- process crash.
- change suspicions.

$$EnvActions \triangleq \wedge \vee Crash$$
$$\vee\ ChangeSuspicion$$
$$\wedge \text{ UNCHANGED } \langle svars,\ rvars,\ tvars \rangle$$

Transaction Manager

Adds a resource manager to a non-terminated transaction.

The transaction manager will only succeed in the adding a resource manager r to a transaction t if r was not crashed by the time it was contacted, and replied to operation request.

$$AddRM \triangleq \wedge \exists\, t \in (TID \setminus termReq),$$
$$r \in \{r \in RM : \wedge\ rm2pid[r] \neq NoPID \quad \text{r has been incarnated,}$$
$$\wedge\ rm2pid[r] \notin badProc\} : \quad \text{replied to an operation,}$$
$$\wedge\ r \notin \text{DOMAIN } part[t] \quad \text{and was not a participant yet}$$
$$\wedge\ part' = [part \text{ EXCEPT } ![t] =$$
$$[e \in \text{DOMAIN } @ \cup \{r\} \mapsto \text{IF } e \in \text{DOMAIN } @$$
$$\text{THEN } @[e]$$
$$\text{ELSE } rm2pid[r]]] \quad \text{now it is.}$$
$$\wedge \text{ UNCHANGED } \langle termReq,\ rm2pid \rangle$$

Request the termination of a transaction.

The transaction manager can, at any time, try to terminate a transaction it has not tried to terminate before, and that executed some operation.

$$RequestTerm \triangleq \wedge \exists\, t \in (TID \setminus termReq) : \quad t \text{ has not tried to terminate yet.}$$
$$\wedge\ part[t] \neq \langle\rangle$$
$$\wedge\ termReq' = termReq \cup \{t\}$$

$\land$ UNCHANGED $\langle part,\ rm2pid \rangle$

The disjunction of Transaction Manager's actions.
- Add a resource manager as a participant.
- Try to terminate a transaction.

$$
\begin{array}{lll}
TMActions \triangleq & \land & \lor\ AddRM \\
& & \lor\ RequestTerm \\
& \land & \text{UNCHANGED } \langle svars,\ rvars,\ evars \rangle
\end{array}
$$

Log Service

Action $Incarnate(pid,\ rm)$ is executed by process pid to incarnate resource manager rm.

$$
\begin{array}{l}
Incarnate \triangleq \\
\quad \text{LET } ApplyUpdates(u) \triangleq \\
\qquad \text{LET } apply[i \in 1\ ..\ Len(u)] \triangleq \\
\qquad\quad \text{IF } i = Len(u) \text{ THEN } ApplyUpdate(u[i]) \\
\qquad\qquad\qquad\quad \text{ELSE } ApplyUpdate(u[i]) \land apply[i+1] \\
\qquad \text{IN} \quad \text{IF } u = \langle\rangle \text{ THEN TRUE} \\
\qquad\qquad\qquad\quad \text{ELSE } apply[1] \\
\quad \text{IN} \quad \exists\, p \in PID \setminus badProc,\ r \in RM : \\
\qquad \land\ pid2rm[p] = NoRM \qquad \text{but is not incarnating,} \\
\qquad \land\ \lor\ rm2pid[r] = NoPID \qquad \text{and } r \text{ is not incarnated} \\
\qquad\quad \lor\ rm2pid[r] \neq NoPID \land suspect[p][rm2pid[r]] \qquad \text{or its process is supected.} \\
\qquad \land\ rm2pid' = [rm2pid \text{ EXCEPT } ![r] = p] \\
\qquad \land\ pid2rm' = [pid2rm \text{ EXCEPT } ![p] = r] \\
\qquad \land\ ApplyUpdates(Updates(r)) \\
\qquad \land \text{ UNCHANGED } \langle terminatingAt,\ terminatedAt,\ vHist,\ tHist,\ LastConcSet \rangle
\end{array}
$$

Action $Vote(v)$ adds v to $vHist$, if not there yet.

$$
\begin{array}{lll}
Vote(v) \triangleq & & \\
\quad \text{LET } ConcSet(vh) \triangleq & \{t \in \{v.tr : \land\ v \in vh\} : & \text{Voted transactions} \\
& \quad Outcome(vh,\ t) = \text{“Undefined”}\} & \text{not terminated yet.} \\
\qquad Commits \triangleq & \land\ Outcome(vHist',\ v.tr) = \text{“Commit”} & \\
& \land\ \lor\ \land\ v.tr \in LastConcSet & \\
& \qquad \land\ tHist' = [tHist \text{ EXCEPT } ![Len(tHist)] = @ \cup \{v.tr\}] & \\
& \qquad \land\ LastConcSet' = LastConcSet \setminus \{v.tr\} & \\
& \quad \lor\ \land\ v.tr \notin LastConcSet & \\
& \qquad \land\ tHist' = Append(tHist,\ \{v.tr\}) & \\
& \qquad \land\ LastConcSet' = ConcSet(vHist') \setminus \{v.tr\} & \\
& \land \text{ UNCHANGED } rm2pid & \\
\qquad Aborts \triangleq & \land\ Outcome(vHist',\ v.tr) = \text{“Abort”} & \\
& \land \text{ UNCHANGED } \langle tHist,\ LastConcSet,\ rm2pid \rangle &
\end{array}
$$

```
IN   ∧ ¬∃ ov ∈ vHist : ∧ ov.rm = v.rm
                       ∧ ov.tr  = v.tr          Add v to vHist
     ∧ vHist' = vHist ∪ {v}
     ∧ StayUndef ∨ Commits ∨ Aborts
```

Action $Terminate(rm,\ t)$ is executed by rm to step towards transaction t's termination.

```
VoteForMyself(r, t) ≜
    ∧ t ∈ (termReq
        \ (terminatingAt[rm2pid[r]] ∪ terminatedAt[rm2pid[r]]))   and has not tried to terminate t.
    ∧ terminatingAt' = [terminatingAt EXCEPT ![rm2pid[r]] = @ ∪ {t}]   Start the termination
    ∧ ∨ Vote([rm ↦ r, tr ↦ t, tset ↦ DOMAIN part[t],
              vt  ↦ "Commit", upd ↦ GetUpdate(r, t)])          voting commit.
      ∨ Vote([rm ↦ r, tr ↦ t, tset ↦ {r}, vt ↦ "Abort", upd ↦ {}])   voting abort.
    ∧ UNCHANGED ⟨terminatedAt⟩

VoteForOthers(r, t) ≜
    ∧ t ∈ terminatingAt[rm2pid[r]]                  rm tried to terminate t
    ∧ Outcome(vHist, t) = "Undefined"               t is stuck.
    ∧ ∃ s ∈ DOMAIN part[t] :
      ∧ suspect[rm2pid[r]][rm2pid[s]]
      ∧ Vote([rm ↦ s, tr ↦ t, tset ↦ DOMAIN part[t],
              vt  ↦ "Abort", upd ↦ {}])
    ∧ UNCHANGED ⟨terminatingAt, terminatedAt⟩

Learn(r, t) ≜
    ∧ Outcome(vHist, t) ≠ "Undefined"               t has terminated.
    ∧ terminatingAt' = [terminatingAt EXCEPT ![rm2pid[r]] = @ \ {t}]   So rm learns it.
    ∧ terminatedAt'  = [terminatedAt  EXCEPT ![rm2pid[r]] = @ ∪ {t}]
    ∧ UNCHANGED ⟨svars⟩

Terminate ≜
  ∃ r ∈ RM, t ∈ TID :
    ∧ r ∈ DOMAIN part[t]                            r is a participant of t.
    ∧ part[t][r] ∉ badProc                          the process is alive,
    ∧ part[t][r] = rm2pid[r]                        and is still the same
    ∧ ∨ VoteForMyself(r, t)
      ∨ VoteForOthers(r, t)
      ∨ Learn(r, t)
    ∧ UNCHANGED ⟨pid2rm⟩

RMActions ≜
    ∧ ∨ Incarnate
      ∨ Terminate
    ∧ UNCHANGED ⟨tvars, evars⟩
```

Specification

$$
\begin{aligned}
Next \;&\triangleq\; \lor\, RMActions \\
&\lor\, TMActions \\
&\lor\, EnvActions
\end{aligned}
$$

$$Spec \;\triangleq\; Init \land \Box[Next]_{\langle svars,\, tvars,\, rvars,\, evars\rangle}$$

Theorems

The specification is type safe.

THEOREM $Spec \Rightarrow \Box\, TypeInvariant$

AC-Validity If an RM decides to commit a transaction, then all RMs voted to commit the transaction.

$$
\begin{aligned}
AC\_Validity \;\triangleq\; &\forall\, t \in TID : Outcome(vHist,\, t) = \text{“Commit”} \\
&\Rightarrow \forall\, r \in \text{DOMAIN } part[t] : \\
&\quad \exists\, v \in vHist : \land\, v.rm = r \\
&\qquad\qquad\quad \land\, v.tr \;\; = t \\
&\qquad\qquad\quad \land\, v.vote = \text{“Commit”}
\end{aligned}
$$

AC-Agreement It is impossible for one RM to commit a transaction and another one to abort the transaction.

$AC\_Agreement \;\triangleq\;$ TRUE From the definition of $Outcome$.

AC-Non-Triviality If all RMs vote to commit the transaction and no RM is suspected throughout the execution of the protocol, then the decision is commit.

$$
\begin{aligned}
AC\_Non\_Triviality \;\triangleq\; &\forall\, t \in TID : \land\, \forall\, r,\, s \in part[t] : \Box(\neg suspect[rm2pid[r]][rm2pid2[s]]) \\
&\qquad\qquad \land\, \forall\, r \in part[t] : \exists\, v \in vHist : \land\, v.rm = r \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\, v.t = t \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\, v.vt = \text{“Commit”} \\
&\qquad\qquad \Rightarrow \\
&\qquad\qquad \land\, \Diamond Outcome(vHist) = \text{“Commit”}
\end{aligned}
$$

AC-Termination Non-faulty RMs eventually decide.

$$
\begin{aligned}
AC\_Termination \;\triangleq\; &\land\, \forall\, t \in TID : \Diamond\Box(Outcome(vHist,\, t) \in \{\text{“Commit”},\, \text{“Abort”}\}) \\
&\land\, \forall\, t \in TID : \forall\, r \in part[t] : \Diamond\Box(t \in terminatedAt[r])
\end{aligned}
$$

THEOREM $Spec \Rightarrow \land\, AC\_Validity \land AC\_Non\_Triviality \land AC\_Termination$

A.3 Correctness

We want to prove that the log service's specification (LSS) we gave at Section 3 satisfies the atomic commitment and the R-Consistency properties, recalled below. We first prove an invariant of the algorithm and then proceed to prove each property individually.

- **AC-Validity** If an RM decides to commit a transaction, then all RMs voted to commit the transaction.
- **AC-Agreement** It is impossible for one RM to commit a transaction and another one to abort the transaction.

- **AC-Non-Triviality** If all RMs vote to commit the transaction and no RM is suspected throughout the execution of the protocol, then the decision is commit.
- **AC-Termination** Non-faulty RMs eventually decide.
- **R-Consistency** The database state of an RM after the recovery from a failure is the same as its committed state before the failure.

Invariant 1 *At any point in time there is at most one process incarnating any given resource manager.*

PROOF: At the initial state, defined by $Init$, no resource manager is incarnated by any process. Resource managers are incarnated only by executing action $Incarnate$, that replaces the previous resource manager by the new one. Therefore, at most one process can be incarnating a resource manager at any point in time, and this relation is kept in $rm2pid$. □

Invariant 2 *Given a transaction t, if $Outcome(vHist, t) \neq$"Undefined" at some point in time, then it will equal $Outcome(vHist, t)$ at any later time.*

PROOF: The variable $vHist$ is only changed in action $Vote$, where a vote v is added to $vHist$ only if there was no other vote for the same transaction $v.tr$ and resource manager $v.rm$ in$vHist$. Therefore, the only change allowed to $vHist$ is the addition of new votes.
At the initial state $vHist = \{\}$ and, by the definition of action $Outcome$, $Outcome(\{\}, t)$ equals "Undefined". From this state, "Abort" and "Commit" votes for t can be added to $vHist$.
By the definition of $Outcome$, if an "Abort" is ever added, then $Outcome(vHist, t)$ will equal "Abort" at any future evaluation. If a "Commit" vote is added for each resource manager involved in t, then $Outcome(vHist, t)$ will equal "Commit" and, because no "Abort" vote for t could be added later, it will equal "Commit" at any future evaluation. □

Proposition A.1 (AC-Valitidy) *LSS satisfies the AC-Validity property.*

PROOF: By the specification, the only action that issues "Commit" votes is $VoteForMyself$. As its name suggests, the action is executed for a resource manager r only to cast its own vote and, therefore, a vote v such that $v.vt =$"Commit" and $v.rm = r$ can only be casted by r itself.
By specification, $Outcome(vHist, t)$ will equal "Commit" only if a "Commit" vote has been issued for t from all resource managers involved. By the previous paragraph, we know that each resource manager involved in t must have issued its own "Commit" vote for t, and the **AC-Validity** property is true. □

Proposition A.2 (AC-Agreement) *LSS satisfies the AC-Validity property.*

PROOF: Suppose that $Outcome(vHist, t)$ evaluated to "Commit" at some point in time for some transaction t. By the initial state it was initially evaluated to "Undefined" and turned to "Commit" at some later state. By the Invariant A.2, it must have turned directly from "Undefined" to "Commit", and will be "Commit" on any future evalution. Therefore, any resource manager either sees "Commit" or "Undefined"; because a resource manager will give t for terminated only if $Outcome(vHist, t) \neq$"Undefined", all resource managers will see the same termination outcome. Changing "Commit" for "Abort" renders the equivalent result, and therefore the **AC-Agreement** property is true. □

Proposition A.3 (AC-Non-Triviality) *LSS satisfies the AC-Non-Triviality property.*

PROOF: By the specification, for any transaction t, a resource manager will only vote "Abort" abort on behalf of another resource manager if it suspects that it is crashed. If there are no suspicions, then all resource manager will vote for themselves. If no resource manager votes "Abort" for itself, then only "Commit" votes will be issued and added to $vHist$. Because, by assumption, all crashes are eventually suspected, the lack of suspicions implies that no resource manager crashed. Therefore, eventually all resource managers involved in t will have their votes added to $vHist$ and, at this point in time, $Outcome(vHist, t)$ will turn from "Undefined" to "Commit", satisfying the **AC-Non-Triviality** property. □

To prove the next property we must assume some kind of fairness on the system; we assume the following: actions that become enabled and remain in such state until executed, are eventually executed. This property is equivalent to the Weak Fairness described in [10].

Proposition A.4 (AC-Termination) *LSS satisfies the AC-Termination property.*

Assuming that any transaction t that is started will eventually be requested to terminate, resource managers will eventually crash or vote for themselves. As, by assumption, non-faulty resource managers eventually suspect any crashed ones and vote on their behalf if they have issued their own votes, eventually some "Abort" vote or all "Commit" votes for t will be gathered by the log service, t will be terminated, and resource managers that did not crash will learn the transaction's outcome, hence, satisfying the **AC-Termination** property. □

Proposition A.5 (R-Consistency) *LSS satisfies the R-Consistency property.*

Let s be the state of some given database. We denote by $s.u$ the state obtained by an update u to the database at state s. To prove the the **R-Consistency** property we use the following assumptions regarding this change of states. We call two transactions "non-concurrent" if their termination procedure was executed within non-overlapping periods of time.

Assumption 1 *Applying an update is a deterministic operation. That is, given two databases at states s_1 and s_2 and and update u, $(s_1 = s_2) \Rightarrow (s_1.u = s_2.u)$.*

Assumption 2 *Updates to different data items are commutable. That is, given two updates u_1 and u_2 and a database in some state s, if u_1 and u_2 do not write to the same data item, then $s.u_1.u_2 = s.u_2.u_1$.*

Assumption 3 *Concurrent transactions do not access the same data items. I.e., if two transactions execute their commit procedure in parallel, then they do not read items written by each other.*

PROOF SKETCH: We divide the proof in several steps. First we show that any non-concurrent transactions that committed, who possibly accessed the same data items, are totally ordered in $tHist$, according to their termination order. Then we show that the reincarnation procedure apply the updates of committed transactions in the same order they were committed (the order in $tHist$) and, finally, show that this ensures that a recovered resource manager has the same committed stated as before crashing or being replaced.

1. ASSUME: $(C, \leq_C)$ is the poset where C is the set of committed transactions of some run of the system, $\leq_C$ their commit order, and
   $vHist = (H, \leq_H)$.
   PROVE: $\forall t_1, t_2 \in C : t_1 \leq_C t_2 \Rightarrow t_1 \leq_H t_2$
   PROOF: Let t_1 and t_2 be two non-concurrent transactions that committed and, without loss of generality, let t_1 be the one to be committed first. By the specification, all resource managers involved in t_1 executed the action $Vote$ with "Commit" vote for t_1 before any has voted for t_2.
   By the definition of $Vote$, when the first vote for t_1 was issued, t_1 was added to $ConcSet$. After the last vote, t_1 was removed from $ConcSet$ and added to $vHist$, and $LastConcSet$ was changed to a set not containing t_2. When t_2 receives its first vote to commit, it is added to $ConcSet$, and upon the last vote, it is removed from $ConcSet$ and added to $vHist$. Because t_2 cannot belong to the $LastConcSet$ defined when t_1 was committed, t_2 will belong to a set in $tHist$ different from the one t_1 belongs.
   Recalling the meaning of the data-structure $tHist$, if a transaction t_1 belongs to the set $tHist[i]$ for some natural number i, then $t_1 \leq_H t_2$, for any transaction t_2 that belongs to $tHist[j]$ for any $j > i$.
2. ASSUME: r is a resource manager,
   t is a transaction that committed, and
   r was involved in t.
   PROVE: If $Updates(r)$ is evaluated after t committed, then
   $\exists i \in 1..Len(Updates(r)) : Updates(r)[i] = u$,
   where u are the updates executed by r on transaction t.
   PROOF: Because t committed, by the definition of $Vote$, there must be a set in $tHist$ to whom t belongs. By de definition of $Updates$ and $IsInvolved$, r will be identified as participant of t and, by the definition of $UpdateOn$, inside the definition of $Updates$, r's updates on t are in $Updates(r)$.
3. ASSUME: r is a resource manager,
   t_1 and t_2 are non-concurrent transactions,
   t_1 and t_2 committed,
   t_1 committed before t_2,
   r was involved in t_1 and t_2,
   r's updates on t_1 and t_2 are u_1 and u_2, respectively.
   PROVE: $(Updates(r)[i] = u_1) \wedge (Updates(r)[j] = u_2) \Rightarrow i < j$
   PROOF: The definition of $Updates$ constructs $Updates(r)$ accessing $tHist$ backwards, but orderly. Because each update is added to before the previous one in the sequence, the final result is that updates are in the same order as their respective transaction. By the step 2, both u_1 and u_2 are in $Updates(r)$ and, by step 1, t_1 is ordered before t_2 in $tHist$, and u_1 appear before u_2 in $Updates(r)$.
4. Q.E.D.
   PROOF: By the definition of $ApplyUpdates$, updates are applied sequentially. Consequently, by steps 1,2, and 3, all updates of non-concurrent transactions are applied in the recovering resource manager in the same order they were originally executed. By Assumption 3, concurrent transactions access only different items and are, by Assumption 2, commutable. Finally, by Assumption 3, the recovering resource manager must have the same committed as in the previous incarnation. □

B Coordinated Implementation

B.1 Specification

MODULE *CoordLogService*

This module is specifies the log service's Coordinated Implementation.

EXTENDS *Naturals*, *FiniteSets*, *Sequences*,
LogServiceConstants,
Consensus

CONSTANTS *Coord*

The environment's variables:

badProc	processes that crashed
suspect	process X process suspicions.
msgs	messages sent.

VARIABLES *badProc*,
suspect,
msgs

The Coordinators's variables:

vHistAt	votes received per coordinator.
tHistAt	transactions that committed.
recSet	set of awaiting recovery transactions.
bSet	sets of votes to be proposed.
instances	instance to be used by the coordinator(s).

VARIABLES *vHist*,
tHist,
recSet,
bSet,
instances

The Resource Managers' variables:

terminatingAt	transactions terminating at rm.
terminatedAt	transactions terminated at rm.
pid2rm	PID $\mapsto$ RM.
incarns	the incarnation of each rm.
outcome	local view of Outcome.

VARIABLES *terminatingAt*,
terminatedAt,
pid2rm,
incarns,
outcome

The Transaction Managers's variables:

termReq	transactions requested to terminate.
part	participants on each transaction, and incarnating process.

VARIABLES *termReq*,
part

Defining some aliases.

$svars \triangleq \langle vHist,\ tHist,\ recSet,\ bSet,\ instances\rangle$
$rvars \triangleq \langle terminatingAt,\ terminatedAt,\ pid2rm,\ incarns,\ outcome\rangle$
$tvars \triangleq \langle part,\ termReq\rangle$
$evars \triangleq \langle badProc,\ suspect\rangle$
$ovars \triangleq \langle decision\rangle$
$avars \triangleq \langle svars,\ rvars,\ tvars,\ evars,\ ovars,\ msgs\rangle$

Types

$Incarnating \triangleq \text{CHOOSE } p : p \notin (PID \cup \{NoRM\})$

Vote extended with *PID*

$EVotes \triangleq [rm : RM,\ tr : TID,\ tset : \text{SUBSET } RM,$
$\quad vt : \{\text{“Commit”, “Abort”}\},\ upd : Update,\ pid : PID]$

Votes to incarnate a resource manager.

$IncarnT \triangleq [vt : \{\text{“Incarnate”}\},\ pid : PID,\ rm : RM]$

Types of messages exchanged.

$Msgs \triangleq [type : \{\text{“Recover”}\},\quad rm : RM,\ pid : PID] \cup$
$[type : \{\text{“Recovered”}\},\ rm : RM,\ pid : PID,$
$\ upd : Seq(Update),\ inc : Nat] \cup$
$[type : \{\text{“Incarnated”}\},\ rm : RM,\ inc : Nat] \cup$
$[type : \{\text{“Vote”}\},\quad rm : RM,\ pid : PID,\ tr : TID,$
$\ tset : \text{SUBSET } RM,\ vt : \{\text{“Commit”, “Abort”}\},\ upd : Update] \cup$
$[type : \{\text{“Terminated”}\},\ tr : TID,\ out : \{\text{“Commit”, “Abort”}\}]$

Invariants

Type invariant.

$TypeInvariant \triangleq$
$\quad \wedge\ vHist \in \text{SUBSET } EVotes$
$\quad \wedge\ tHist \in Seq(TID \cup IncarnT)$
$\quad \wedge\ bSet \in [Coord \to \text{SUBSET } (EVotes \cup IncarnT)]$
$\quad \wedge\ recSet \in \text{SUBSET } IncarnT$
$\quad \wedge\ instances \in Nat$
$\quad \wedge\ terminatingAt \in [PID \to \text{SUBSET } TID]$
$\quad \wedge\ terminatedAt \in [PID \to \text{SUBSET } TID]$
$\quad \wedge\ pid2rm \in [PID \to RM \cup \{Incarnating,\ NoRM\}]$
$\quad \wedge\ outcome \in [PID \to [TID \to \{\text{“Undefined”, “Abort”, “Commit”}\}]]$
$\quad \wedge\ part \in [TID \to \text{UNION } \{[S \to PID] : S \in \text{SUBSET } RM\}]$
$\quad \wedge\ termReq \in \text{SUBSET } TID$
$\quad \wedge\ badProc \in \text{SUBSET } PID$
$\quad \wedge\ suspect \in [PID \to [PID \to \text{BOOLEAN }]]$
$\quad \wedge\ msgs \in \text{SUBSET } Msgs$

$\land\ ConsensusTypeInv$

Initial State

$Init \triangleq$

$\land\ vHist = \{\}$ — No vote received.

$\land\ tHist = \langle\rangle$ — No transaction committed,

$\land\ bSet = [c \in Coord \mapsto \{\}]$

$\land\ recSet = [c \in Coord \mapsto \{\}]$

$\land\ instances = 0$

$\land\ terminatingAt = [p \in PID \mapsto \{\}]$ — No transactions terminating.

$\land\ terminatedAt = [p \in PID \mapsto \{\}]$ — No transactions terminating.

$\land\ pid2rm = [r \in PID \mapsto NoRM]$ — No RM is incarnated.

$\land\ outcome = [p \in PID \mapsto [t \in TID \mapsto \text{“Undefined”}]]$

$\land\ incarns = [p \in PID \mapsto 0]$

$\land\ part = [t \in TID \mapsto [e \in \{\} \mapsto \{\}]]$ — No transaction started.

$\land\ termReq = \{\}$ — No termination request issued.

$\land\ badProc = \{\}$ — No bad process.

$\land\ suspect = [p \in PID \mapsto [q \in PID \mapsto \text{FALSE}]]$

$\land\ msgs = \{\}$

$\land\ ConsensusInit$

Operators

The pid of the currently incarnating r.

$rm2pid(r) \triangleq$

IF $\exists\, p \in PID,\ i \in 1\,..\,Len(tHist)$:
$\land\ tHist[i] = [vt \mapsto \text{“Incarnate”},\ pid \mapsto p,\ rm \mapsto r]$
$\land \neg\exists\, j \in i+1\,..\,Len(tHist),\ op \in PID$:
$tHist[j] = [vt \mapsto \text{“Incarnate”},\ pid \mapsto op,\ rm \mapsto r]$
THEN CHOOSE $p \in PID$:
$\exists\, i \in 1\,..\,Len(tHist)$:
$\land\ tHist[i] = [vt \mapsto \text{“Incarnate”},\ pid \mapsto p,\ rm \mapsto r]$
$\land \neg\exists\, j \in i+1\,..\,Len(tHist),\ op \in PID$:
$tHist[j] = [vt \mapsto \text{“Incarnate”},\ pid \mapsto op,\ rm \mapsto r]$
ELSE $NoPID$

Operator $Outcome(h,\ t)$ gives the termination status of transaction t, considering the history of votes h.

$Outcome(h,\ t) \triangleq$

IF $\exists\, v \in h : v.tr = t \land v.vt = \text{“Abort”}$
THEN “Abort”
ELSE IF $\exists\, v \in h$:
$\land\ v.tr = t$
$\land \forall\, p \in v.tset$:
$\exists\, vv \in h :\ \land\ vv.rm = p$

$\land\ vv.tr = t$
$\land\ vv.vt =$ "Commit"

THEN "Commit"
ELSE "Undefined"

Actions

Environment

This action crashes a good process.

$Crash \triangleq$
$\land \exists\, p \in (PID \setminus badProc) : badProc' = badProc \cup \{p\}$
$\land$ UNCHANGED $\langle suspect \rangle$

This action changes the suspicion status.

$ChangeSuspicion \triangleq$
$\land \exists\, p \in (PID \setminus badProc),\ q \in PID :$
$\land\ p \neq q$
$\land\ suspect' = [suspect$ EXCEPT $![p][q] = \neg@]$
$\land$ UNCHANGED $\langle badProc \rangle$

EnvActions:
- process crash.
- change suspicions.

$EnvActions \triangleq \land Crash \lor ChangeSuspicion$
$\land$ UNCHANGED $\langle msgs,\ svars,\ tvars,\ rvars,\ ovars \rangle$

Transaction Manager

Adds a resource manager to a non-terminated transaction.

The transaction manager will only succeed in the adding a resource manager r to a transaction t if r was not crashed by the time it was contacted, and replied to operation request.

$AddRM \triangleq$
$\land \exists\, t \in (TID \setminus termReq),$ — t has not tried to terminate yet.
$r \in \{r \in RM : \land rm2pid(r) \neq NoPID$ — r has been incarnated,
$\land rm2pid(r) \notin badProc\} :$ — replied to an operation,
$\land\ r \notin$ DOMAIN $part[t]$ — and was not a participant yet
$\land\ part' = [part$ EXCEPT $![t] =$
$[e \in$ DOMAIN $@ \cup \{r\} \mapsto$ IF $e \in$ DOMAIN $@$
THEN $@[e]$
ELSE $rm2pid(r)]]$ — now it is.
$\land$ UNCHANGED $\langle termReq \rangle$

Request the termination of a transaction.

The transaction manager can, at any time, try to terminate a transaction it has not tried to terminate before, and that executed some operation.

$RequestTerm \triangleq$ $\land \exists\, t \in (TID \setminus termReq) :$ t has not tried to terminate yet.
$\land\ part[t] \neq \langle\rangle$
$\land\ termReq' = termReq \cup \{t\}$
$\land$ UNCHANGED $\langle part \rangle$

The disjunction of Transaction Manager's actions.
- Add a resource manager as a participant.
- Try to terminate a transaction.

$TMActions \triangleq$ $\land\ \lor\ AddRM$
$\lor\ RequestTerm$
$\land$ UNCHANGED $\langle rvars,\ evars,\ svars,\ msgs,\ ovars \rangle$

Log Service

Starts the incarnate procedure.

$IncarnateStart(p,\ r) \triangleq$
$\land\ pid2rm[p] = NoRM$ p is neither incarnating nor trying
$\land\ \lor\ rm2pid(r) = NoPID$ and r is not incarnated
$\lor\ rm2pid(r) \neq NoPID \land suspect[p][rm2pid(r)]$ or its process is supected.
$\land\ pid2rm' = [pid2rm$ EXCEPT $![p] = Incarnating]$
$\land\ msgs' = msgs \cup \{[type \mapsto$ "Recover", $pid \mapsto p,\ rm \mapsto r]\}$
$\land$ UNCHANGED $\langle incarns \rangle$

Ends the incarnate procedure.

$IncarnateEnd(p,\ r) \triangleq$
LET $ApplyUpdates(u) \triangleq$
LET $apply[i \in 1\ ..\ Len(u)] \triangleq$
IF $i = Len(u)$ THEN $ApplyUpdate(u[i])$
ELSE $ApplyUpdate(u[i]) \land apply[i+1]$

IN IF $u = \langle\rangle$ THEN TRUE
ELSE $apply[1]$

IN $\land\ pid2rm[p] = Incarnating$
$\land \exists\, m \in msgs :$
$\land\ m.type =$ "Recovered"
$\land\ m.pid = p$
$\land\ m.rm = r$
$\land\ pid2rm' = [pid2rm$ EXCEPT $![p] = r]$
$\land\ incarns' = [incarns$ EXCEPT $![p] = m.inc]$
$\land\ ApplyUpdates(m.upd)$
$\land$ UNCHANGED $\langle msgs \rangle$

Executed by process p to give up incarnating r, when another process incarnates it.

$Desincarnate(p,\ r) \triangleq$
$\land\ incarns[p] > 0$
$\land \exists\, m \in msgs :$

$\land\ m.type = \text{“Incarnated”}$
$\land\ m.rm = r$
$\land\ m.inc > incarns[p]$
$\land\ incarns' = [incarns \text{ EXCEPT } ![p] = 0]$
$\land\ \text{UNCHANGED } \langle msgs,\ pid2rm \rangle$

IncarnateStub is a “stub” to the abstract log service *Incarnate* action.

$IncarnateStub \triangleq$
$\land \exists\, p \in PID \setminus badProc,\ r \in RM :$ p is good
$\lor\ IncarnateStart(p,\ r)$
$\lor\ IncarnateEnd(p,\ r)$
$\lor\ Desincarnate(p,\ r)$
$\land\ \text{UNCHANGED } \langle terminatingAt,\ terminatedAt,\ outcome \rangle$

VoteForMyself is executed to vote on some transaction.

$VoteForMyself(r,\ t) \triangleq$
$\land\ t \in (termReq \setminus (terminatingAt[part[t][r]] \cup terminatedAt[part[t][r]]))$
$\land\ terminatingAt' = [terminatingAt \text{ EXCEPT } ![part[t][r]] = @ \cup \{t\}]$ Start the termination
$\land\ \lor\ msgs' = msgs \cup \{[type \mapsto \text{“Vote”},\ pid \mapsto part[t][r],\ rm \mapsto r,$
$tr \mapsto t,\ tset \mapsto \text{DOMAIN } part[t],$
$vt \mapsto \text{“Commit”},\ upd \mapsto GetUpdate(r,\ t)]\}$ voting commit.
$\lor\ msgs' = msgs \cup \{[type \mapsto \text{“Vote”},\ pid \mapsto part[t][r],\ rm \mapsto r,$
$tr \mapsto t,\ tset \mapsto \{r\},\ vt \mapsto \text{“Abort”},\ upd \mapsto \{\}]\}$ voting abort.
$\land\ \text{UNCHANGED } \langle terminatedAt,\ outcome \rangle$

VoteForOthers is executed to vote for some slow participant.

$VoteForOthers(r,\ t) \triangleq$
$\land\ t \in terminatingAt[part[t][r]]$ rm tried to terminate t
$\land\ outcome[part[t][r]][t] = \text{“Undefined”}$ but did not succeed yet.
$\land \neg\exists\, m \in msgs :$
$\land\ m.type = \text{“Terminated”}$
$\land\ m.tr = t$
$\land \exists\, s \in \text{DOMAIN } part[t] :$
$\land\ suspect[part[t][r]][part[t][s]]$
$\land\ msgs' = msgs \cup \{[type \mapsto \text{“Vote”},\ pid \mapsto part[t][s],\ rm \mapsto s,\ tr \mapsto t,$
$tset \mapsto \text{DOMAIN } part[t],\ vt \mapsto \text{“Abort”},\ upd \mapsto \{\}]\}$ voting abort.
$\land\ \text{UNCHANGED } \langle terminatingAt,\ terminatedAt,\ outcome \rangle$

Learn action is performed when a new transaction has terminated.

$Learn(r,\ t) \triangleq$
$\land\ outcome[part[t][r]][t] = \text{“Undefined”}$ $outcome[t]$ is undefined
$\land \exists\, m \in msgs :$ but t terminated.
$\land\ m.type = \text{“Terminated”}$
$\land\ m.tr = t$
$\land\ outcome' = [outcome \text{ EXCEPT } ![part[t][r]][t] = m.out]$
$\land\ terminatingAt' = [terminatingAt \text{ EXCEPT } ![part[t][r]] = @ \setminus \{t\}]$

$\land\ terminatedAt' = [terminatedAt \ \text{EXCEPT}\ ![part[t][r]] = @ \cup \{t\}]$
$\land\ \text{UNCHANGED}\ \langle msgs \rangle$

Action $TerminateStub$ is executed by rm to step towards transaction t's termination.

It is a "stub" to the abstract log service's Terminate action.

$TerminateStub \triangleq$
$\land \exists\, r \in RM,\ t \in TID :$
$\land\ r \in \text{DOMAIN}\ part[t]$ — r is a participant of t.
$\land\ part[t][r] \notin badProc$ — the process is still alive,
$\land\ pid2rm[part[t][r]] = r$ — and sees itself as the rm.
$\land\ \lor\ VoteForMyself(r,\ t)$ — first attempt to terminate t.
$\lor\ VoteForOthers(r,\ t)$ — other attempts
$\lor\ Learn(r,\ t)$ — Learn that it was decided.
$\land\ \text{UNCHANGED}\ \langle incarns,\ pid2rm \rangle$

The disjunction of Resource Manager's actions.
- Execute the incarnation procedure.
- Try to terminate a transaction.

$RMActions \triangleq$
$\land\ \lor\ IncarnateStub$
$\lor\ TerminateStub$
$\land\ \text{UNCHANGED}\ \langle tvars,\ evars,\ ovars,\ svars \rangle$

$IncarnateRequest \triangleq$
$\land \exists\, p \in PID,\ r \in RM,\ c \in Coord,\ m \in msgs :$
$\land\ m.type = \text{"Recover"} \land m.pid = p \land m.rm = r$
$\land \neg\exists\, i \in 1\,..\,Len(tHist) :$
$tHist[i] = [vt \mapsto \text{"Incarnate"},\ pid \mapsto p,\ rm \mapsto r]$ — p is not incarnating.
$\land\ recSet' = [recSet \ \text{EXCEPT}\ ![c] = @ \cup \{[vt \mapsto \text{"Incarnate"},\ pid \mapsto p,\ rm \mapsto r]\}]$
$\land\ bSet' = [bSet \ \text{EXCEPT}\ ![c] = @ \cup \{[vt \mapsto \text{"Incarnate"},\ pid \mapsto p,\ rm \mapsto r]\}]$
$\land\ \text{UNCHANGED}\ \langle vHist,\ tHist,\ instances,\ ovars,\ msgs \rangle$

$IncarnateReply \triangleq$
$\text{LET}\ Urm(tinc) \triangleq$
$\text{LET}\ try[i \in 0\,..\,tinc] \triangleq$
$\text{IF}\ i = 0$
$\text{THEN}\ \langle\rangle$
$\text{ELSE}\ try[i-1] \circ \text{IF}\ \land\ tHist[i] \in TID$ — Not reincarn transaction,
$\land \exists\, v \in vHist :$
$\land\ v.tr = tHist[i]$ — is committed before $tinc$,
$\land\ v.rm = tHist[tinc].rm$ — r took part in it.
$\text{THEN}\ \langle(\text{CHOOSE}\ v \in vHist :$
$\land\ v.tr = tHist[i]$
$\land\ v.rm = tHist[tinc].rm).upd \rangle$
$\text{ELSE}\ \langle\rangle$

```
        IN    try[tinc]

    Inc(tinc) ≜ Cardinality({i ∈ 1 .. tinc :
                                ∧ tHist[i] ∉ TID
                                ∧ tHist[i].rm = tHist[tinc].rm})

IN  ∃ tinc ∈ 1 .. Len(tHist), c ∈ Coord :
      ∧ tHist[tinc] ∈ recSet[c]
      ∧ recSet' = [recSet EXCEPT ![c] = @ \ {tHist[tinc]}]
      ∧ msgs' = msgs ∪ {[type ↦ "Recovered", rm ↦ tHist[tinc].rm, upd ↦ Urm(tinc),
                         inc ↦ Inc(tinc), pid ↦ tHist[tinc].pid],
                        [type ↦ "Incarnated", rm ↦ tHist[tinc].rm, inc ↦ Inc(tinc)]}
      ∧ UNCHANGED ⟨vHist, tHist, bSet, instances, ovars⟩

VoteRequest ≜
  ∃ c ∈ Coord, m ∈ msgs :
  ∧ m.type = "Vote"
  ∧ ¬∃ ev   ∈ (bSet[c] ∩ EVotes) ∪ vHist :
     ∧ ev.rm = m.rm
     ∧ ev.tr  = m.tr
     ∧ bSet'  = [bSet EXCEPT ![c] = @ ∪ {[pid ↦ m.pid, rm ↦ m.rm, tr ↦ m.tr,
                                          tset ↦ m.tset, vt ↦ m.vt, upd ↦ m.upd]}]
  ∧ UNCHANGED ⟨vHist, tHist, recSet, instances, msgs, ovars⟩

CoordPropose ≜
  ∧ ∃ c ∈ Coord :
     ∧ bSet[c] ≠ {}
     ∧ Propose(instances, bSet[c])
  ∧ UNCHANGED ⟨svars, msgs⟩

CoordDecide ≜
  LET D ≜ Decide(instances)

      EVotesInD ≜ {v ∈ D : ∧ v.vt ∈ {"Commit", "Abort"}
                           ∧ ¬∃ ov ∈ vHist : ∧ ov.rm = v.rm
                                             ∧ ov.tr  = v.tr}

      V2V(v)    ≜ [rm ↦ v.rm, tr ↦ v.tr, tset ↦ v.tset, pid ↦ v.pid,
                   vt  ↦ IF rm2pid(v.rm) = v.pid THEN v.vt ELSE "Abort",
                   upd ↦ IF rm2pid(v.rm) = v.pid THEN v.upd ELSE {}]

      VotesInD  ≜ {V2V(v) : v ∈ EVotesInD}

      Set2Seq(S) ≜
        LET set2seq[SS ∈ SUBSET S] ≜
          IF SS = {} THEN ⟨⟩
                     ELSE LET ss ≜ CHOOSE ss ∈ SS : TRUE
                          IN  Append(set2seq[SS \ {ss}], ss)
```

IN $set2seq[S]$

$newCommitted \triangleq \{t \in TID : \land Outcome(vHist, t) = \text{“Undefined”}$
$\land Outcome(vHist', t) = \text{“Commit”}\}$

$newTermMsgs \triangleq \{[type \mapsto \text{“Terminated”}, tr \mapsto t, out \mapsto \text{“Commit”}] :$
$t \in newCommitted\}$
$\cup$
$\{[type \mapsto \text{“Terminated”}, tr \mapsto v.tr, out \mapsto \text{“Abort”}] :$
$v \in \{vv \in VotesInD : vv.vt = \text{“Abort”}\}\}$

$IncarnationReqInD \triangleq \{i \in D : i.vt = \text{“Incarnate”}\}$

IN $\land D \neq NoProposal$
$\land vHist' = vHist \cup VotesInD$
$\land msgs' = msgs \cup newTermMsgs$
$\land tHist' = tHist \circ (\text{IF } newCommitted \neq \{\} \text{ THEN } Set2Seq(newCommitted) \text{ ELSE } \langle\rangle)$
$\circ Set2Seq(IncarnationReqInD)$
$\land instances' = instances + 1$
$\land bSet' = [c \in Coord \mapsto bSet[c] \setminus D]$
$\land \text{UNCHANGED } \langle recSet, ovars\rangle$

The disjunction of Coordinator's actions.
- Process requests to incarnate a resource manager.
- Process votes from resource managers.
- Handle consensus instances.

$CoordActions \triangleq$
$\land \lor IncarnateRequest \lor IncarnateReply$
$\lor VoteRequest$
$\lor CoordPropose \lor CoordDecide$
$\land \text{UNCHANGED } \langle tvars, evars, rvars\rangle$

Specification

The next-state action, as a disjunction of all possible action.

$Next \triangleq$
$\lor RMActions \lor CoordActions$ Implement $RMActions$
$\lor TMActions$ Implement $TMActions$
$\lor EnvActions$ Implement $EnvActions$

The specification.

$Spec \triangleq Init \land \Box[Next]_{\langle avars\rangle}$

Refinement Mapping

$rm\_pid2rm \triangleq [r \in RM \mapsto \text{IF } pid2rm[p] = \text{“Incarnating”} \text{ THEN } NoRM \text{ ELSE } pid2rm[p]]$

$rm\_rm2pid \triangleq [r \in RM \mapsto rm2pid(r)]$

$rm\_LastConcSet \triangleq \{\}$

$rm\_vHist \triangleq \{[f \in \text{DOMAIN } v \setminus \{\text{“pid”}\} \mapsto v.f] : v \in vHist\}$

$rm\_tHist \triangleq \text{LET } Test(e) \triangleq e \notin IncarnT$
$\text{IN } \quad SelectSeq(vHist, Test)$

B.2 Implementation Proof

To prove that the Coordinated Log Service (CLS) is, indeed, an implementation of the Log Service's specification (LS) we give a refinement mapping of the CLS's variables to the LS's, and show that the execution of CLS's actions implies the execution of one of LS's actions, or in a stuttering step. For a thorough explanation of refinement mappings the reader is referred to [8] and [1].

We substitute every expression of the specification for an overlined expression with the same name, meaning that any variable defined in its scope is replaced by an overlined one, with the same name; these overlined variables witness the implementation of the specification. We prove that these witnesses exist by defining them from the variables in the implementation, i.e., by giving a refinement mapping.

The actual refinement is defined at the end of the specification. Below we simply rename each definition to conform the overlined notation. Variables that are not redefined are the same as in the implementation.

$\overline{pid2rm} \triangleq rm\_pid2rm$

$\overline{rm2pid} \triangleq rm\_rm2pid$

$\overline{LastConcSet} \triangleq rm\_LastConcSet$

$\overline{vHist} \triangleq rm\_vHist$

$\overline{tHist} \triangleq rm\_tHist$

Proposition B.1 $Spec \Rightarrow \overline{Spec}$

1. ASSUME: $Init$
   PROVE: $\overline{Init}$
   PROOF: Except for $LastConcSet$ and $rm2pid$, all the variables are initialized in $Init$ exactly as their overlined counterparts in $\overline{Init}$. By the refinement mapping, $\overline{LastConcSet}$ is always the empty set, therefore conforming the initialization in $\overline{Init}$. Finally, by the definition of operator $rm2pid$, $tHist = \langle\rangle$ implies that $rm2pid$ maps from all resource managers to $NoPid$.

2. ASSUME: $Next$
   PROVE: $\overline{Next} \vee \text{UNCHANGED } \langle\overline{svars}, \overline{tvars}, \overline{rvars}, \overline{evars}\rangle$
   2.1. ASSUME: $RMActions$
   PROVE: $\overline{RMActions}$

2.1.1. ASSUME: $IncarnateStub \land \text{UNCHANGED } \langle tvars, evars \rangle$
PROVE: $\Diamond \land \overline{Incarnate} \lor \text{UNCHANGED } \langle svars, rvars \rangle$
$\land \text{UNCHANGED } \langle \overline{tvars, evars} \rangle$

PROOF SKETCH: We want to show that the execution of $IncarnateStart(p, r)$ for some process p and resource manager r leads to the execution $IncarnateEnd(p, r)$, if p does not crash and its messages are lost. Because $\underline{IncarnateEnd(p, r)}$ can only be executed if $IncarnateStart(p, r)$ was previously executed and because the pre-conditions of $\overline{Incarnate}$ shared with $IncarnateStart$ do not change until $IncarnateEnd$ is executed, and the conditions of $\overline{Incarnate}$ shared with $\underline{IncarnateEnd}$ complement the set of $\overline{Incarnate}$ pre and post-conditions already true, the execution of $IncarnateEnd$ implies an $\overline{Incarnate}$ step. If just the start action is performed, then it implies a stuttering step of $\overline{Spec}$.

PROOF: The $IncarnateStub$ action is a disjunction of actions

- $IncarnateStart$,
- $IncarnateEnd$, and
- $Desincarnate$

It is clear by the specification that action $IncarnateEnd(p, r)$ cannot execute for a resource manager r and process p before an $IncarnateStart(p, r)$ is executed: $IncarnateEnd(p, r)$ only executes after receiving a message $m = [type \mapsto \langle Recovered \rangle, rm \mapsto r, pid \mapsto p]$, and such message will not be sent by action $IncarnateReply$ before a vote $v = [vt \mapsto \text{“}\texttt{Incarnate}''$, $pid \mapsto p, rm \mapsto r]$ is added to $tHist$. Hence, v will only be added to $tHist$ in action $CoordDecide$, after being proposed in action $CoordPropose$. $CoordPropose$ can only propose such value if it belongs to $bSet[c]$, for some coordinator c, what can only happen if a request message for p to incarnate r is received in action $IncarnateRequest$, and such message is only sent by the execution of action $IncarnateStart(p, r)$.

When $IncarnateStart(p, r)$ is executed, it adds a “`Recover`” message with fields $pid = p$ and $rm = r$ to $msgs$; this is the first pre-condition for $IncarnateRequest(p, r)$ to execute. The second pre-condition is satisfied until action $CoordDecide$ adds the “`Incarnate`” vote for p and r to $tHist$, what can only happen if $IncarnateRequest$ has been executed first, since $IncarnateRequest$ is the only action that creates “`Incarnate`” votes.

When the action $IncarnateRequest$ is executed, it adds an “`Incarnate`” vote for p and r to $bSet[c]$, for some coordinator c, making it not empty. This is the only pre-condition for the $CoordPropose$ action execute for coordinator c, and the action is eventually executed. Since coordinators insist on proposing its $bSet[c]$ until it is empty, and only removes votes from it if they are decided in some instance, c will keep proposing the “`Incarnate`” vote until it is decided or c crashes. Coordinators are deterministic state machines, and can be replicated at will (their state is only based on the outcomes of consensus instances) and, therefore, as long as coordinators can recover after crashes or infinitely many of them are available, some coordinator eventually completes the execution of $IncarnateRequest$ and $IncarnateReply$.

By the consensus problem definition, **C-Progress** ensures that a decision will eventually be reached on each instance (given that the minimum number of acceptors eventually stay up long enough for the instances to finish). If p crashes, then the request for incarnation is simply discarded or is decided but will be followed by another request for the same r.

When an instance containing the vote for p to incarnate r is decided, the vote is added to $tHist$, and action $IncarnateReply$ will be enabled for all coordinators that proposed it. If c crashes before this action is performed, p will be blocked and never execute another action, as if it had crashed. Because the change made by $IncarnateStart(r, p)$ to variable $pid2rm[p]$ does not affect $\overline{pid2rm}$, this would imply that $\langle \overline{svars, rvars} \rangle$ did not change.

Once $IncarnateReply$ is performed, p will eventually receive the “`Recovered`” message, unless c crashes, enabling action $IncarnateEnd$. By the definition of Urm, p will receive all the

updates performed by previous incarnations of r. By the definition of $ApplyUpdates$, the p will apply all the updates and recover the committed state r had on its previous incarnation. The pre condition of $IncarnateStub$, $IncarnatedStart$, and $IncarnatedEnd$, and the post-conditions of $IncarnateEnd$ imply the pre and post-conditions of $\overline{Incarnate}$.

By the assumption, variables in $\langle \overline{tvars}, \overline{evars} \rangle$ do not change.

2.1.2. ASSUME: $TerminateStub \wedge \text{UNCHANGED } \langle tvars, evars, ovars, svars \rangle$
PROVE: $\Diamond \wedge \overline{Terminate} \vee \text{UNCHANGED } \langle \overline{svars}, \overline{rvars} \rangle$
$\wedge \text{UNCHANGED } \langle \overline{tvars}, \overline{evars} \rangle$

PROOF SKETCH: The pre-conditions of action $TerminateStub$ are the same as those of $\overline{Terminate}$. Therefore, it is enough to show that each of $TerminateStub$'s sub-actions, $VoteForMyself$, $VoteForOthers$, $Learn$, and the actions they lead to, imply a step of their equivalent overlined actions.

2.1.2.1. ASSUME: $\overline{VoteForMyself(r, t)}$
PROVE: $\Diamond \wedge \overline{VoteForMyself(r, t)} \vee \text{UNCHANGED } \langle \overline{terminatingAt, vHist, tHist} \rangle$
$\wedge \text{UNCHANGED } \langle \overline{terminatedAt} \rangle$

Because the pre-condition and the first post-condition of both actions are the same, it is enough to prove that the second post-condition of $\overline{VoteForMyself(r, t)}$, the addition of a message m to $msgs$, may the execution of $\overline{Vote(v)}$, where m is "`Vote`" message and v is a vote, and the fields $vt, upd, tr, tset$, and rm of m and v are equal, or has no effect on variables $\langle \overline{terminatingAt, vHist, tHist} \rangle$.

If message m is received by some coordinator c, a vote with its contents, therefore equal to v, is added to $bSet[c]$, enabling the action $CoordPropose$. Action $CoordPropose$ will be executed with a proposal containing this vote until it is decided and added to $vHist$ by action $CoordDecide$, where all coordinators can see it ($vHist$ is changed deterministically based on the consensus outcomes, and would be the same for all coordinators if represented independently at each one.), or until c crashes. If no coordinator succeeds in getting the vote decided, then either another vote, resulting from the execution of $VoteForOthers(r, t)$ will be decided, or all resource managers involved in transaction t will have crashed before their "`Vote`" messages are seen by non-faulty coordinators. It is up to the transaction manager, to then vote to abort the transaction; in the case the transaction manager also crashes and no vote for t is ever decided, t is simply forgotten, implying that $\langle \overline{svars}, \overline{rvars} \rangle$ is kept unchanged.

$CoordDecide$ also appends newly committed transactions to $tHist$: each transaction is added in a different set, as if they were not concurrent, and $\overline{LastConcSet}$ is always empty, ensuring its type invariance and constructing $tHist$ in a way compatible with the specification of $\overline{tHist}$. Because this is the only action to change $vHist$ and $tHist$, the mapping to $\overline{vHist}$ and $\overline{tHist}$ is correct.

$\langle \overline{tvars}, \overline{evars} \rangle$ are kept since none of actions changes them.

2.1.2.2. ASSUME: $\overline{VoteForOthers(r, t)}$
PROVE: $\Diamond \wedge \overline{VoteForOthers(r, t)} \vee \text{UNCHANGED } \langle \overline{vHist, tHist} \rangle$
$\wedge \text{UNCHANGED } \langle \overline{terminatedAt, terminatingAt} \rangle$

PROOF: It is true by the same arguments of step 2.1.2.1.

2.1.2.3. ASSUME: $\overline{Learn(r, t)}$
PROVE: $\wedge \overline{Learn(r, t)} \vee \text{UNCHANGED } \langle \overline{terminatedAt, terminatingAt} \rangle$
$\wedge \text{UNCHANGED } \langle \overline{svars} \rangle$

PROOF: As action $Learn(r, t)$ has the same post-conditions of $\overline{Learn(r, t)}$, it is enough to show that the pre-conditions of the first imply the pre-conditions of the latter. Since the reception of "`Terminated`" message for transaction t implies that it was sent and since it is only

sent by action $CoordDecide$ if the transaction has terminated, the reception of such message implies that the $\overline{Outcome(vHist, t)} \neq$ "`Undefined`".

2.1.2.4. Q.E.D.

2.1.3. Q.E.D.

2.2. $TMActions \Rightarrow \overline{TMActions}$

2.2.1. $AddRM \Rightarrow \overline{AddRM}$

PROOF: Trivially true, since the their definitions are equal.

2.2.2. $RequestTerm \Rightarrow \overline{RequestTerm}$

PROOF: Trivially true, since the their definitions are equal.

2.2.3. UNCHANGED $\langle rvars, evars, svars, msgs, ovars \rangle \Rightarrow$ UNCHANGED $\langle \overline{svars, rvars, evars} \rangle$

PROOF:Clearly true since either the left-hand side of the expression contains all variables in the spec.

2.2.4. Q.E.D.

2.3. $EnvActions \Rightarrow \overline{EnvActions}$

PROOF: Trivially true, since their definitions are equal.

2.4. Q.E.D.

3. Q.E.D.

□

C Uncoordinated Implementation

C.1 Specification

MODULE *UnCoordLogService*

This module is specifies the log service's Uncoordinated Implementation.

EXTENDS *Naturals*, *FiniteSets*, *Sequences*,
LogServiceConstants,
Consensus

The implementation's constants:
- *MPL* RM's multi-programming level.
- *VoteSort* Function that sorts votes according to the commit counter.

CONSTANTS MPL, *MultiProgrammingLevel*
$VoteSort(\_)$

The environment's variables:
- *badProc* processes that crashed
- *suspect* process X process suspicions.

VARIABLES $badProc$, Environments's
$suspect$ Who suspects whom?

The Resource Managers and processes that might incarnate resource managers' variables:
- *terminatedAt* transactions terminated at rm.
- *terminatingAt* transactions terminating at a process.
- *pid2rm* PID $\mapsto$ RM.
- *instances* PID $\mapsto$ consensus instance to be used.
- *countdown* number of consensus instances yet to be closed.
- *incarnation* status of the incarnation procedure.

VARIABLES $terminatedAt$,
$terminatingAt$,
$pid2rm$,
$instances$,
$countdown$,
$incarnation$

The Transaction Managers's variables:
- *termReq* transactions requested to terminate.
- *part* participants on each transaction, and incarnating process.
- *inst* RM X TID $\mapsto$ IID.
- *nextInst* RM $\mapsto$ the next IID to be attributed to a transaction.

VARIABLES $termReq$,
$part$,
$inst$,
$nextInst$

Defining some aliases.

$rvars \triangleq \langle terminatingAt,\ terminatedAt,\ pid2rm,\ instances,$

$countdown,\ incarnation\rangle$
$tvars \triangleq \langle part,\ termReq,\ nextInst,\ inst\rangle$
$evars \triangleq \langle badProc,\ suspect\rangle$
$ovars \triangleq \langle decision\rangle$
$avars \triangleq \langle rvars,\ tvars,\ evars,\ ovars\rangle$

Types

Vote is extended with commit counter.

$EVotes \triangleq [rm : RM,\ tr : TID \cup \{NoTID\},\ tset : \text{SUBSET } RM,$
$vt : \{\text{“Commit”, “Abort”}\},\ upd : Update,\ cn : Nat]$

Votes to incarnate a resource manager.

$IncarnT \triangleq [vt : \{\text{“Incarnate”}\},\ pid : PID]$

Invariants

Type invariant.

$TypeInvariant \triangleq$
$\wedge\ terminatingAt \in [PID \to \text{SUBSET } TID]$
$\wedge\ terminatedAt \in [RM \to \text{SUBSET } TID]$
$\wedge\ pid2rm \in [PID \to RM \cup \{NoRM\}]$
$\wedge\ instances \in [PID \to Nat]$
$\wedge\ incarnation \in [PID \to \{\text{“NotInc”},$
“Inc1”, “Inc2”, “Inc3”, “Inc4”, “Inc”,
“DeInc”}]
$\wedge\ countdown \in [PID \to Nat]$
$\wedge\ part \in [TID \to \text{UNION } \{[S \to PID] : S \in \text{SUBSET } RM\}]$
$\wedge\ termReq \in \text{SUBSET } TID$
$\wedge\ nextInst \in [RM \to Nat]$
$\wedge\ inst \in [RM \to [TID \to Nat \cup \{NoIID\}]]$
$\wedge\ badProc \in \text{SUBSET } PID$
$\wedge\ suspect \in [PID \to [PID \to \text{BOOLEAN }]]$
$\wedge\ ConsensusTypeInv$

Initial State

$Init \triangleq$
$\wedge\ terminatingAt = [p \in PID \mapsto \{\}]$ No transaction terminating.
$\wedge\ terminatedAt = [r \in RM \mapsto \{\}]$ No transaction terminated .
$\wedge\ pid2rm = [p \in PID \mapsto NoRM]$ No RM is incarnated.
$\wedge\ instances = [p \in PID \mapsto 0]$ No consensus instance started.
$\wedge\ incarnation = [p \in PID \mapsto \text{“NotInc”}]$ No process is recovering.
$\wedge\ countdown = [p \in PID \mapsto MPL - 1]$ No process is recovering.
$\wedge\ part = [t \in TID \mapsto [e \in \{\} \mapsto \{\}]]$ No transaction started.

$\land\ termReq \quad = \{\}$ No termination request issued.
$\land\ nextInst \quad = [r \in RM \mapsto 0]$ No operation executed.
$\land\ inst \quad = [r \in RM \mapsto [t \in TID \mapsto NoIID]]$ No instance used.
$\land\ badProc \quad = \{\}$ No bad process.
$\land\ suspect \quad = [p \in PID \mapsto [q \in PID \mapsto \text{FALSE}]]$ No suspicion.
$\land\ ConsensusInit$

Operators

The pid of the process currently incarnating r.

$rm2pid(r) \triangleq$
IF $\exists\, i \in Nat :$
$\land\ Decide(\langle r,\ i\rangle) \neq NoProposal$ If there is a vote
$\land\ Decide(\langle r,\ i\rangle).vt = \text{“Incarnate”}$ to incarnate r,
THEN CHOOSE $p \in PID :$ choose the process p
$\exists\, i \in Nat :$
$\land\ Decide(\langle r,\ i\rangle) \neq NoProposal$
$\land\ Decide(\langle r,\ i\rangle).vt = \text{“Incarnate”}$ who voted for last.
$\land\ Decide(\langle r,\ i\rangle).pid = p$
$\land \neg\exists\, j \in Nat :\ \land j > i$
$\land\ Decide(\langle r,\ j\rangle) \neq NoProposal$
$\land\ Decide(\langle r,\ j\rangle).vt = \text{“Incarnate”}$
ELSE $NoPID$ Choose $NoPID$ otherwise.

Operator $Outcome(r,\ t)$ gives the termination status of transaction t, considering the decisions learned by r.

$Outcome(r,\ t) \triangleq$
IF $\land\ Decide(\langle r,\ inst[r][t]\rangle) \neq NoProposal$ If r's vote
$\land\ \lor\ Decide(\langle r,\ inst[r][t]\rangle).vt = \text{“Incarnate”}$ was to change incarnation
$\lor\ Decide(\langle r,\ inst[r][t]\rangle).vt = \text{“Abort”}$ or to $Abort$
$\lor\ \land\ Decide(\langle r,\ inst[r][t]\rangle).vt = \text{“Commit”}$ or to commit but
$\land \exists\, or \in Decide(\langle r,\ inst[r][t]\rangle).tset :$ another participant
$\land\ Decide(\langle or,\ inst[or][t]\rangle) \neq NoProposal$ has a voted to
$\land\ \lor\ Decide(\langle or,\ inst[or][t]\rangle).vt = \text{“Incarnate”}$ change incarnation
$\lor\ Decide(\langle r,\ inst[r][t]\rangle).vt = \text{“Abort”}$ or to $Abort$
THEN “Abort” then it is $Abort$.
ELSE IF $\lor\ Decide(\langle r,\ inst[r][t]\rangle) = NoProposal$ If r's vote has not been decided
$\lor\ \land\ Decide(\langle r,\ inst[r][t]\rangle) \neq NoProposal$ or it has
$\land\ Decide(\langle r,\ inst[r][t]\rangle).vt = \text{“Commit”}$ and is $Commit$, but
$\land \exists\, or \in Decide(\langle r,\ inst[r][t]\rangle).tset :$ someone else's
$Decide(\langle or,\ inst[or][t]\rangle) = NoProposal$ has not.
THEN “Undefined” then it is $Undefined$.
ELSE “Commit” Otherwise it is $Commit$.

Actions

Environment

This action crashes a good process.

$$\begin{array}{l} Crash \triangleq \land \exists\, p \in (PID \setminus badProc) : badProc' = badProc \cup \{p\} \\ \qquad \land \text{UNCHANGED } \langle suspect \rangle \end{array}$$

This action changes the suspicion status.

$$\begin{array}{l} ChangeSuspicion \triangleq \land \exists\, p \in (PID \setminus badProc),\ q \in PID : \\ \qquad\qquad \land p \neq q \\ \qquad\qquad \land suspect' = [suspect \text{ EXCEPT } ![p][q] = \neg @] \\ \qquad\qquad \land \text{UNCHANGED } \langle badProc \rangle \end{array}$$

EnvActions is the disjunction of the environment's actions.
- process crash.
- change suspicions.

$$\begin{array}{l} EnvActions \triangleq \land Crash \lor ChangeSuspicion \\ \qquad\qquad \land \text{UNCHANGED } \langle tvars,\ rvars,\ ovars \rangle \end{array}$$

Transaction Manager

Adds a resource manager to a non-terminated transaction.

The transaction manager will only succeed in the adding a resource manager r to a transaction t if r was not crashed by the time it was contacted, and replied to the operation request.

$$\begin{array}{l} AddRM \triangleq \\ \quad \text{LET } processingAt(r) \triangleq \{tr \in TID : \land r \in \text{DOMAIN } part[tr] \\ \qquad\qquad\qquad\qquad\qquad\qquad \land tr \notin terminatedAt[r]\} \\ \qquad Smaller(r) \triangleq \text{LET } instS \triangleq \{inst[r][t] : t \in processingAt(r)\} \\ \qquad\qquad\qquad\quad \text{IN CHOOSE } i \in instS : \forall\, j \in instS : i \leq j \\ \qquad CanProcess(r) \triangleq processingAt(r) \neq \{\} \\ \qquad\qquad\qquad\qquad\qquad \Rightarrow nextInst[r] - Smaller(r) < MPL \\ \quad \text{IN} \quad \land \exists\, t \in (TID \setminus termReq), \qquad t \text{ has not had terminaiton requested.} \\ \qquad\quad r \in \{r \in RM : rm2pid(r) \neq NoPID\} : \qquad r \text{ has been incarnated,} \\ \qquad\qquad \land incarnation[rm2pid(r)] = \text{“Inc”} \\ \qquad\qquad \land rm2pid(r) \notin badProc \qquad \text{and is contactable,} \\ \qquad\qquad \land r \notin \text{DOMAIN } part[t] \qquad \text{and is not a participant of } t \\ \qquad\qquad \land CanProcess(r) \qquad \text{and has free slots.} \\ \qquad\qquad \land part' = [part \text{ EXCEPT } ![t] = \qquad \text{Now it is a participant.} \\ \qquad\qquad\qquad\qquad [e \in \text{DOMAIN } @ \cup \{r\} \mapsto \text{IF } e \in \text{DOMAIN } @ \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{THEN } @[e] \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{ELSE } rm2pid(r)]] \\ \qquad\qquad \land inst' = [inst \text{ EXCEPT } ![r][t] = nextInst[r]] \\ \qquad\qquad \land nextInst' = [nextInst \text{ EXCEPT } ![r] = @ + 1] \\ \qquad\quad \land \text{UNCHANGED } \langle termReq \rangle \end{array}$$

Request the termination of a transaction.

The transaction manager can, at any time, request the termination a transaction it has not requested to terminate before, as long as it has some participant.

$$
\begin{array}{l}
RequestTerm \triangleq \\
\quad \land \exists\, t \in (TID \setminus termReq) : \\
\qquad \land part[t] \neq \langle\rangle \\
\qquad \land termReq' = termReq \cup \{t\} \\
\quad \land \text{UNCHANGED } \langle part,\ nextInst,\ inst\rangle
\end{array}
$$

t has not tried to terminate yet.

The disjunction of Transaction Manager's actions.

- Add a resource manager as a participant.
- Try to terminate a transaction.

$$
\begin{array}{l}
TMActions \triangleq \\
\quad \land \lor AddRM \\
\qquad \lor RequestTerm \\
\quad \land \text{UNCHANGED } \langle rvars,\ evars,\ ovars\rangle
\end{array}
$$

Resource Manager

For a process p to incarnate a resource manager r it must go through several steps.

In the first step, p is marked as incarnating.

$$
\begin{array}{l}
IncStart(p,\ r) \triangleq \\
\quad \land incarnation[p] = \text{“NotInc”} \qquad \text{p is neither incarnating nor trying} \\
\quad \land \lor rm2pid(r) = NoPID \qquad \text{and r is not incarnated} \\
\qquad \lor \land rm2pid(r) \neq NoPID \\
\qquad\quad \land suspect[p][rm2pid(r)] \qquad \text{or its process is supected.} \\
\quad \land pid2rm' = [pid2rm \text{ EXCEPT } ![p] = r] \\
\quad \land incarnation' = [incarnation \text{ EXCEPT } ![p] = \text{“Inc1”}] \\
\quad \land \text{UNCHANGED } \langle terminatingAt,\ terminatedAt,\ instances,\ countdown,\ ovars,\ nextInst\rangle
\end{array}
$$

In the second step p tries to pass an *Incarnate* vote.

$$
\begin{array}{l}
Inc1Step(p,\ r) \triangleq \\
\quad \land incarnation[p] = \text{“Inc1”} \qquad \text{p is in the first loop of incarnate,} \\
\quad \land pid2rm[p] = r \qquad \text{to incarnate r,} \\
\quad \land Propose(\langle r,\ instances[p]\rangle, \\
\qquad\qquad [vt \mapsto \text{“Incarnate”},\ pid \mapsto p]) \qquad \text{and votes to incarnate r.} \\
\quad \land \text{UNCHANGED } \langle terminatingAt,\ terminatedAt,\ pid2rm,\ incarnation,\ instances, \\
\qquad\qquad countdown,\ nextInst\rangle
\end{array}
$$

$$
\begin{array}{l}
Inc1BreakOrLoop(p,\ r) \triangleq \\
\quad \land incarnation[p] = \text{“Inc1”} \qquad \text{p is in the first loop of incarnate,} \\
\quad \land pid2rm[p] = r \qquad \text{to incarnate r,} \\
\quad \land Decide(\langle r,\ instances[p]\rangle) \neq NoProposal \qquad \text{something got decided} \\
\quad \land \lor \land Decide(\langle r,\ instances[p]\rangle).vt = \text{“Incarnate”} \qquad \text{it was an incarnation request} \\
\qquad \land \lor \land Decide(\langle r,\ instances[p]\rangle).pid = p \qquad \text{and it was its own} \\
\qquad\quad \land incarnation' = [incarnation \text{ EXCEPT } ![p] = \text{“Inc2”}] \qquad \text{so move on.} \\
\qquad \lor \land Decide(\langle r,\ instances[p]\rangle).pid \neq p \qquad \text{or it's someone else's.} \\
\qquad\quad \land \text{UNCHANGED } \langle incarnation\rangle \\
\quad \lor \land Decide(\langle r,\ instances[p]\rangle).vt \in \{\text{“Commit”},\ \text{“Abort”}\} \qquad \text{or just something else.} \\
\qquad \land \text{UNCHANGED } \langle incarnation\rangle
\end{array}
$$

$\land\ instances' = [instances \text{ EXCEPT } ![p] = @ + 1]$ anyway, move to the next instance.
$\land$ UNCHANGED $\langle terminatingAt,\ terminatedAt,\ pid2rm,\ countdown,\ ovars,\ nextInst\rangle$

In the second step, p makes sure that MPL possibly open instances are closed.

$Inc2Step(p,\ r)\ \triangleq$
$\land\ incarnation[p] = \text{“Inc2”}$ p is in the second loop of incarnate,
$\land\ pid2rm[p] = r$ to incarnate r,
$\land\ countdown[p] > 0$ and there is some instance to close,
$\land\ Propose(\langle r,\ instances[p]\rangle,$
$[rm \mapsto r,\ tr \mapsto NoTID,\ tset \mapsto \{r\},\ vt \mapsto \text{“Abort”},$
$upd \mapsto \{\},\ cn \mapsto 0])$ so propose.
$\land$ UNCHANGED $\langle terminatingAt,\ terminatedAt,\ pid2rm,\ incarnation,\ instances,$
$countdown,\ nextInst\rangle$

$Inc2BreakOrLoop(p,\ r)\ \triangleq$
$\land\ incarnation[p] = \text{“Inc2”}$ p is in the $2nd$ step of incarnate,
$\land\ pid2rm[p] = r$ to incarnate r,
$\land\ \lor\ \land\ countdown[p] = 0$ and is done closing instances,
$\land\ incarnation' = [incarnation \text{ EXCEPT } ![p] = \text{“Inc3”}]$ and will move to the $3rd$ step.
$\land$ UNCHANGED $\langle countdown,\ instances\rangle$
$\lor\ \land\ countdown[p] > 0$ but it is not done closing instances
$\land\ Decide(\langle r,\ instances[p]\rangle) \neq NoProposal$ so it checks what has been decided
$\land\ \lor\ \land\ Decide(\langle r,\ instances[p]\rangle).vt = \text{“Incarnate”}$ and sees a new incarnation of r
$\land\ incarnation' = [incarnation \text{ EXCEPT } ![p] = \text{“DeInc”}]$ and gives up.
$\land$ UNCHANGED $\langle instances,\ countdown\rangle$
$\lor\ \land\ Decide(\langle r,\ instances[p]\rangle).vt \neq \text{“Incarnate”}$ and sees some vote,
$\land\ countdown' = [countdown \text{ EXCEPT } ![p] = @ - 1]$ decreases the count of instances
$\land\ instances' = [instances \text{ EXCEPT } ![p] = @ + 1]$ and move to the $2nd$ step.
$\land$ UNCHANGED $\langle incarnation\rangle$
$\land$ UNCHANGED $\langle terminatingAt,\ terminatedAt,\ pid2rm,\ ovars,\ nextInst\rangle$

In the $3rd$ step, p makes sure to know the outcome of all transactions that possibly committed.

$Inc3Step(p,\ r)\ \triangleq$
$\land\ incarnation[p] = \text{“Inc3”}$ p is in the $3rd$ step of incarnate,
$\land\ pid2rm[p] = r$ to incarnate r,
$\land\ \lor \exists\, i \in 0\,..\,instances[p] - 1 :$
$\land\ Decide(\langle r,\ i\rangle) \neq NoProposal$ There is a vote
$\land\ Decide(\langle r,\ i\rangle).vt = \text{“Commit”}$ to commit a transaction
$\land\ Outcome(r,\ Decide(\langle r,\ i\rangle).tr) = \text{“Undefined”}$ that did not finish,
$\land \exists\, or \in Decide(\langle r,\ i\rangle).tset :$ because some participant has not voted
$Propose(\langle r,\ inst[or][Decide(\langle r,\ i\rangle).tr]\rangle,$
$[rm \mapsto or,\ tr \mapsto Decide(\langle r,\ i\rangle).tr,$
$tset \mapsto Decide(\langle r,\ i\rangle).tset,$
$vt \mapsto \text{“Abort”},\ upd \mapsto \{\},\ cn \mapsto 0])$ so vote to $Abort$ for it.
$\land$ UNCHANGED $\langle incarnation\rangle$
$\lor\ \land\ incarnation' = [incarnation \text{ EXCEPT } ![p] = \text{“Inc4”}]$ Move to the last step.
$\land$ UNCHANGED $\langle ovars\rangle$

$\land$ UNCHANGED $\langle terminatingAt,\ terminatedAt,\ pid2rm,\ instances,$
$countdown,\ nextInst\rangle$

$IncEnd(p,\ r)\ \triangleq$
LET $CVrm\ \triangleq\ \{Decide(\langle r,\ i\rangle)$: Votes for instances that committed.
$i \in \{j \in 1\ ..\ instances[p] - 1 :$
$\land\ Decide(\langle r,\ j\rangle).vt =$ "Commit"
$\land\ Outcome(r,\ Decide(\langle r,\ j\rangle).tr) =$ "Commit"$\}\}$

$ApplyUpdates(u)\ \triangleq$
LET $apply[i \in 1\ ..\ Len(u)]\ \triangleq$
IF $i = Len(u)$ THEN $ApplyUpdate(u[i])$
ELSE $ApplyUpdate(u[i]) \land apply[i+1]$

IN IF $u = \langle\rangle$ THEN TRUE
ELSE $apply[1]$

IN $\land\ incarnation[p] =$ "Inc4" p is on the last step of incarnate
$\land\ pid2rm[p] = r$ to incarnate r,
$\land\ incarnation' = [incarnation$ EXCEPT $![p] =$ "Inc"] and is moving forward;
$\land\ ApplyUpdates(\ VoteSort(CVrm))$ it is applying the updates,
$\land\ nextInst' = [nextInst$ EXCEPT $![r] = instances[p]]$ and setting the first instance accordingly.
$\land$ UNCHANGED $\langle terminatingAt,\ terminatedAt,\ pid2rm,\ instances,$
$countdown,\ ovars\rangle$

Desincarnate is executed when p sees that another process has incarnated r.

$Desincarnate(p,\ r)\ \triangleq$
$\land\ incarnation[p] =$ "Inc" p is incarnating
$\land\ pid2rm[p] = r$ r
$\land\ \exists\, i \in instances[p]\ ..\ nextInst[r]$: but there is a new vote
$\land\ Decide(\langle r,\ i\rangle) \neq NoProposal$
$\land\ Decide(\langle r,\ i\rangle).vt =$ "Incarnate" to incarnate r
$\land\ Decide(\langle r,\ i\rangle).pid \neq p$ from another process.
$\land\ incarnation' = [incarnation$ EXCEPT $![p] =$ "DeInc"]
$\land$ UNCHANGED $\langle terminatingAt,\ terminatedAt,\ pid2rm,\ instances,$
$countdown,\ nextInst,\ ovars\rangle$

IncarnateStub is the disjunction of the steps needed to incarnate or desincarnate a resource manager.
It is a "stub" to the abstract log service's *Incarnate* action.

$IncarnateStub\ \triangleq$
$\exists\, p \in PID \setminus badProc,$ p is good
$r \in RM$:
$\lor\ IncStart(p,\ r)$ and wants to incarnate
$\lor\ Inc1Step(p,\ r) \lor Inc1BreakOrLoop(p,\ r)$
$\lor\ Inc2Step(p,\ r) \lor Inc2BreakOrLoop(p,\ r)$
$\lor\ Inc3Step(p,\ r)$
$\lor\ IncEnd(p,\ r)$
$\lor\ Desincarnate(p,\ r)$ or desincarnate r.

Executed by r to issue its own vote for transaction t.

$VoteForMyself(r,\ t) \triangleq$
LET $CommittedCounter \triangleq Cardinality(\{i \in Nat : Decide(\langle r,\ i\rangle) \neq NoProposal\})$
IN $\land\ t \in (termReq \setminus (terminatingAt[part[t][r]] \cup terminatedAt[r]))$
$\land\ terminatingAt' = [terminatingAt$ EXCEPT $![part[t][r]] = @ \cup \{t\}]$ Try to terminate
$\land\ \lor\ Propose(\langle r,\ inst[r][t]\rangle,\ [rm \mapsto r,\ tr \mapsto t,\ tset \mapsto$ DOMAIN $part[t],$
$vt \mapsto$ "Commit", $upd \mapsto GetUpdate(r,\ t),$ voting commit.
$cn \mapsto CommittedCounter])$
$\lor\ Propose(\langle r,\ inst[r][t]\rangle,\ [rm \mapsto r,\ tr \mapsto t,\ tset \mapsto$ DOMAIN $part[t],$
$vt \mapsto$ "Abort", $upd \mapsto \{\},\ cn \mapsto 0])$ voting abort.
$\land$ UNCHANGED $\langle terminatedAt,\ instances,\ countdown,\ incarnation\rangle$

Executed by r to issue votes on behalf of other resource managers.

$VoteForOthers(r,\ t) \triangleq$
$\land\ t \in terminatingAt[part[t][r]]$ r tried to terminate t
$\land\ Outcome(r,\ t) =$ "Undefined" but did not succeed yet
$\land \exists\ or \in$ DOMAIN $part[t]$: because some participant did not vote
$\land\ suspect[part[t][r]][part[t][or]]$ * and I don't want to wait
$\land\ Propose(\langle or,\ inst[or][t]\rangle,\ [rm \mapsto or,\ tr \mapsto t,\ tset \mapsto$ DOMAIN $part[t],$
$vt \mapsto$ "Abort", $upd \mapsto \{\},\ cn \mapsto 0])$ so vote *Abort* on its behalf.
$\land$ UNCHANGED $\langle terminatingAt,\ terminatedAt,\ instances,\ countdown,\ incarnation\rangle$

Executed by r to learn that transaction t terminated.

$Learn(r,\ t) \triangleq$
$\land\ Outcome(r,\ t) \neq$ "Undefined" The instance is finished
$\land\ t \in terminatingAt[part[t][r]] \setminus terminatedAt[r]$ but r did not know
$\land\ terminatingAt' = [terminatingAt$ EXCEPT $![part[t][r]] = @ \setminus \{t\}]$ so take note.
$\land\ terminatedAt' = [terminatedAt$ EXCEPT $![r] = @ \cup \{t\}]$
$\land$ UNCHANGED $\langle instances,\ countdown,\ incarnation,\ ovars\rangle$

TerminateStub is executed by a resource manager to step towards the termination of some transaction.

It is a "stub" to the abstract log service's Terminate action.

$TerminateStub \triangleq$
$\land \exists\ r \in RM,\ t \in TID :$
$\land\ r \in$ DOMAIN $part[t]$ r is a participant of t.
$\land\ part[t][r] \notin badProc$ The process is still alive.
$\land\ incarnation[part[t][r]] =$ "Inc" and sees itself as rm's incarnation.
$\land\ \lor\ VoteForMyself(r,\ t)$ first attempt to terminate t.
$\lor\ VoteForOthers(r,\ t)$ other attempts
$\lor\ Learn(r,\ t)$ Learn that it was decided.
$\land$ UNCHANGED $\langle pid2rm,\ nextInst\rangle$

The disjunction of resource managers actions.

$RMActions \triangleq\ \land\ IncarnateStub \lor TerminateStub$
$\land$ UNCHANGED $\langle part,\ termReq,\ inst,\ evars\rangle$

Specification

$Next \triangleq \lor RMActions$ Implement $RMActions$

$\lor TMActions$ Implement $TMActions$

$\lor EnvActions$ Implement $EnvActions$

$Spec \triangleq Init \land \Box[Next]_{\langle avars \rangle}$

Refinement Mapping

$rm\_pid2rm \triangleq [p \in PID \mapsto \text{IF } incarnation[p] = \text{“Inc”} \text{ THEN } pid2rm[p] \text{ ELSE } NoRM]$

$rm\_rm2pid \triangleq [r \in RM \mapsto rm2pid(r)]$

$rm\_LastConcSet \triangleq$

LET $LastToCommitAt(r) \triangleq$

CHOOSE $t \in TID :$ Last to commit as some resource manager.

$\land Outcome(r, t) = \text{“Commit”}$

$\land \neg\exists\, ot \in TID :$

$\land Outcome(r, ot) = \text{“Commit”}$

$\land Decide(\langle r, inst[r][ot]\rangle).cn > Decide(\langle r, inst[r][t]\rangle).cn$

$LastToCommit \triangleq$ Last to commit on each resource managers.

$\{LastToCommitAt(r) : r \in \{$

$or \in RM :$

$\exists\, t \in TID :$

$Outcome(r, t) = \text{“Commit”}\}\}$

$DistLastToCommit \triangleq$

$\{t \in LastToCommit :$ Last to commit globally.

$\neg\exists\, r \in RM :$

$\exists\, ot \in TID :$

$\land Outcome(r, ot) = \text{“Commit”}$

$\land Decide(\langle r, inst[r][ot]\rangle).cn > Decide(\langle r, inst[r][t]\rangle).cn$

$\}$

$ConcToLast \triangleq \{ot \in TID :$ Concurrent to the last to commit.

$\exists\, r \in RM, t \in DistLastToCommit :$

$\land Decide(r, inst[r][ot]).cn = Decide(r, inst[r][t]).cn\}$

IN $ConcToLast$

$rm\_terminatedAt \triangleq$

LET $IncInstOf(p) \triangleq$

CHOOSE $i \in IID : \land i[1] = pid2rm[p]$ The instance that incarnated p

$\land Decide(i).vt = \text{“Incarnate”}$

$\land Decide(i).pid = p$

```
    DeIncInstOf(p) ≜
        CHOOSE i ∈ IID : ∧ i[1] = pid2rm[p]          The instance that desincarnated p
                         ∧ i[2] > IncInstOf(p)[2]
                         ∧ Decide(i).vt = "Incarnate"
                         ∧ ¬∃ j ∈ IID : ∧ j[2] > IncInstOf(p)[2]
                                        ∧ j[2] < i[2]
                                        ∧ Decide(j).vt = "Incarnate"

    FirstInstOf(p) ≜ ⟨pid2rm[p], IncInsOf(p)[2] + MPL⟩          p's first instance.

    LastInstOf(p) ≜ ⟨pid2rm[p], DeIncInsOf(p)[2] + MPL − 1⟩     p's last instance.

    TermTransOf(p) ≜
        {t : ∧ Outcome(pid2rm[p], t) = "Commit"
             ∧ inst[pid2rm[p]][t][2] > FirstInstOf(p)[2]
             ∧ ∃ i ∈ IID : ∧ i[1] = pid2rm[p]          The instance that desincarnated p
             ∧ i[2] > IncInstOf(p)[2]
             ∧ Decide(i).vt = "Incarnate"
                 ⇒ inst[pid2rm[p]][t][2] ≤ LastInstOf(p)[2]
        }
IN  [p ∈ PID ↦ IF pid2rm[p] = NoRM THEN {} ELSE TermTransOf(p)]


rm_vHist ≜
    LET TheVote(i) ≜
        IF Decide(i).vt ∈ {"Abort", "Incarnate"}
        THEN [rm ↦ i.[i], tr ↦ CHOOSE t ∈ TID : inst[i[1]][t] = i,
              tset ↦ DOMAIN part[t], vt ↦ "Abort", upd ↦ {}]
        ELSE LET v ≜ Decide(i)
             IN  [rm ↦ v.rm, tr ↦ v.tr, tset ↦ v.tset, vt ↦ v.vt, upd ↦ v.upd]

    IN  {TheVote(i) : i ∈ {oi ∈ IID : ∧ ∃ t ∈ TID : inst[i[1]][t] ≠ NoIID
                                     ∧ Decide(i) ≠ NoProposal}}


rm_tHist ≜ CHOOSE s ∈ Seq({t ∈ TID : ∃ r ∈ RM : Outcome(r, ot) = "Commit"}) :
             ∀ i, j ∈ DOMAIN s :
               ∀ r ∈ (DOMAIN part[i] ∩ DOMAIN part[j]) :
                 Decide(⟨r, inst[r][i]⟩).cn < Decide(⟨r, inst[r][j]⟩).cn
                  ⇒ i < j
```

C.2 Implementation Proof

The refinement mapping is given below.

$\overline{pid2rm} \triangleq rm\_pid2rm$

$\overline{rm2pid} \triangleq rm\_rm2pid$

$\overline{LastConcSet} \triangleq rm\_LastConcSet$

$\overline{terminatedAt} \triangleq rm\_terminatedAt$

$\overline{vHist} \triangleq rm\_vHist$

$\overline{tHist} \triangleq rm\_tHist$

Proposition C.1 $Spec \Rightarrow \overline{Spec}$

1. ASSUME: $Init$
   PROVE: $\overline{Init}$
   PROOF: Variables $terminatingAt$, $part$, $termReq$, $badProc$ and $suspect$ are initialized in $Init$ exactly as specified in $\overline{Init}$. By the refinement mapping, $\overline{LastConcSet}$ is initially the empty set and $\overline{terminatedAt}$ maps each process to the empty set, conforming with their initialization in $\overline{Init}$. By the definition of operator $rm2pid$ and because no value has neither been proposed nor decided, $\overline{rm2pid}$ maps from all resource managers to $NoPID$, as specified in $\overline{Init}$. By the refinement mapping and because no has been decided at the initial state, $\overline{vHist} = \{\}$ and $\overline{tHist} = \langle\rangle$, according to $\overline{Init}$.

2. ASSUME: $Next$
   PROVE: $\overline{Next} \vee$ UNCHANGED $\langle\overline{svars}, \overline{tvars}, \overline{rvars}, \overline{evars}\rangle$
   2.1. ASSUME: $RMActions$
   PROVE: $\overline{RMActions}$
   2.1.1. ASSUME: $IncarnateStub \wedge$ UNCHANGED $\langle part, termReq, inst, evars\rangle$
   PROVE: $\Diamond \wedge \overline{Incarnate} \vee$ UNCHANGED $\langle svars, rvars\rangle$
   $\wedge$ UNCHANGED $\langle\overline{tvars}, \overline{evars}\rangle$
   PROOF: The $\overline{Incarnate}$ action is implemented by a series of steps, implemented by actions
   - $IncStart(p, r)$,
   - $Inc1Step(p, r)$,
   - $Inc1BreakOrLoop(p, r)$,
   - $Inc2Step(p, r)$,
   - $Inc2BreakOrLoop(p, r)$,
   - $Inc3Step(p, r)$,
   - $IncEnd(p, r)$, and
   - $Desincarnate(p, r)$,

   where p is a process and r the resource manager p is trying to incarnate. These actions execute in a specific order, according to the state of the variable $incarnation[p]$: $IncStart(p, r)$ requires $incarnation[p] =$ "`NotInc`$''$, and changes it to "`Inc1`$''$. $Inc1Step(p, r)$ requires $incarnation[p] =$ "`Inc1`$''$, and does not change it, possibly being executed until it changes to "`Inc2`". Action $Inc1BreakOrLoop(p, r)$ also has $incarnation[p] =$ "`Inc1`$''$ as a pre-condition, but changes it to "`Inc2`" if some conditions are met. Action $Inc2Step(p, r)$ requires $incarnation[p] =$ "`Inc2`$''$, and executes until it turns to "`Inc3`". $Inc2BreakOrLoop(p, r)$ has $incarnation[p] =$ "`Inc2`$''$ as pre-condition and changes $incarnating[p]$ to "`Inc3`". The change of $incarnating[p]$ from "`Inc3`" to "`Inc4`" is done by action $Inc3Step(p, r)$ and, finally, action $IncEnd(p, r)$ requires $incarnation[p] =$ "`Inc4`$''$ and changes it to "`Inc`", when the incarnation

procedure is over. Once the first step is finished, the process q that previously incarnated r either crashes or eventually learns about p, by the termination property of consensus, and executes action $Desincarnate(q, r)$.

The initial steps in this series serves the purpose of determining the updates executed by previous incarnations of r, ensuring that the consensus instances possibly used by previous incarnations of r are terminated. Because none of the actions, except for the last, changes any of the overlined variables, their execution implies in stuttering steps of $\overline{Spec}$. When the last action, $IncEnd(p, r)$, is executed, an equivalent $\overline{Incarnate}$ step is executed;

The specification does not ensure that $Inc1BreakOrLoop$ will ever succeed in incrementing the variable $instances[p]$, but once it is done, the process incarnating r will crash or eventually learn that it was replaced, since there is a maximum number of consensus instances it can start before learning the outcome of the previous ones (MPL), by the **C-Termination** property of consensus. If the process previously incarnating r is not crashed, then it may proceed executing transactions, and p will eventually learn that it made a mistake and crash itself (although this behavior is not specified, it is clearly correct, as p is allowed to crash at any moment). For the same reason, once $incarnation[p]$ equals "`Inc2`", p will crash or eventually execute the other actions of $IncarnateStub$, learning the updates of all transactions executed by previous incarnations of r. This is performed by simply enforcing the termination of the consensus instances used on those incarnations and by the other resource managers involved in such transactions. Due to the agreement property of consensus, the outcome of each committed transaction is seen by any process that executes the incarnation procedure.

The order p applies these updates is determined by the $VoteSort$ operator, that orders updates according to the cn field contained in their respective votes. Because non-concurrent transactions are guaranteed to have different cn, ordering the updates according to cn ensures that updates of non-concurrent transactions are ordered in the order they committed.

2.1.2. ASSUME: $TerminateStub \land$ UNCHANGED $\langle part, termReq, inst, evars \rangle$
PROVE: $\Diamond \land \overline{Terminate} \lor$ UNCHANGED $\langle \overline{svars}, \overline{rvars} \rangle$
$\land$ UNCHANGED $\langle \overline{tvars}, \overline{evars} \rangle$

PROOF: The first two pre-conditions are the same for both actions. The third pre-condition of $TerminateStub$ is true iff t is in the range of transactions in which r was incarnated by $part[t][r]$ and the action $Desincarnate$ has not been executed by the process. If $Desincarnate$ is not enabled, then the incarnating process cannot have executed action $IncEnd$ and the $part[t][r]$ is still incarnating r. Otherwise, although this pre-condition is true, the action will not succeed adding a vote to $vHist$, since the consensus instance it would use has been already decided by the second step of the incarnation of process of the process starting to incarnate r.

The action $VoteForMyself(r, t)$ has equivalent pre-conditions in both specifications and similar first post-condition. By the consensus properties, the second post-condition ensures that r's vote for t is added to $\overline{vHist}$, if $part[t][r]$ is not suspected of having crashed, as specified in $\overline{Vote}$. Otherwise it implies a stuttering step of $\overline{Spec}$. By the definition of $\overline{vHist}$, deciding on a vote implies in adding a vote to $\overline{vHist}$. If the vote leads to the "Commit" of a transaction, then $\overline{LastConcSet}$ becomes the set of transactions for which votes have been issued with the same commit counter as those for the transaction just committed.

As the case for $VoteForMyself$ and $\overline{VoteForMyself}$, $VoteForOthers$ has conditions that are equivalent to those in $\overline{VoteForOthers}$. By the same argumentation as in the previous paragraph, the execution of a $Propose$ ensures that either a vote for the other resource manager will be added to $vHist$, if $part[t][r]$ does not crash, or it will imply in a stuttering step of $\overline{Spec}$.

A $Learn$ step clearly implies a $\overline{Learn}$ step, since all conditions for the latter are also stated for the first.

2.1.3. Q.E.D.

2.2. $TMActions \Rightarrow \overline{TMActions}$

2.2.1. $AddRM \Rightarrow \overline{AddRM}$

PROOF: Trivially true, since all of $\overline{AddRM}$ pre and post-conditions are also conditions for $AddRM$.

2.2.2. $RequestTerm \Rightarrow \overline{RequestTerm}$

PROOF: Trivially true, since their definitions are equal.

2.3. $EnvActions \Rightarrow \overline{EnvActions}$

PROOF: Trivially true, since their definitions are equal.

2.4. Q.E.D.

3. Q.E.D.

□