A Highly Available Log Service for Transaction Termination*

Lásaro Camargos
University of Campinas (Unicamp)
Brazil

Marcin Wieloch
University of Lugano (USI)
Switzerland

Fernando Pedone
University of Lugano (USI)
Switzerland

Edmundo Madeira
University of Campinas (Unicamp)
Brazil

Abstract

Distributed transaction processing hinges on enforcing agreement among the involved resource managers on whether to commit or abort transactions (atomicity) and on making their updates permanent (durability). This paper introduces a log service *which abstracts these tasks. The service logs commit and abort votes as well as the updates performed by each resource manager. Based on the votes, the log service outputs the transaction's outcome. The service also totally orders non-concurrent transactions and makes the sequence of updates performed by each resource manager available as a means to consistently recover resource managers without relying on their local state. Besides the specification, we overview two highly available implementations of this service and present an experimental performance evaluation.*

1. Introduction

The need to atomicaly commit transactions in distributed management systems is recurrent. Briefly, to terminate a distributed transaction, each participating resource manager votes to either commit or abort it. The transaction's outcome is then determined based on these votes: commit, if all resource managers vote to commit the transaction, or abort, otherwise. If the vote from each participant is always necessary, the procedure may block in the absence of some resource manager. To avoid this scenario, a weaker version of atomic commitment allows the outcome to be abort if some participant is suspected to have failed. Commit will be guaranteed only if all participants vote to commit the transaction and none is suspected. This weaker problem is known as *non-blocking atomic commitment*.

In a recent paper, Gray and Lamport have introduced Paxos Commit (PC) [9], a non-blocking atomic commitment protocol in which each vote is cast using an instance of the Paxos consensus algorithm [12]. To handle failures, resource managers conservatively vote abort on behalf of (supposedly) crashed resource managers by proposing ABORT in their consensus instances. Consensus ensures that all involved parts agree on each single vote in spite of possibly different proposals. PC has the same expected latency as the *Two-Phase Commit* (2PC) protocol. In fact, it is a generalization of 2PC that tolerates failures both of resource managers and of the transaction manager, the process that triggers and coordinates the protocol. Compared to *Three-Phase Commit* (3PC) [1], PC has fewer communication steps and is simpler in case of a coordinator failure.

Besides ensuring atomicity, a transaction termination protocol should also guarantee the durability of committed transactions. Once committed, the changes made by a transaction should not be forgotten despite failures. In conventional protocols this is achieved by having each resource manager store its updates on a local stable media before voting. Should a resource manager fail, it can, at recovery time, read the updates from the local storage and replay them to recover its previous state. A drawback of this approach is that it couples the availability of the resource manager with the availability of the server hosting it. In a clustered environment, for example, one could recover a resource manager on a different node, should its current host fail. But obviously, this can only be done if the state of the resource manager is not stored on the crashed node. PC suggests that information critical to transaction termination should not reside at the transaction participants, but at a separate entity, such as the acceptors—the processes agreeing on the decision in the Paxos consensus protocol— whose availability can be easily tuned.

This paper proposes abstracting transaction termination in terms of a *log service*. The log service has a very simple interface, through which resource managers vote and receive the outcome of transactions. In addition to storing the participant votes, the service may be also used to store transaction updates. As a consequence, crashed resource managers can

*The work presented in this paper has been partially funded by the SNSF, Switzerland (project #200020-115879).

be restarted on any operational node, relying only on the log service to recover their prior state. The power of the service is reflected on its simplicity of use: once the termination is triggered, transaction participants submit their votes to the log service and wait for the outcome. Should a resource manager be suspected of having crashed, others can submit abort votes on its behalf to the log service. If multiple votes are received for the same resource manager, the service will ignore all but the first vote received to determine the transaction's outcome, properly ensuring consistency.

Defining transaction termination in terms of our log service has two advantages. First, transaction termination becomes oblivious to particularities of the system, taken care of or explored by the log service implementation in a transparent way. For example, the service could be implemented using message passing, as we illustrate in this paper, or shared memory, e.g., in a multiprocessor environment. Moreover, if enough nodes can be relied upon not to crash simultaneously, then the service could be implemented in main memory only, removing disk access times from the termination of transactions. Second, the overall availability of resource managers is improved by using a highly available implementation of the log service. As mentioned earlier, the service can be used to migrate crashed or slow resource managers to functional and more dependable hosts. As a consequence, resource managers may choose to asynchronously store their state locally for later recovery or rely solely on the state kept at the log service.

We have designed two implementations of the log service, both relying on consensus to provide high availability, but in different ways. In the first, *uncoordinated*, voting is completely distributed (this approach abstracts PC). In the second, *coordinated*, voting is centralized, managed by a coordinating process. This difference has performance implications on both the termination of transactions and on the recovery of resource managers. The two approaches abstract the tradeoff "message complexity versus number of communication steps" between atomic commit protocols. Our coordinated implementation has linear message complexity, similarly to 3PC, but needs 5 steps to terminate a transaction; the uncoordinated approach, similarly to PC and other proposals (e.g., [10, 11]), reduces the number of steps to 3, at the expense of a quadratic message complexity. Due to space constraints, we just overview each approach in this paper. For an in-depth description, the reader is referred to the extended version of this paper [3]. Our experiments show that the coordinated solution outperforms the uncoordinated one by 8x when transactions are short, what typically happens in performance-critical systems. The performance gain is mainly due to batching of concurrently issued votes by the coordinator, saving on network and disk utilization.

In summary, this paper makes three contributions: First, it introduces and specifies the log service abstraction for transaction termination. Second, it presents two highly available implementations of the service, namely coordinated and uncoordinated. Third, it compares both approaches analytically and experimentally to previous work in the area and shows that reducing the number of communication steps at the expense of increasing the message complexity not always leads to better performance.

2. Problem statement

A *resource manager* (RM) is the owner of some resource that can be read or written, such as a file, a region of memory, or a table in a relational database. We define RMs not as processes, but as roles that can be "incarnated" by different processes at different points in time. We assume the availability of infinitely many processes and, therefore, that there is always a process to incarnate an RM. This model allows an RM to be incarnated at an operational host, should its previous host crash, without waiting for the crashed node to recover. Hence, RMs have crash-recovery failure pattern.

A distributed transaction is a partially ordered set of read and write operations executed by RMs on their resources. An RM is a *participant* of transactions for which it has been requested to execute operations. Each transaction is managed by a *transaction manager* (TM), another role in our model. To terminate a transaction, the TM asks its participants to agree on committing or aborting the transaction through a non-blocking atomic commit (NBAC) protocol. An RM can only vote to commit a transaction after receiving a request from the TM along with the complete list of participants of the transaction. Abort votes, however, may be sent by RMs at any time, even before receiving the request and the participant's list.

Formally, NBAC is defined by the following properties.

Validity If a participant decides to commit a transaction, then all participants voted to commit the transaction.

Agreement No two participants decide differently.

Nontriviality If all participants vote to commit the transaction and none is suspected of failing throughout the execution of the protocol, then the decision is commit.

Termination All non-faulty participants eventually decide.

Nontriviality implies that RMs may be "suspected" of failing. The only assumption we make about failure detection is that if an RM fails (actually, the process incarnating it), then it will eventually be suspected by the other processes.[1] Hence, transactions may be unnecessarily aborted due to incorrect failure suspicions. TM failures are handled with RM unilateral aborts.

NBAC defines proper transaction termination but not durability, *i.e.*, making the effects of committed transaction last in spite of RM crashes. We define *durability* as follows [14]:

[1]This property is similar to the *completeness* property of Chandra and Toueg's failure detectors [5].

Durability After a transaction commits, its changes to the database persist despite system failures.

Durability can be ensured, for example, by reinitializing the database and replaying the updates of committed transaction in the commit order. RMs can store updates on a local stable storage, as traditionally done in database systems, or on an external and possibly replicated media to improve availability.

3. The Log Service

In this section we specify the log service and show how resource managers use it to atomically terminate transactions and to recover crashed resource managers. In Sections 4 and 5 we show how these behaviors can be implemented in a shared-nothing asynchronous distributed system. We start by explaining the formalism used in the specifications.

3.1. Terminology and notation

We describe our algorithms in terms of actions and operators as the following:

$\text{IF}(a, b, c) \triangleq$ **if** a **then** b **else** c

$\text{ADDVALTOB}(val) \triangleq B \leftarrow B + val$

According to these definitions, the operator $\text{IF}(a, b, c)$ evaluates to b if a is true and to c otherwise. Action $\text{ADDVALTOB}(val)$ increments the variable B by the value val. We use **let** and **in** to limit the scope of a definition for conciseness. In the expression,

let $\text{SUM}(a, b) \triangleq a + b$ **in** $\text{SUM}(\text{SUM}(1, 2), \text{SUM}(3, 4))$

for example, $\text{SUM}(a, b)$ is defined as $a + b$ within the scope of $\text{SUM}(\text{SUM}(1, 2), \text{SUM}(3, 4))$.

For simplicity we assume that, except for action TERMINATE of Algorithm 1, all actions presented in the paper are performed atomically. In [3] we give and extended specification with atomic actions only.

When comparing sequences in the algorithm, the symbol "_" matches any value. For example, $\langle a, b\rangle = \langle \_, b\rangle$ and $\langle \_, b\rangle = \langle c, b\rangle$ even though c and a are different. The length of sequence s is given by Len(s), and its i-th element by $s[i]$. $a <_s b$ (resp. $>_s$) holds iff (i)a and b happen only once in s and (ii)$s[i] = a$ and $s[j] = b$ implies $i < j$ (resp. $i > j$). $s \bullet e$ is defined as the sequence s appended by the 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[Len(ss)], (e.g., $\langle\{a, b\}, \{c\}\rangle \oplus d = \langle\{a, b\}, \{c, d\}\rangle$).

3.2. The log service specification

We now present the log service specification, which may be seen as a single process accessed by the RMs through remote procedure calls. The service manipulates the following five data structures:

$\mathcal{V}$ The set of all votes received by the service. Votes are sequences of the form $\langle rm, t, tset, vote, update\rangle$, read as "$rm$ voted $vote$ on transaction t, with updates $update$, if any (*i.e.*, the empty set, if $vote$ = ABORT)". $tset$ is the possibly incomplete list of RMs involved in t. Initially, $\mathcal{V} = \{\}$.

$\mathcal{T}$ The partially ordered set $(S, \preceq)$, where S is the set of committed transactions and $\preceq$ is a partial order on S such that for any two transactions $t_1, t_2 \in S$ such that their commit are not seen as overlapping by the log service, then $t_1 \preceq t_2$. We say that such transactions are "non-concurrent". 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 $t \in \mathcal{T}[i]$ and $u \in \mathcal{T}[j]$, $t \preceq u$. Initially, $\mathcal{T} = \langle\rangle$.

$\mathcal{C}$ The set of committing transactions, that is, the set of non-terminated transactions for which votes have been issued. Initially, $\mathcal{C} = \{\}$.

$LastC$ The subset of $\mathcal{C}$ with transactions known to be concurrent with the last committed transaction, *i.e.*, their termination was concurrent. Initially, $LastC = \{\}$.

$\mathcal{R}$ A map from RMs to the processes currently "incarnating" them. An RM is mapped to $\perp$ (not a valid process) if it has never been incarnated. Initially, all RMs are mapped to $\perp$.

The first part of Algorithm 1 (lines 1–40) initializes the data structures and defines the operators OUTCOME, ISINVOLVED, and UPDATES, and the action VOTE.

OUTCOME(t) evaluates to the outcome of transaction t: ABORT, if at least one vote for t is ABORT; COMMIT, if all votes are present and equal COMMIT; and UNDEFINED, if all known votes for t are COMMIT but there are missing votes, which could turn out to be of either kind.

ISINVOLVED(t, rm) evaluates to TRUE if rm is a participant of t, *i.e.*, if rm is in the list of participants ($tset$) of any known vote for t. Because ABORT votes may not carry the complete participants' list, they are not enough to give negative answers. That is, if only ABORT votes are known and none has rm in the $tset$ field, then ISINVOLVED(t, rm) evaluates to UNKNOWN, but if a COMMIT vote is known and rm is not in the $tset$ field, then it evaluates to FALSE.

VOTE(v) adds vote v to $\mathcal{V}$. A vote is added only if no other vote for the same resource manager and transaction is already in $\mathcal{V}$. This ensures that if conflicting votes are issued by mistake for the same participant, only one vote per participant is considered.

UPDATES(rm) evaluates to the sequence of sets of updates performed by resource manager rm, partially ordered accordingly to $\mathcal{T}$. The evaluation is done by recursively iterating over the sets of committed transactions to find the ones in which rm took part and with which updates.

Algorithm 1 Log service specification

```
1:Initially:
2:   $\mathcal{V} \leftarrow \emptyset$                                   ◁ The history of votes.
3:   $\mathcal{T} \leftarrow \langle\rangle$                              ◁ Sequence of sets of committed trans.
4:   $\mathcal{C} \leftarrow \emptyset$                                   ◁ Set of concurrent trans.
5:   $LastC \leftarrow \emptyset$                                         ◁ Set of trans. concurrent to the last committed.
6:   $\forall r \in RM, \mathcal{R}[r] \leftarrow \bot$                   ◁ Processes incarnating RMs.
7:OUTCOME($t$) $\triangleq$
8:   if $\exists\langle\_, t, \_, \text{ABORT}, \_\rangle \in \mathcal{V}$        ◁ Any ABORTs?
9:     ABORT
10:  else if $\exists\langle\_, t, tset, \text{COMMIT}, \_\rangle \in \mathcal{V}$ :   ◁ All COMMITs?
         $\forall p \in tset : \langle p, t, \_, \text{COMMIT}, \_\rangle \in \mathcal{V}$
11:    COMMIT
12:  else
13:    UNDEFINED                                                          ◁ Not enough commits?
14:ISINVOLVED($t, rm$) $\triangleq$
15:  if $\exists\langle\_, t, tset, \_, \_\rangle \in \mathcal{V} : rm \in tset$      ◁ Is rm in any list?
16:    TRUE
17:  else if $\exists\langle\_, t, \_, v, \_\rangle \in \mathcal{V} : v = \text{COMMIT}$   ◁ Is tset a complete list?
18:    FALSE
19:  else
20:    UNKNOWN                                                            ◁ Not enough information.
21:VOTE($\langle rm, t, tset, vote, update\rangle$) $\triangleq$
22:  if OUTCOME($t$) = UNDEFINED                                         ◁ If t has not terminated yet
23:    $\mathcal{C} \leftarrow \mathcal{C} \cup \{t\}$                    ◁ add it to $\mathcal{C}$.
24:  if $\neg\exists\langle rm, t, \_, \_, \_\rangle \in \mathcal{V}$    ◁ If rm has not voted for t yet
25:    $currState \leftarrow$ OUTCOME($t$)                                ◁ Current state (ABORT or UNDEF.).
26:    $\mathcal{V} \leftarrow \mathcal{V} \cup \{\langle rm, t, tset, vote, update\rangle\}$   ◁ Store vote
27:    if ($currState$ = UNDEFINED) $\wedge$ (OUTCOME($t$) = COMMIT)
28:      if $t \in LastC$                                                 ◁ If t can be added to the last set...
29:        $\mathcal{T} \leftarrow \mathcal{T} \oplus t$                  ◁ ...do it;
30:      else
31:        $\mathcal{T} \leftarrow \mathcal{T} \bullet \{t\}$             ◁ otherwise, add it to a new set
32:        $LastC \leftarrow \mathcal{C}$                                 ◁ with a new $LastC$.
33:      $C \leftarrow C \backslash \{t\}$
34:UPDATES($rm$) $\triangleq$
35:  let UPD($i$) $\triangleq$
36:    if $i = 0$                                                         ◁ Recursion end.
37:      $\langle\rangle$
38:    else                                                               ◁ Get updates and recurse.
39:      UPD($i-1$) $\bullet$
       $\left\{ upd : \langle rm, t, \_, \text{COMMIT}, upd\rangle \in \mathcal{V} : \begin{array}{l} \wedge\, t \in \mathcal{T}[i] \\ \wedge \text{ISINVOLVED}(rm, t) \end{array} \right\}$
40:  in UPD($Len(\mathcal{T})$)                                          ◁ Return updates for rm
41:INCARNATE($rm, pid$) $\triangleq$
42:  $\mathcal{R}[rm] \leftarrow pid$
43:  $updates \leftarrow$ UPDATES($rm$)                                   ◁ Get committed state.
44:  for $i = 1$ to Len($updates$)                                        ◁ For all committed transactions...
45:    apply updates in $updates[i]$                                      ◁ ...apply it to the database.
46:TERMINATE($rm, t, tset, vote, upd$) $\triangleq$
47:  VOTE($\langle rm, t, tset, vote, upd\rangle$)                        ◁ Vote for t and wait...
48:  while OUTCOME($t, rm$) = UNDEFINED
49:    wait (OUTCOME($t, rm$) $\neq$ UNDEFINED) $\vee$ (suspect $r \in tset$)
50:    if suspected $r \in tset$                                          ◁ If suspect someone to have crashed...
51:      VOTE($\langle r, t, tset, Abort, \emptyset\rangle$)              ◁ ... vote on its behalf.
52:  if OUTCOME($t, rm$) = ABORT
53:    abort $t$ in the database
54:  else
55:    apply $upd$ to database
```

3.3. Termination and recovery

The second part of Algorithm 1 (lines 41–55) defines the actions INCARNATE and TERMINATE.

INCARNATE(rm, pid) is used by process pid to incarnate resource manager rm. First, the process sets $\mathcal{R}[rm]$ to its own identifier, pid. Second, it evaluates UPDATES, described above, to get the updates executed by the previous incarnations of rm. Third, pid scans the updates from the first to the last set, applying all updates in one set before those in the next set to its state; updates in the same set need not be ordered. The action ends with pid incarnating rm, with a state equal to the previous incarnation. pid will accept and process new transactions as rm until it crashes or another process incarnates rm (*i.e.*, pid ceases equaling $\mathcal{R}[rm]$). If more than one process try to incarnate rm, a quick succession of incarnations will happen, but only one will remain incarnated.

TERMINATE is used by resource managers (the processes incarnating them) to trigger or join the termination of transactions in which they participate. To terminate t, a resource manager rm executes TERMINATE($\langle rm, t, tset, vote, upd\rangle$), where $tset$ is the set of resource managers known by rm to be participants of t, and $vote$ is either ABORT or COMMIT, depending on whether rm is willing to commit the transaction or not. If $vote$ equals COMMIT, then upd contains the updates performed by rm in t. If $vote$ equals ABORT, upd equals $\emptyset$.

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 release the related locks, if locking is used. If rm learns that t aborted, it locally aborts the transaction. Updates are made durable by the log service. We assume that the execution of TERMINATE is not necessarily an atomic operation, allowing multiple resource managers to vote in parallel and the same resource manager to terminate distinct transactions in parallel, if its scheduling model allows it.

Termination using our log service provably solves NBAC, and recovery through the INCARNATE action provably ensures the durability property. Due to space restrictions, the reader is referred to [3] for detailed proof of correctness.

4. From specifications to implementations

So far we have described the log service abstractly in terms of variables and atomic actions triggered by processes incarnating resource managers. We now describe the building blocks we used in our implementations of the service. We then follow with a description of the implementations.

4.1. Building blocks

Communications links We assume that processes communicate by message passing using quasi-reliable communication channels: channels ensuring that (a)if neither the sender nor the receiver of a message crashes, then the message is eventually delivered, and (b)messages are neither corrupted nor duplicated.

Leader-election oracle We assume that processes have access to a leader-election oracle like Ω [4]. The oracle guarantees that eventually all participants 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, "forever" is reduced to "long enough to accomplish some useful computation", *e.g.*, deciding on a transaction's outcome.

Consensus Processes also rely on a consensus black box to ensure consistency. In the consensus problem, a set of processes tries to agree on a common value. As in [12], we define the consensus problem in terms of the three roles played by processes: *proposers* propose values to be agreed; *acceptors* interact to choose a proposed value; and *learners* identify that a value was decided. A process can play any number of these roles. Hence, a process 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 acceptors. 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.

4.2. Coordinated implementation

The coordinated implementation is named after a coordinator process that serves as the interface for the log service to RMs. Instead of simply accessing the service, RMs exchange messages with the coordinator to implement each action in the service's specification. To vote in a transactions, RMs send a $vote$ message to the coordinator, which ensures that votes become durable. It does so by proposing them in a sequence of consensus instances until they are decided in some instance, at which point they are durable. The coordinator analyzes durable votes to determine transactions' outcomes and inform the RMs, which then complete the implementation of action VOTE. To amortize the cost of terminating each transaction, the coordinator batches as many votes as possible in the proposal of each consensus instance. If an RM has enough local information to implement some action without waiting for the coordinator's reply, then it does so to improve performance. One such example is the ability to abort a transaction locally when the RMs' vote itself is ABORT.

Multiple processes capable of coordinating the implementation run in parallel and, along with the RMs, they use the leader-election oracle to elect the effective coordinator. The elected coordinator is the one to which RMs send their votes and to which non-elected coordinators forward all votes that they may receive from mistaken RMs. Because all coordinators decide on the transactions' outcome based on the same sequence of consensus instances, they all take the same decisions. Hence, safety is not violated even if several processes become coordinator simultaneously. Liveness, on the other hand, may be violated in this case, because coordinators could jeopardize each other's attempts to reach agreement. Eventually the leader-election oracle ensures that a single coordinator exists, and liveness is ensured.

Incarnating a given resource manager happens in a similar way: the process trying to incarnate the RM sends a special message to the coordinator, which proposes the change in a consensus instance. Once an instance decides on an incarnation change, all next instances consider the newly incarnated resource manager. The decision is also informed to all processes incarnating some RM so that they can identify if they have been replaced.

Traditionally, RMs write their logs on disk before voting. Even when using the log service, RMs may still write locally for a number of reasons, such as (i)recovery speedup, because reading locally is faster than reading remotely, and (ii)minimizing network usage, by sending just empty updates to the service. If the RM does not write its updates locally, then it can resort to the coordinator to obtain them. In this case, the coordinator scans the decisions of all consensus instances to determine the updates of committed transactions concerning the RM when recovering. Since all coordinators maintain the same state, any can be safely contacted.

4.3. Uncoordinated implementation

The uncoordinated implementation is based on the Paxos Commit protocol. The main purpose of this implementation is to save time by not having the votes sent to the service through a coordinator. Instead, each vote is cast in a distinct consensus instance.

When an RM is first contacted by a TM in the context of a transaction, besides executing the requested operation the RM also names the consensus instance in which it will vote to terminate the transaction. The instances are sequentially taken from a per-RM pool and informed to the TM along with the reply to the first operation. The TM informs the named instances to all participants along with its commit request so that they know which instance to use to learn votes and to vote on behalf of each other, should they need.

Besides voting, RMs also use consensus instances to propose changes of incarnations. Because RMs cannot rely on a total order of INCARNATE requests, as in the coordinated implementation, the protocol must rely on some other assumption to limit the scope of each incarnation. The assumption we make is that at most k transactions are executed in parallel by each RM (multiprogramming level).

Let p be a process incarnating RM rm and q a process not incarnating any RM. If q suspects p to have crashed and wants to takeover the role of rm, it proposes the change in the smallest consensus instance in rm's pool that q believes not to be decided yet. q repeats this step with subsequent instances until one of them decides on the INCARNATE request.

When q's proposal to incarnate rm is decided, q still needs to ensure that all consensus instances in which p had possibly voted have been decided. Otherwise, q might attribute one of such instances to a transaction different from the one p had originally done. There are at most $k - 1$ such transactions. That is, if the incarnation change was decided in instance i, then q knows that p can only have voted on instances up to $i + k - 1$, and q's incarnation effectively starts at instance $i + k$. To avoid blocking, q conservatively votes ABORT on all instances in the range $[i + 1, i + k]$.

To recover the state of p, q must recover the updates sent along with votes for all instances smaller than $i + k - 1$. These updates are learned along the decisions of such instances and applied to the database. To desambiguate the order of transactions that had votes issued in instances i and j such that $i > j$ and $i - j < k$, each vote also carries a counter of how many transactions have been committed before issuing such a vote, what is enough to reorder them. (See [3] for more details.)

5. Evaluation

In this section we compare the coordinated implementation of the log service with other relevant commit protocols in the literature. In Section 5.1 we give a brief overview of such approaches, compare them in terms of communication steps, number of messages, and resilience. In Section 5.2, we experimentally compare the coordinated implementation with an uncoordinated one.

5.1. Analytical evaluation

In the 2-Phase Commit protocol (2PC) [8], the TM asks RMs to vote, collects their votes, decides on the transaction's outcome, and informs the RMs. The protocol therefore requires three communication steps and sends up to 3 R messages, where R is the number of RMs. If the TM crashes during the second step, the protocol blocks until the process is recovered.

In the commit protocol of Guerraoui *et al.* [10] (GLS), the RMs do not centralize the decision on the TM: on the second step, they exchange their votes and, on the third step, they communicate their decisions to each other, using a total of $N + 2N^2$ messages—to simplify the analysis, we count messages sent by processes to themselves. Hence, in good runs, all RMs see the decision after three communication steps but, in case of suspicions of failures, they must resort to a consensus instance to ensure correct termination, what adds at least one step to the execution. Assuming an unreliable failure detector to solve consensus, the protocol tolerates any minority of RM crashes.

Paxos Commit (PC) [9] has the same communication complexity and, in some cases, the same communication pattern as GLS in good runs. In PC, however, the second and third steps are used to run the Paxos consensus protocol [12] to agree on the vote of each RM. In the case of suspicions, an RM proposes ABORT on behalf of the suspected RM using its Paxos instance. In PC, the role of deciding on the transaction's outcome is logically dissociated from the RMs; they are played by the acceptors. PC is non-blocking in the presence of crashes of any minority of acceptors. When there is a single acceptor, PC can be configured to have the same communication pattern and resilience as 2PC. If every RM acts also as an acceptor, then it equals GLS. We report this latter case in Table 1, which summarizes aspects of each approach.

The Uncoordinated Log Service (Uncoord) is an extension of PC to cope with the ordering of transactions, durability of updates, and replacement of failed resource managers. These aspects, however, to not enter in our analytical evaluation, where PC and Uncoord are regarded as the same.

While the previous approaches try improve the resilience of 2PC by reducing and distributing the role of the TM, other protocols tried to handle its failure by other means. In the 3-Phase Commit (3PC) [9] protocols, the TM is replaced once it has crashed. Problems may arise, though, if the TM has not in fact crashed, possibly leading to inconsistent termination [1]. Besides, to be replaceable, the TM cannot be the only one to keep data essential for the termination, and disseminating this data introduces more communication steps to the termination (See Table 1).

In the Coordinated Log Service, the replacement of a non-crashed TM may lead to unnecessary aborts, but it does not break the consistency of the protocol. By using Paxos in the coordinated log service, the protocol takes the same number of communications steps to terminate as 3PC: five. Because the same coordinator is used on many transactions, the cost of terminating them is amortized by aggregating votes to be proposed, reducing the number of instances executed in parallel. Parallel instances, in practice, impact negatively on each other due to resource contention.

Jiménez-Peres *et al.* [11] proposed a commit service abstraction and a set of implementations. Different from our abstraction, their commit service is defined for single commit instances. Their implementations provide an optimistic outcome after three steps, and a confirmation after the fourth step, when the transaction can be committed. As we have done with RMs and acceptors in PC, we analyse their protocol co-locating the commit servers and the RMs.

Table 1, below, compares the discussed approaches in terms of number of communication steps required by each protocol, the number of messages sent on each one, and their resilience, that is, the number of resouce manager failures that does not prevent the protocols from terminating.

	Comm. Steps	Messages	Resilience
2-PC	3	$3R$	0
GLS [10]	3	$2R^2 + R$	$< R/2$
PC [9]/Uncoord.	3	$2R^2 + R$	$< R/2$
Commit Service [11]	(3)4	$3R^2 + 2R$	$< R/2$
3-PC [1]	5	$5R$	—
Coord. Log Service	5	$5R$	$< R/2$

Table 1. The cost of some commit protocols

5.2. Experimental evaluation

We have implemented a variant of PC in which votes carry RMs' updates. Moreover, we also augmented it with a procedure to recover RMs based on the acceptors state. We used this Uncoordinated Log Service implementation in our experimental evaluation to show the cost of storing updates on the acceptors instead of locally at the RMs. Analytically, this approach has the same costs and resilience of PC.

We have prototyped the Uncoordinated and the Coordinated log service in Java, as well as the Paxos consensus protocol that underlies both of them, and compared them using the Sprint infrastructure. More details on the prototype can be found in [2]. The evaluation consisted of two benchmarks, explained below, run in the Emulab testbed [16]. All nodes used were equipped with 64-bit Xeon 3GHz processors, interconnected through a Gigabit Ethernet switch.

Micro-benchmark In the first experiment we used a micro-benchmark of non-conflicting transactions, each comprising one write operation executed by one RM; each operation writes 0, 1000, or 7000 bytes of data, allowing us to evaluate the impact of the size of updates carried by votes. To assess the impact of disk writes on the log service, we have run experiments in which consensus acceptors have disk writes enabled and disabled (i.e., consistency relies on at most a minority of acceptors crashing simultaneously).

To compare the different configurations, we first determined the workload needed on each of them to reach a 10ms latency per transaction execution. Each entry in Table 2 presents the throughput, in transactions per second, of the coordinated and the uncoordinated techniques, and their ratio (number between parenthesis). The differences in the first and second lines evidences the cost of writing on stable storage: both approaches benefit from disabling the disk, but the effects are more significant with the uncoordinated approach. The reason is that the coordinated approach already optimizes disk access by using the same consensus instance and a single disk write for multiple transactions.

Going from the left to the right columns in Table 2 we see the effects of update sizes in the log service. As the size grows, fewer votes can fit in the same proposal, increasing the number of Paxos instances needed until each vote requires one full instance, as in the uncoordinated version. At

Disk	0 bytes	1000 bytes	7000 bytes
On	1539/184 (8.36)	771/189 (4.08)	96/97 (0.99)
Off	2936/1249 (2.35)	1559/1181 (1.32)	370/357 (1.04)

Table 2. Coord/Uncoord throughput

this point, the benefits of the coordinator are minimal. Likewise, the gains of not writing on disk are smaller for bigger updates, because more time is spent transferring the data.

TPC-C based benchmark We also evaluated our implementations with a variant of the TPC-C [6] benchmark in which clients submit requests without think times. The transactions and their frequencies, however, are those specified by TPC-C. All tables but one (the read only Items table) were range partitioned among RMs according to their primary keys; the read-only table was replicated on all RMs. As a result, at most 15% transactions involved more than one RM (up to all RMs). Update transactions were 92% of the workload and produced data to be logged not exceeding 1500 bytes. In the experiments, we varied the load by increasing the number of clients, and measured the resulting throughput in transactions per second and their respective response times. Each dotted curve in Figure 1 gives the theoretical relation between throughput and response time for different numbers of clients, as defined by the Little's Law: number of clients = throughput $\times$ response time.

As Figure 1 shows, very small loads (dotted curve with 2 clients) do not significantly differ in performance between configurations. With higher loads, the coordinated version outperforms the uncoordinated one and scales better. 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 is still better off as it can combine data from different RMs.

6. Final remarks

In this paper we have introduced the specification of a log service for transaction processing systems, which provides atomicity and durability to transactions through a non-blocking termination protocol. The service totally orders non-concurrent transactions. 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. Moreover, it safely copes with multiple copies of a resource manager. Due to its general design and simple specification, we believe it can serve as basis for further work.

Some works in the literature have similarities with our service. In Stamus and Cristian's approach [15], for example, the log records of resource managers are aggregated and

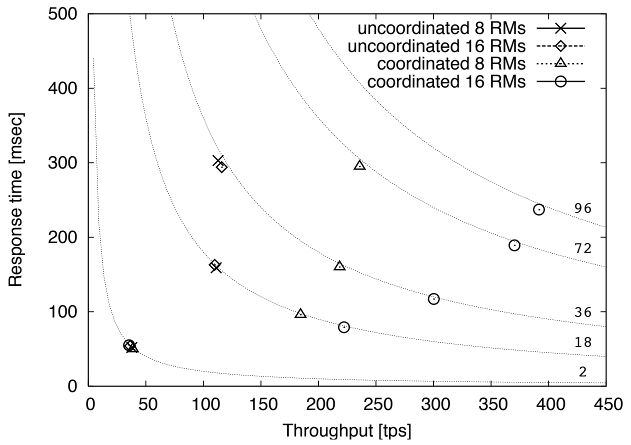

Figure 1. Maximum throughput versus response time of TPC-C transactions. The number of clients is shown next to the curves. Disk writes at acceptors were enabled.

stored at the transaction manager, which is relied upon to implement the service. Their protocol, though, uses a byzantine agreement abstraction and makes strong synchrony assumptions. Besides, it does not consider the recovery of resource and transaction managers on different nodes in case of malfunctioning. Conversely, the log service of Daniels *et al.* [7] does allow recovery on different machines but, instead, lacks the transaction termination feature. Also Mohan *et al.* [13] used byzantine tolerant agreement abstractions. Their extended 2PC protocol can be seen as a byzantine uncoordinated log service implementation.

We also presented two highly available implementations of our log service, namely coordinated and uncoordinated, and provided a comparative experimental performance evaluation. As we have shown in the previous section, these implementations are representative of several well-known protocols in the literature. Although in theory the uncoordinated approach outperforms the coordinated one by two communication steps, in our experimental evaluation the coordinated approach has led to a much higher transaction throughput and smaller response times for small transactions. This result is explained by the higher number of messages sent in parallel in the uncoordinated version, negatively effecting on each other, and by the coordinator being able to terminate possibly many transactions using a single instance of consensus.

As in other consensus-based commit protocols, resource managers and acceptors (consensus deciding processes) can be co-located to minimize the number of nodes in the system. However, we see a strong reason to decouple these two roles in practical scenarios: resource managers are generally more complex software artifacts and, therefore, more error prone than acceptors; the latter must be available for the sake of all transactions in the system and should not risk crashing because of a resource manager error. Besides, by decoupling these roles, the availability of the system becomes determined by the availability of quorum of acceptors, and more easily assessed.

References

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

[2] L. Camargos, F. Pedone, and M. Wieloch. Sprint: a middleware for high-performance transaction processing. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 385–398, New York, NY, USA, 2007. ACM Press.

[3] L. Camargos, M. Wieloch, F. Pedone, and E. Madeira. A Highly Available Log Service for Distributed Transaction Processing. Techical report 2006/08, University of Lugano, Dec. 2006.

[4] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, 1996.

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

[6] T. P. P. Council. http://www.tpc.org/.

[7] 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.

[8] J. Gray. Notes on database op. systems. In *Advanced Course: Operating Systems*, pages 393–481, 1978.

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

[10] 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.

[11] R. Jiménez-Peris, M. Patino-Martinez, G. Alonso, and S. Arevalo. A low-latency non-blocking commit service. *Distributed Computing Conference (DISC01)*, pages 93–107, 2001.

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

[13] 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.

[14] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database Systems Concepts*. McGraw-Hill Science/Engineering/Math, fourth edition, October 2001.

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

[16] 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.