Multi-Shot Distributed Transaction Commit

Gregory Chockler
Royal Holloway, University of London, UK

Alexey Gotsman[1]
IMDEA Software Institute, Madrid, Spain

Abstract

Atomic Commit Problem (ACP) is a single-shot agreement problem similar to consensus, meant to model the properties of transaction commit protocols in fault-prone distributed systems. We argue that ACP is too restrictive to capture the complexities of modern transactional data stores, where commit protocols are integrated with concurrency control, and their executions for different transactions are interdependent. As an alternative, we introduce Transaction Certification Service (TCS), a new formal problem that captures safety guarantees of multi-shot transaction commit protocols with integrated concurrency control. TCS is parameterized by a certification function that can be instantiated to support common isolation levels, such as serializability and snapshot isolation. We then derive a provably correct crash-resilient protocol for implementing TCS through successive refinement. Our protocol achieves a better time complexity than mainstream approaches that layer two-phase commit on top of Paxos-style replication.

2012 ACM Subject Classification Theory of computation → Distributed computing models

Keywords and phrases Atomic commit problem, two-phase commit, Paxos

Digital Object Identifier 10.4230/LIPIcs.DISC.2018.14

1 Introduction

Modern data stores are often required to manage massive amounts of data while providing stringent transactional guarantees to their users. They achieve scalability by partitioning data into independently managed *shards* (aka *partitions*) and fault-tolerance by replicating each shard across a set of servers [9, 14, 42, 34]. Implementing such systems requires sophisticated protocols to ensure that distributed transactions satisfy a conjunction of desirable properties commonly known as ACID: Atomicity, Consistency, Isolation and Durability.

Traditionally, distributed computing literature abstracts ways of achieving these properties into separate problems: in particular, atomic commit problem (ACP) for Atomicity and concurrency control (CC) for Isolation. ACP is formalised as a *one-shot* agreement problem in which multiple shards involved in a transaction need to reach a decision on its final outcome: COMMIT if all shards voted to commit the transaction, and ABORT otherwise [13]. Concurrency control is responsible for determining whether a shard should vote to commit or abort a transaction based on the locally observed conflicts with other active transactions. Although both ACP and CC must be solved in any realistic transaction processing system, they are traditionally viewed as disjoint in the existing literature. In particular, solutions for ACP treat the votes as the inputs of the problem, and leave the interaction with CC, which is responsible for generating the votes, outside the problem scope [38, 2, 23, 16].

[1] Alexey Gotsman was supported by an ERC Starting Grant RACCOON.

32nd International Symposium on Distributed Computing (DISC 2018).
Editors: Ulrich Schmid and Josef Widder; Article No. 14; pp. 14:1–14:18
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

This separation, however, is too simplistic to capture the complexities of many practical implementations in which commit protocols and concurrency control are tightly integrated, and as a result, may influence each other in subtle ways. For example, consider the classical *two-phase commit (2PC)* protocol [15] for solving ACP among reliable processes. A transaction processing system typically executes a 2PC instance for each transaction [32, 31, 34, 39]. When a processes p_i managing a shard s receives a transaction t, it performs a local concurrency-control check and accordingly votes to commit or abort t. The votes on t by different processes are aggregated, and the final decision is then distributed to all processes. If p_i votes to commit t, as long as it does not know the final decision on t, it will have to conservatively presume t as committed. This may cause p_i to vote ABORT in another 2PC instance for a transaction t' conflicting with t, even if in the end t is aborted. In this case, the outcome of one 2PC instance (for t') depends on the internals of the execution of another instance (for t) and the concurrency-control policy used.

At present, the lack of a formal framework capturing such intricate aspects of real implementations makes them difficult to understand and prove correct. In this paper, we take the first step towards bridging this gap. We introduce *Transaction Certification Service (TCS, §2)*, a new formal problem capturing the safety guarantees of a multi-shot transaction commit protocol with integrated concurrency control. The TCS exposes a simple interface allowing clients to submit transactions for *certification* via a `certify` request, which returns COMMIT or ABORT. A TCS is meant to be used in the context of transactional processing systems with optimistic concurrency control, where transactions are first executed optimistically, and the results (e.g., read and write sets) are submitted for certification to the TCS. In contrast to ACP, TCS does not impose any restrictions on the number of repeated `certify` invocations or their concurrency. It therefore lends itself naturally to formalising the interactions between transaction commit and concurrency control. To this end, TCS is parameterised by a *certification function*, which encapsulates the concurrency-control policy for the desired isolation level, such as serializability and snapshot isolation [1]. The correctness of TCS is then formulated by requiring that its certification decisions be consistent with the certification function.

We leverage TCS to develop a formal framework for constructing provably correct multi-shot transaction commit protocols with customisable isolation levels. The core ingredient of our framework is a new *multi-shot two-phase commit protocol* (§3). It formalises how the classical 2PC interacts with concurrency control in many practical transaction processing systems [32, 31, 34, 39] in a way that is parametric in the isolation level provided. The protocol also serves as a *template* for deriving more complex TCS implementations. We prove that the multi-shot 2PC protocol correctly implements a TCS with a given certification function, provided the concurrency-control policies used by each shard match this function.

We next propose a *crash fault-tolerant* TCS implementation and establish its correctness by proving that it simulates multi-shot 2PC (§4). A common approach to making 2PC fault-tolerant is to get every shard to simulate a reliable 2PC process using a replication protocol, such as Paxos [16, 18, 9, 14, 42]. Similarly to recent work [41, 26], our implementation optimises the time complexity of this scheme by weaving 2PC and Paxos together. In contrast to previous work, our protocol is both generic in the isolation level and rigorously proven correct. It can therefore serve as a reference solution for future distributed transaction commit implementations. Moreover, a variant of our protocol has a time complexity matching the lower bounds for consensus [24, 6] and non-blocking atomic commit [13].

The main idea for achieving such a low time complexity is to eliminate the Paxos consensus required in the vanilla fault-tolerant 2PC to persist the final decision on a transaction at a shard. Instead, the decision is propagated to the relevant shard replicas asynchronously.

This means that different shard replicas may receive the final decision on a transaction at different times, and thus their states may be inconsistent. To deal with this, in our protocol the votes are computed locally by a single shard *leader* based on the information available to it; other processes merely store the votes. Similarly to [29, 22], such a *passive replication* approach requires a careful design of recovery from leader failures. Another reduction in time complexity comes from the fact that our protocol avoids consistently replicating the 2PC coordinator: we allow any process to take over as a coordinator by accessing the current state of the computation at shards. The protocol ensures that all coordinators will reach the same decision on a transaction.

2 Transaction Certification Service

Interface. A *Transaction Certification Service (TCS)* accepts *transactions* from $\mathcal{T}$ and produces *decisions* from $\mathcal{D} = \{\textsc{abort}, \textsc{commit}\}$. Clients interact with the TCS using two types of *actions*: certification requests of the form $\mathtt{certify}(t)$, where $t \in \mathcal{T}$, and responses of the form $\mathtt{decide}(t, d)$, where $d \in \mathcal{D}$.

In this paper we focus on transactional processing systems using optimistic concurrency control. Hence, we assume that a transaction submitted to the TCS includes all the information produced by its optimistic execution. As an example, consider a transactional system managing *objects* in the set Obj with values in the set Val, where transactions can execute reads and writes on the objects. The objects are associated with a totally ordered set Ver of *versions* with a distinguished minimum version v_0. Then each transaction t submitted to the TCS may be associated with the following data:

- *Read set* $R(t) \subseteq 2^{\mathsf{Obj} \times \mathsf{Ver}}$: the set of objects with their versions that t read, which contains at most one version per object.
- *Write set* of $W(t) \subseteq 2^{\mathsf{Obj} \times \mathsf{Val}}$: the set of objects with their values that t wrote, which contains at most one value per object. We require that any object written has also been read: $\forall (x, \_) \in W(t).\, (x, \_) \in R(t)$.
- *Commit version* $V_c(t) \in \mathsf{Ver}$: the version to be assigned to the writes of t. We require that this version be higher than any of the versions read: $\forall (\_, v) \in R(t).\, V_c(t) > v$.

Certification functions. A TCS is specified using a *certification function* $f : 2^{\mathcal{T}} \times \mathcal{T} \to \mathcal{D}$, which encapsulates the concurrency-control policy for the desired isolation level. The result $f(T, t)$ is the decision for the transaction t given the set of the previously committed transactions T. We require f to be *distributive* in the following sense:

$$\forall T_1, T_2, t.\, f(T_1 \cup T_2, t) = f(T_1, t) \sqcap f(T_2, t), \tag{1}$$

where the $\sqcap$ operator is defined as follows: $\textsc{commit} \sqcap \textsc{commit} = \textsc{commit}$ and $d \sqcap \textsc{abort} = \textsc{abort}$ for any d. This requirement is justified by the fact that common definitions of $f(T, t)$ check t for conflicts against each transaction in T separately.

For example, given the above domain of transactions, the following certification function encapsulates the classical concurrency-control policy for serializability [40]: $f(T, t) = \textsc{commit}$ iff none of the versions read by t have been overwritten by a transaction in T, i.e.,

$$\forall x, v.\, (x, v) \in R(t) \implies (\forall t' \in T.\, (x, \_) \in W(t') \implies V_c(t') \le v). \tag{2}$$

A certification function for snapshot isolation (SI) [1] is similar, but restricts the certification check to the objects the transaction t writes: $f(T, t) = \textsc{commit}$ iff

$$\forall x, v.\, (x, v) \in R(t) \wedge (x, \_) \in W(t) \implies (\forall t' \in T.\, (x, \_) \in W(t') \implies V_c(t') \le v). \tag{3}$$

It is easy to check that the certification functions (2) and (3) are distributive.

Histories. We represent TCS executions using *histories* – sequences of `certify` and `decide` actions such that every transaction appears at most once as a parameter to `certify`, and each `decide` action is a response to exactly one preceding `certify` action. For a history h we let $\mathsf{act}(h)$ be the set of actions in h. For actions $a, a' \in \mathsf{act}(h)$, we write $a \prec_h a'$ when a occurs before a' in h. A history h is *complete* if every `certify` action in it has a matching `decide` action. A complete history is *sequential* if it consists of pairs of `certify` and matching `decide` actions. A transaction t *commits* in a history h if h contains $\mathtt{decide}(t, \textsc{commit})$. We denote by $\mathsf{committed}(h)$ the projection of h to actions corresponding to the transactions that are committed in h. For a complete history h, a *linearization* ℓ of h [21] is a sequential history such that: *(i)* h and ℓ contain the same actions; and *(ii)*

$$\forall t, t'.\, \mathtt{decide}(t, \_) \prec_h \mathtt{certify}(t') \implies \mathtt{decide}(t, \_) \prec_\ell \mathtt{certify}(t').$$

TCS correctness. A complete sequential history h is *legal* with respect to a certification function f, if its certification decisions are computed according to f:

$$\forall a = \mathtt{decide}(t, d) \in \mathsf{act}(h).\, d = f(\{t' \mid \mathtt{decide}(t', \textsc{commit}) \prec_h a\}, t).$$

A history h is *correct* with respect to f if $h \mid \mathsf{committed}(h)$ has a legal linearization. A TCS implementation is *correct* with respect to f if so are all its histories.

A correct TCS can be readily used in a transaction processing system. For example, consider the domain of transactions defined earlier. A typical system based on optimistic concurrency control will ensure that transactions submitted for certification read versions that already exist in the database. Formally, it will produce only histories h such that, for a transaction t submitted for certification in h, if $(x, v) \in R(t)$, then there exists a t' such that $(x, v) \in W(t')$, and h contains $\mathtt{decide}(t', \textsc{commit})$ before $\mathtt{certify}(t)$. It is easy to check that, if such a history h is correct with respect to the certification function (2), then it is also serializable. Hence, TCS correct with respect to certification function (2) can indeed be used to implement serializability.

3 Multi-Shot 2PC and Shard-Local Certification Functions

We now present a multi-shot version of the classical *two-phase commit (2PC)* protocol [15], parametric in the concurrency-control policy used by each shard. We then prove that the protocol implements a correct transaction certification service parameterised by a given certification function, provided per-shard concurrency control matches this function. Like 2PC, our protocol assumes reliable processes. In the next section, we establish the correctness of a protocol that allows crashes by proving that it simulates the behaviour of multi-shot 2PC.

System model. We consider an asynchronous message-passing system consisting of a set of processes $\mathcal{P}$. In this section we assume that processes are reliable and are connected by reliable FIFO channels. We assume a function $\mathsf{client} : \mathcal{T} \to \mathcal{P}$ determining the client process that issued a given transaction. The data managed by the system are partitioned into *shards* from a set $\mathcal{S}$. A function $\mathsf{shards} : \mathcal{T} \to 2^{\mathcal{S}}$ determines the shards that need to certify a given transaction, which are usually the shards storing the data the transaction accesses. Each shard $s \in \mathcal{S}$ is managed by a process $\mathsf{proc}(s) \in \mathcal{P}$. For simplicity, we assume that different processes manage different shards.

Algorithm 1: Multi-shot 2PC protocol at a process p_i managing a shard s_0.

```
1  next ← $-1 \in \mathbb{Z}$;
2  txn[ ] $\in \mathbb{N} \rightarrow \mathcal{T}$;
3  vote[ ] $\in \mathbb{N} \rightarrow \{\textsc{commit}, \textsc{abort}\}$;
4  dec[ ] $\in \mathbb{N} \rightarrow \{\textsc{commit}, \textsc{abort}\}$;
5  phase[ ] ← $(\lambda k.\, \textsc{start}) \in \mathbb{N} \rightarrow \{\textsc{start}, \textsc{prepared}, \textsc{decided}\}$;

6  function certify($t$)
7      send PREPARE($t$) to proc(shards($t$));

8  when received PREPARE($t$)
9      next ← next + 1;
10     txn[next] ← $t$;
11     vote[next] ← $f_{s_0}(\{\mathsf{txn}[k] \mid k < \mathsf{next} \wedge \mathsf{phase}[k] = \textsc{decided} \wedge \mathsf{dec}[k] = \textsc{commit}\}, t) \sqcap$
                      $g_{s_0}(\{\mathsf{txn}[k] \mid k < \mathsf{next} \wedge \mathsf{phase}[k] = \textsc{prepared} \wedge \mathsf{vote}[k] = \textsc{commit}\}, t)$;
12     phase[next] ← PREPARED;
13     send PREPARE_ACK($s_0$, next, $t$, vote[next]) to coord($t$);

14 when received PREPARE_ACK($s, pos_s, t, d_s$) for every $s \in$ shards($t$)
15     send DECISION($t, \bigsqcap_{s \in \mathsf{shards}(t)} d_s$) to client($t$);
16     forall $s \in$ shards($t$) do
17         send DECISION($pos_s, \bigsqcap_{s \in \mathsf{shards}(t)} d_s$) to proc($s$)

18 when received DECISION($k, d$)
19     dec[$k$] ← $d$;
20     phase[$k$] ← DECIDED;

21 non-deterministically for some $k \in \mathbb{N}$
22     pre: phase[$k$] = DECIDED;
23     phase[$k$] ← PREPARED;

24 non-deterministically for some $k \in \mathbb{N}$
25     pre: phase[$k$] $\neq$ START;
26     send PREPARE_ACK($s_0, k$, txn[$t$], vote[$k$]) to coord($t$);
```

Protocol: common case. We give the pseudocode of the protocol in Algorithm 1 and illustrate its message flow in Figure 1a. Each handler in Algorithm 1 is executed atomically.

To certify a transaction t, a client sends it in a PREPARE message to the relevant shards (line 6)[2]. A process managing a shard arranges all transactions received into a total *certification order*, stored in an array txn; a next variable points to the last filled slot in the array. Upon receiving a transaction t (line 8), the process stores t in the next free slot of txn. The process also computes its *vote*, saying whether to COMMIT or ABORT the transaction, and stores it in an array vote. We explain the vote computation in the following; intuitively,

[2] In practice, the client only needs to send the data relevant to the corresponding shard. We omit this optimisation to simplify notation.

the vote is determined by whether the transaction t conflicts with a previously received transaction. After the process managing a shard s receives t, we say that t is *prepared* at s. The process keeps track of transaction status in an array phase, whose entries initially store START, and are changed to PREPARED once the transaction is prepared. Having prepared the transaction t, the process sends a `PREPARE_ACK` message with its position in the certification order and the vote to a *coordinator* of t. This is a process determined using a function $\mathsf{coord} : \mathcal{T} \to \mathcal{P}$ such that $\forall t.\, \mathsf{coord}(t) \in \mathsf{proc}(\mathsf{shards}(t))$.

The coordinator of a transaction t acts once it receives a `PREPARE_ACK` message for t from each of its shards s, which carries the vote d_s by s (line 14). The coordinator computes the final decision on t using the $\sqcap$ operator (§2) and sends it in `DECISION` messages to the client and to all the relevant shards. When a process receives a decision for a transaction (line 18), it stores the decision in a dec array, and advances the transaction's phase to DECIDED.

Vote computation. A process managing a shard s computes votes as a conjunction of two *shard-local certification functions* $f_s : 2^{\mathcal{T}} \times \mathcal{T} \to \mathcal{D}$ and $g_s : 2^{\mathcal{T}} \times \mathcal{T} \to \mathcal{D}$. Unlike the certification function of §2, the shard-local functions are meant to check for conflicts only on objects managed by s. They take as their first argument the sets of transactions already decided to commit at the shard, and respectively, those that are only prepared to commit (line 11). We require that the above functions be distributive, similarly to (1).

For example, consider the transaction model given in §2 and assume that the set of objects Obj is partitioned among shards: $\mathsf{Obj} = \biguplus_{s \in \mathcal{S}} \mathsf{Obj}_s$. Then the shard-local certification functions for serializability are defined as follows: $f_s(T, t) = \text{COMMIT}$ iff

$$\forall x \in \mathsf{Obj}_s.\, \forall v.\, (x, v) \in R(t) \implies (\forall t' \in T.\, (x, \_) \in W(t') \implies V_c(t') \le v), \tag{4}$$

and $g_s(T, t) = \text{COMMIT}$ iff

$$\begin{aligned} \forall x \in \mathsf{Obj}_s.\, \forall v.\, ((x, \_) \in R(t) &\implies (\forall t' \in T.\, (x, \_) \notin W(t'))) \wedge \\ ((x, \_) \in W(t) &\implies (\forall t' \in T.\, (x, \_) \notin R(t'))). \end{aligned} \tag{5}$$

The function f_s certifies a transaction t against previously committed transactions T similarly to the certification function (2), but taking into account only the objects managed by the shard s. The function g_s certifies t against transactions T prepared to commit.

The first conjunct of (5) aborts a transaction t if it read an object written by a transaction t' prepared to commit. To motivate this condition, consider the following example. Assume that a shard managing an object x votes to commit a transaction t' that read a version v_1 of x and wants to write a version $v_2 > v_1$ of x. If the shard now receives another transaction t that read the version v_1 of x, the shard has to abort t: if t' does commit in the end, allowing t to commit would violate serializability, since it would have read stale data. On the other hand, once the shard receives the abort decision on t', it is free to commit t.

The second conjunct of (5) aborts a transaction t if it writes to an object read by a transaction t' prepared to commit. To motivate this, consider the following example, adapted from [37]. Assume transactions t_1 and t_2 both read a version v_1 of x at shard s_1 and a version v_2 of y at shard s_2; t_1 wants to write a version $v_2' > v_2$ of y, and t_2 wants to write a version $v_2 > v_1$ of x. Assume further that s_1 receives t_1 first and votes to commit it, and s_2 receives t_2 first and votes to commit it as well. If s_1 now receives t_2 and s_2 receives t_1, the second conjunct of (5) will force them to abort: if the shards let the transactions commit, the resulting execution would not be serializable, since one of the transactions must read the value written by the other.

A simple way of implementing (5) is, when preparing a transaction, to acquire read locks on its read set and write locks on its write set; the transaction is aborted if the locks cannot be acquired. The shard-local certification functions are a more abstract way of defining the behaviour of this and other implementations [32, 31, 34, 39, 37]. They can also be used to define weaker isolation levels than serializability. As an illustration, we can define shard-local certification functions for snapshot isolation as follows: $f_s(T, t) = \text{COMMIT}$ iff

$$\forall x \in \mathsf{Obj}_s.\, \forall v.\, (x, v) \in R(t) \wedge (x, \_) \in W(t) \implies (\forall t' \in T.\, (x, \_) \in W(t') \implies V_c(t') \leq v),$$

and $g_s(T, t) = \text{COMMIT}$ iff

$$(x, \_) \in W(t) \implies (\forall t' \in T.\, (x, \_) \notin W(t')).$$

The function f_s restricts the global function (3) to the objects managed by the shard s. Since snapshot isolation allows reading stale data, the function g_s only checks for write conflicts.

For shard-local certification functions to correctly approximate a given global function f, we require the following relationships. For a set of transactions $T \subseteq \mathcal{T}$, we write $T \mid s$ to denote the *projection* of T on shard s, i.e., $\{t \in T \mid s \in \mathsf{shards}(t)\}$. Then we require that

$$\forall t \in \mathcal{T}.\, \forall T \subseteq \mathcal{T}.\, f(T, t) = \text{COMMIT} \iff \forall s \in \mathsf{shards}(t).\, f_s((T \mid s), t) = \text{COMMIT}. \qquad (6)$$

In addition, for each shard s, the two functions f_s and g_s are required to be related to each other as follows:

$$\forall t.\, s \in \mathsf{shards}(t) \implies (\forall T.\, g_s(T, t) = \text{COMMIT} \implies f_s(T, t) = \text{COMMIT}); \qquad (7)$$

$$\forall t, t'.\, s \in \mathsf{shards}(t) \cap \mathsf{shards}(t') \implies (g_s(\{t\}, t') = \text{COMMIT} \implies f_s(\{t'\}, t) = \text{COMMIT}). \qquad (8)$$

Property (7) requires the conflict check performed by g_s to be no weaker than the one performed by f_s. Property (8) requires a form of commutativity: if t' is allowed to commit after a still-pending transaction t, then t would be allowed to commit after t'. The above shard-local functions for serializability and snapshot isolation satisfy (6)-(8).

Forgetting and recalling decisions. The protocol in Algorithm 1 has two additional handlers at lines 21 and 24, executed non-deterministically. As we show in §4, these are required for the abstract protocol to capture the behaviour of optimised fault-tolerant TCS implementations. Because of process crashes, such implementations may temporarily lose the information about some final decisions, and later reconstruct it from the votes at the relevant shards. In the meantime, the absence of the decisions may affect some vote computations as we explained above. The handler at line 21 forgets the decision on a transaction (but not its vote). The handler at line 24 allows processes to resend the votes they know to the coordinator, which will then resend the final decisions (line 14). This allows a process that forgot a decision to reconstruct it from the votes stored at the relevant shards.

Correctness. The following theorem shows the correctness of multi-shot 2PC. In particular, it shows that the shard-local concurrency control given by f_s and g_s correctly implements the shard-agnostic concurrency control given by a global certification function f.

▶ **Theorem 1.** *A transaction certification service implemented using the multi-shot 2PC protocol in Algorithm 1 is correct with respect to a certification function f, provided shard-local certification functions f_s and g_s satisfy (6)-(8).*

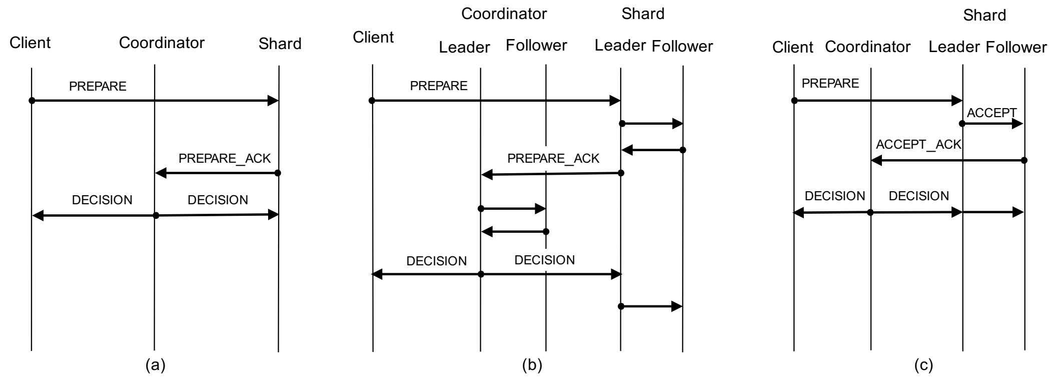

Figure 1 Message flow diagrams illustrating the behaviour of (a) multi-shot 2PC; (b) multi-shot 2PC with shards replicated using Paxos; (c) optimised protocol weaving together multi-shot 2PC and Paxos.

We give the proof in [7, §A]. Its main challenge is that, in multi-shot 2PC, certification orders at different shards may disagree on the order of concurrently certified transactions; however, a correct TCS has to certify transactions according to a single total order. We use the commutativity property (8) to show that per-shard certification orders arising in the protocol can be merged into the desired single total order.

4 Fault-Tolerant Commit Protocol

System model. We now weaken the assumptions of the previous section by allowing processes to fail by crashing, i.e., permanently stopping execution. We still assume that processes are connected by reliable FIFO channels in the following sense: messages are delivered in the FIFO order, and messages between non-faulty processes are guaranteed to be eventually delivered. Each shard s is now managed by a group of $2f+1$ processes, out of which at most f can fail. We call a set of $f+1$ processes in this group a *quorum* for s. For a shard s we redefine $\mathsf{proc}(s)$ to be the set of processes managing this shard. For simplicity, we assume that the groups of processes managing different shards are disjoint.

Vanilla protocol. A straightforward way to implement a TCS in the above model is to use state-machine replication [36] to make a shard simulate a reliable process in multi-shot 2PC; this is usually based on a consensus protocol such as Paxos [27]. In this case, final decisions on transactions are never forgotten, and hence, the handlers at lines 21 and 24 are not simulated. Even though this approach is used by several systems [9, 14, 42], multiple researchers have observed that the resulting protocol requires an unnecessarily high number of message delays [41, 26, 28]. Namely, every action of multi-shot 2PC in Figure 1a requires an additional round trip to a quorum of processes in the same shard to persist its effect, resulting in the message-flow diagram in Figure 1b. Note that the coordinator actions have to be replicated as well, since multi-shot 2PC will block if the coordinator fails. The resulting protocol requires 7 message delays for a client to learn a decision on a transaction.

Optimised protocol overview. In Algorithms 2 and 3 we give a commit protocol that reduces the number of message delays by weaving together multi-shot 2PC across shards and a Paxos-like protocol within each shard. We omit details related to message retransmissions from the code. We illustrate the message flow of the protocol in Figure 1c and summarise the key invariants used in its proof of correctness in Figure 2.

A process maintains the same variables as in the multi-shot 2PC protocol (Algorithm 1) and a few additional ones. Every process in a shard is either the *leader* of the shard or a *follower*. If the leader fails, one of the followers takes over. A status variable records whether the process is a LEADER, a FOLLOWER or is in a special RECOVERING state used during leader changes. A period of time when a particular process acts as a leader is denoted using integer *ballots*. For a ballot $b \geq 1$, the process $\mathsf{leader}(b) = ((b-1) \mod (2f+1))$ is the leader of the ballot. At any given time, a process participates in a single ballot, stored in a variable ballot. During leader changes we also use an additional ballot variable cballot.

Unlike the vanilla protocol illustrated in Figure 1b, our protocol does not perform consensus to persist the contents of a DECISION message in a shard. Instead, the final decision on a transaction is sent to the members of each relevant shard asynchronously. This means that different shard members may receive the decision on a transaction at different times. Since the final decision on a transaction affects vote computations on transactions following it in the certification order (§3), computing the vote on a later transaction at different shard members may yield different outcomes. To deal with this, in our protocol only the leader constructs the certification order and computes votes. Followers are passive: they merely copy the leader's decisions. A final decision is taken into account in vote computations at a shard once it is received by the shard's leader.

Failure-free case. To certify a transaction t, a client sends it in a PREPARE message to the relevant shards (line 10). A process p_i handles the message only when it is the leader of its shard s_0 (line 12). We defer the description of the cases when another process p_j is resending the PREPARE message to p_i (line 13), and when p_i has already received t in the past (line 14).

Upon receiving PREPARE(t), the leader p_i first determines a process p that will serve as the coordinator of t. If the leader receives t for the first time (line 16), then, similarly to multi-shot 2PC, it appends t to the certification order and computes the vote based on the locally available information. The leader next performs an analogue of "phase 2" of Paxos, trying to convince its shard s_0 to accept its proposal. To this end, it sends an ACCEPT message to s_0 (including itself, for uniformity), which is analogous to the "2a" message of Paxos (line 21). The message carries the leader's ballot, the transaction t, its position in the certification order, the vote and the identity of t's coordinator. The leader code ensures Invariant 1 in Figure 2: in a given ballot b, a unique transaction-vote pair can be assigned to a slot k in the certification order.

A process handles an ACCEPT message only if it participates in the corresponding ballot (line 23). If the process has not heard about t before, it stores the transaction and the vote and advances the transaction's phase to PREPARED. It then sends an ACCEPT_ACK message to the coordinator of t, analogous to the "2b" message of Paxos. This confirms that the process has accepted the transaction and the vote. The certification order at a follower is always a prefix of the certification order at the leader of the ballot the follower is in, as formalised by Invariant 2. This invariant is preserved when the follower receives ACCEPT messages due to the FIFO ordering of channels.

The coordinator of a transaction t acts once it receives a quorum of ACCEPT_ACK messages for t from each of its shards $s \in \mathsf{shards}(t)$, which carry the vote d_s by s (line 29). The coordinator computes the final decision on t and sends it in DECISION messages to the client and to each of the relevant shards. When a process receives a decision for a transaction (line 33), the process stores it and advances the transaction's phase to DECIDED.

Once the final decision on a transaction is delivered to the leader of a shard, it is taken into account in future vote computations at this shard. Taking as an example the shard-local functions for serializability (4) and (5), if a transaction that wrote to an object x is finally

1. If $\mathtt{ACCEPT}(b, k, t_1, d_1, \_)$ and $\mathtt{ACCEPT}(b, k, t_2, d_2, \_)$ messages are sent to the same shard, then $t_1 = t_2$ and $d_1 = d_2$.
2. After a process receives and acknowledges $\mathtt{ACCEPT}(b, k, t, d, \_)$, we have $\mathsf{txn} = txn\rfloor_k$ and $\mathsf{vote} = vote\rfloor_k$, where txn and $vote$ are the values of the arrays txn and vote at $\mathsf{leader}(b)$ when it sent the ACCEPT message.
3. Assume that a quorum of processes in s received $\mathtt{ACCEPT}(b, k, t, d, \_)$ and responded to it with $\mathtt{ACCEPT\_ACK}(s, b, k, t, d)$, and at the time $\mathsf{leader}(b)$ sent $\mathtt{ACCEPT}(b, k, t, d, \_)$ it had $\mathsf{txn}\rfloor_k = txn$ and $\mathsf{vote}\rfloor_k = vote$. Whenever at a process in s we have $\mathsf{status} \in \{\text{LEADER}, \text{FOLLOWER}\}$ and $\mathsf{ballot} = b' > b$, we also have $\mathsf{txn}\rfloor_k = txn$ and $\mathsf{vote}\rfloor_k = vote$.
4. If $\mathtt{ACCEPT}(b, k_1, t, \_, \_)$ and $\mathtt{ACCEPT}(b, k_2, t, \_, \_)$ messages are sent to the same shard, then $k_1 = k_2$.
5. At any process, all transactions in the txn array are distinct.
6. **a.** For any messages $\mathtt{DECISION}(\_, k, d_1)$ and $\mathtt{DECISION}(\_, k, d_2)$ sent to processes in the same shard, we have $d_1 = d_2$.
   b. For any messages $\mathtt{DECISION}(t, d_1)$ and $\mathtt{DECISION}(t, d_2)$ sent, we have $d_1 = d_2$.
7. **a.** Assume that a quorum of processes in s have sent $\mathtt{ACCEPT\_ACK}(s, b_1, k, t_1, d_1)$ and a quorum of processes in s have sent $\mathtt{ACCEPT\_ACK}(s, b_2, k, t_2, d_2)$. Then $t_1 = t_2$ and $d_1 = d_2$.
   b. Assume that a quorum of processes in s have sent $\mathtt{ACCEPT\_ACK}(s, b_1, k_1, t, d_1)$ and a quorum of processes in s have sent $\mathtt{ACCEPT\_ACK}(s, b_2, k_2, t, d_2)$. Then $k_1 = k_2$ and $d_1 = d_2$.

Figure 2 Key invariants of the fault-tolerant protocol. We let $\alpha\rfloor_k$ be the prefix of the sequence α of length k.

decided to abort, then delivering this decision to the leader may allow another transaction writing to x to commit.

Leader recovery. We next explain how the protocol deals with failures, starting from a leader failure. The goal of the leader recovery procedure is to preserve Invariant 3: if in a ballot b a shard s accepted a vote d on a transaction t at the position k in the certification order, then this vote will persist in all future ballots; this is furthermore true for all votes the leader of ballot b took into account when computing d. The latter property is necessary for the shard to simulate the behaviour of a reliable process in multi-shot 2PC that maintains a unique certification order. To ensure this property, our recovery procedure includes an additional message from the new leader to the followers ensuring that, before a follower starts accepting proposals from the new leader, it has brought its state in sync with that of the leader (this is similar to [29, 22]). The ballot of the last leader a follower synchronised with in this way is recorded in cballot.

We now describe the recovery procedure in detail. When a process p_i suspects the leader of its shard of failure, it may try to become a new leader by executing the `recover` function (line 37). The process picks a ballot that it leads higher than ballot and sends it in a NEW_LEADER message to the shard members (including itself); this message is analogous to the "1a" message in Paxos. When a process receives a $\mathtt{NEW\_LEADER}(b)$ message (line 39), it first checks that the proposed ballot b is higher than his. In this case, it sets its ballot to b and changes its status to RECOVERING, which causes it to stop processing PREPARE, ACCEPT and DECISION messages. It then replies to the new leader with a NEW_LEADER_ACK message containing all components of its state, analogous to the "1b" message of Paxos.

Algorithm 2: Fault-tolerant commit protocol at a process p_i in a shard s_0: failure-free case.

```
$\mathsf{next} \leftarrow -1 \in \mathbb{Z}$;
$\mathsf{txn}[\,] \in \mathbb{N} \rightarrow \mathcal{T}$;
$\mathsf{vote}[\,] \in \mathbb{N} \rightarrow \{\text{COMMIT}, \text{ABORT}\}$;
$\mathsf{dec}[\,] \in \mathbb{N} \rightarrow \{\text{COMMIT}, \text{ABORT}\}$;
$\mathsf{phase}[\,] \leftarrow (\lambda k.\,\text{START}) \in \mathbb{N} \rightarrow \{\text{START}, \text{PREPARED}, \text{DECIDED}\}$;
$\mathsf{status} \in \{\text{LEADER}, \text{FOLLOWER}, \text{RECOVERING}\}$;
$\mathsf{ballot} \leftarrow 0 \in \mathbb{N}$;
$\mathsf{cballot} \leftarrow 0 \in \mathbb{N}$;

function certify($t$)
    send PREPARE($t$) to $\mathsf{proc}(\mathsf{shards}(t))$;

when received PREPARE($t$) from $p_j$ or a client
    pre: $\mathsf{status} = \text{LEADER}$;
    if received from a process $p_j$ then $p \leftarrow p_j$ else $p \leftarrow \mathsf{coord}(t)$;
    if $\exists k.\, t = \mathsf{txn}[k]$ then
        send ACCEPT($\mathsf{ballot}, k, t, \mathsf{vote}[k], p$) to $\mathsf{proc}(s_0)$
    else
        $\mathsf{next} \leftarrow \mathsf{next} + 1$;
        $\mathsf{txn}[\mathsf{next}] \leftarrow t$;
        $\mathsf{vote}[\mathsf{next}] \leftarrow f_{s_0}(\{\mathsf{txn}[k] \mid k < \mathsf{next} \wedge \mathsf{phase}[k] = \text{DECIDED} \wedge \mathsf{dec}[k] = \text{COMMIT}\}, t) \sqcap$
            $g_{s_0}(\{\mathsf{txn}[k] \mid k < \mathsf{next} \wedge \mathsf{phase}[k] = \text{PREPARED} \wedge \mathsf{vote}[k] = \text{COMMIT}\}, t)$;
        $\mathsf{phase}[\mathsf{next}] \leftarrow \text{PREPARED}$;
        send ACCEPT($\mathsf{ballot}, \mathsf{next}, t, \mathsf{vote}[\mathsf{next}], p$) to $s_0$;

when received ACCEPT($b, k, t, d, p$)
    pre: $\mathsf{status} \in \{\text{LEADER}, \text{FOLLOWER}\} \wedge \mathsf{ballot} = b$;
    if $\mathsf{phase}[k] = \text{START}$ then
        $\mathsf{txn}[k] \leftarrow t$;
        $\mathsf{vote}[k] \leftarrow d$;
        $\mathsf{phase}[k] \leftarrow \text{PREPARED}$;
    send ACCEPT_ACK($s_0, b, k, t, d$) to $p$;

when for every $s \in \mathsf{shards}(t)$ received a quorum of ACCEPT_ACK($s, b_g, pos_g, t, d_g$)
    send DECISION($t, \bigsqcap_{s \in \mathsf{shards}(t)} d_s$) to $\mathsf{client}(t)$;
    forall $s \in \mathsf{shards}(t)$ do
        send DECISION($b_s, pos_s, \bigsqcap_{s \in \mathsf{shards}(t)} d_s$) to $\mathsf{proc}(s)$

when received DECISION($b, k, d$)
    pre: $\mathsf{status} \in \{\text{LEADER}, \text{FOLLOWER}\} \wedge \mathsf{ballot} \geq b \wedge \mathsf{phase}[k] = \text{PREPARED}$;
    $\mathsf{dec}[k] \leftarrow d$;
    $\mathsf{phase}[k] \leftarrow \text{DECIDED}$;
```

Algorithm 3: Fault-tolerant commit protocol at a process p_i in a shard s_0: recovery.

```
37 function recover()
38     send NEW_LEADER(any ballot $b$ such that $b >$ ballot $\wedge$ leader(ballot) $= p_i$) to $s_0$;

39 when received NEW_LEADER($b$) from $p_j$
40     pre: $b >$ ballot;
41     status $\leftarrow$ RECOVERING;
42     ballot $\leftarrow b$;
43     send NEW_LEADER_ACK(ballot, cballot, txn, vote, dec, phase) to $p_j$;

44 when received $\{$NEW_LEADER_ACK($b, cballot_j, txn_j, vote_j, dec_j, phase_j$) $\mid p_j \in Q\}$
       from a quorum $Q$ in $s_0$
45     pre: status = RECOVERING $\wedge$ ballot $= b$;
46     var $J \leftarrow$ the set of $j$ with the maximal $cballot_j$;
47     forall $k$ do
48         if $\exists j \in J.\, phase_j[k] \geq$ PREPARED then
49             txn[$k$] $\leftarrow txn_j[k]$;
50             vote[$k$] $\leftarrow vote_j[k]$;
51             phase[$k$] $\leftarrow$ PREPARED;
52         if $\exists j.\, phase_j[k] =$ DECIDED then
53             dec $\leftarrow dec_j[k]$;
54             phase[$k$] $\leftarrow$ DECIDED;
55     next $\leftarrow \min\{k \mid$ phase[$k$] $\neq$ START$\}$;
56     cballot $\leftarrow b$;
57     status $\leftarrow$ LEADER;
58     send NEW_STATE($b$, txn, vote, dec, phase) to proc($s_0$) $\setminus \{p_i\}$;

59 when received NEW_STATE($b, txn, vote, dec, phase$) from $p_j$
60     pre: $b \geq$ ballot;
61     status $\leftarrow$ FOLLOWER;
62     cballot $\leftarrow b$;
63     txn $\leftarrow txn$;
64     vote $\leftarrow vote$;
65     dec $\leftarrow dec$;
66     phase $\leftarrow phase$;

67 function retry($k$)
68     pre: phase[$k$] = PREPARED;
69     send PREPARE(txn[$k$]) to proc(shards(txn[$k$]));
```

The new leader waits until it receives NEW_LEADER_ACK messages from a quorum of shard members (line 44). Based on the states reported by the processes, it computes a new state from which to start certifying transactions. Like in Paxos, the leader focusses on the states of processes that reported the maximal cballot (line 46): if the k-th transaction is PREPARED at such a process, then the leader marks it as accepted and copies the vote; furthermore, if the transaction is DECIDED at some process (with any ballot number), then the leader marks it as decided and copies the final decision. Given Invariant 2, we can show that the resulting certification order does not have holes: if a transaction is PREPARED or DECIDED, then so are the previous transactions in the certification order.

The leader sets next to the length of the merged sequence of transactions, cballot to the new ballot and status to LEADER, which allows it to start processing new transactions (lines 55-57). It then sends a NEW_STATE message to other shard members, containing the new state (line 58). Upon receiving this message (line 59), a process overwrites its state with the one provided, changes its status to FOLLOWER, and sets cballot to b, thereby recording the fact that it has synchronised with the leader of b. Note that the process will not accept transactions from the new leader until it receives the NEW_STATE message. This ensures that Invariant 2 is preserved when the process receives the first ACCEPT message in the new ballot.

Coordinator recovery. If a process that accepted a transaction t does not receive the final decision on it, this may be because the coordinator of t has failed. In this case the process may decide to become a new coordinator by executing the retry function (line 67). For this the process just re-sends the PREPARE(t) message to the shards of t. A leader handles the PREPARE(t) message received from another process p_j similarly to one received from a client. If it has already certified the transaction t, it re-sends the corresponding ACCEPT message to the shard members, asking them to reply to p_j (line 14). Otherwise, it handles t as before. In the end, a quorum of processes in each shard will reply to the new coordinator (line 28), which will then broadcast the final decision (lines 30-31). Note that the check at line 14 ensures Invariants 4 and 5: in a given ballot b, a transaction t can only be assigned to a single slot in the certification order, and all transactions in the txn array are distinct.

Our protocol allows any number of processes to become coordinators of a transaction at the same time: unlike in the vanilla protocol of Figure 1b, coordinators are not consistently replicated. Nevertheless, the protocol ensures that they will all reach the same decision, even in case of leader changes. We formalise this in Invariant 6: part (a) ensures an agreement on the decision on the k-th transaction in the certification order at a given shard; part (b) ensures a system-wide agreement on the decision on a given transaction t. The latter part establishes that the fault-tolerant protocol computes a unique decision on each transaction.

By the structure of the hander at line 29, Invariant 6 follows from Invariant 7, since, if a coordinator has computed the final decision on a transaction, then a quorum of processes in each relevant shard has accepted a corresponding vote. We next prove Invariant 7.

Proof of Invariant 7. *(a)* Let us assume that quorums of processes in s have sent ACCEPT_ACK(s, b_1, k, t_1, d_1) and ACCEPT_ACK(s, b_2, k, t_2, d_2). Then ACCEPT($b_1, k, t_1, d_1, \_$) and ACCEPT($b_2, k, t_2, d_2, \_$) have been sent to s. Assume without loss of generality that $b_1 \leq b_2$. If $b_1 = b_2$, then by Invariant 1 we must have $t_1 = t_2$ and $d_1 = d_2$. Assume now that $b_1 < b_2$. By Invariant 3, when leader(b_2) sends the ACCEPT message, it has txn[k] $= t_1$. But then due to the check at line 14, we again must have $t_1 = t_2$ and $d_1 = d_2$.

(b) Assume that quorums of processes in s have sent ACCEPT_ACK(s, b_1, k_1, t, d_1) and ACCEPT_ACK(s, b_2, k_2, t, d_2). Then ACCEPT($b_1, k_1, t, d_1, \_$) and ACCEPT($b_2, k_2, t, d_2, \_$) have been sent to s. Without loss of generality, we can assume $b_1 \leq b_2$. We first show that $k_1 = k_2$. If $b_1 = b_2$, then we must have $k_1 = k_2$ by Invariant 4. Assume now that $b_1 < b_2$. By Invariant 3, when leader(b_2) sends the ACCEPT message, it has txn[k_1] $= t$. But then due to the check at line 14 and Invariant 5, we again must have $k_1 = k_2$. Hence, $k_1 = k_2$. But then by Invariant 7a we must also have $d_1 = d_2$. ◀

Protocol correctness. We only establish the safety of the protocol (in the sense of the correctness condition in §2) and leave guaranteeing liveness to standard means, such as assuming either an oracle that is eventually able to elect a consistent leader in every shard [5], or that the system eventually behaves synchronously for sufficiently long [12].

▶ **Theorem 2.** *The fault-tolerant commit protocol in Algorithms 2–3 simulates the multi-shot 2PC protocol in Algorithm 1.*

We give the proof in [7, §B]. Its main idea is to show that, in an execution of the fault-tolerant protocol, each shard produces a single certification order on transactions from which votes and final decisions are computed. These certification orders determine the desired execution of the multi-shot 2PC protocol. We prove the existence of a single per-shard certification order using Invariant 3, showing that certification orders and votes used to compute decisions persist across leader changes. However, this property does not hold of final decisions, and it is this feature that necessitates adding transitions for forgetting and recalling final decisions to the protocol in Algorithm 1 (lines 21 and 24).

For example, assume that the leader of a ballot b at a shard s receives the decision ABORT on a transaction t. The leader will then take this decision into account in its vote computations, e.g., allowing transactions conflicting with t to commit. However, if the leader fails, a new leader may not find out about the final decision on t if this decision has not yet reached other shard members. This leader will not be able to take the decision into account in its vote computations until it reconstructs the decision from the votes at the relevant shards (line 67). Forgetting and recalling the final decisions in the multi-shot 2PC protocol captures how such scenarios affect vote computations.

Optimisations. Our protocol allows the client and the relevant servers to learn the decision on a transaction in four message delays, including communication with the client (Figure 1c). As in standard Paxos, this can be further reduced to three message delays at the expense of increasing the number of messages sent by eliminating the coordinator: processes can send their `ACCEPT_ACK` messages for a transaction directly to all processes in the relevant shards and to the client. Each process can then compute the final decision independently. The resulting time complexity matches the lower bounds for consensus [24, 6] and non-blocking atomic commit [13].

In practice, the computation of a shard-local function for s depends only on the objects managed by s: e.g., Obj_s for (4) and (5). Hence, once a process at a shard s receives the final decision on a transaction t, it may discard the data of t irrelevant to s. Note that the same cannot be done when t is only prepared, since the complete information about it may be needed to recover from coordinator failure (line 67).

5 Related Work

The existing work on the Atomic Commit Problem (ACP) treats it as a one-shot problem with the votes being provided as the problem inputs. The classic ACP solution is the Two-Phase Commit (2PC) protocol [15], which blocks in the event of the coordinator failure. The *non-blocking* variant of ACP known as Non-Blocking Atomic Commit (NBAC) [38] has been extensively studied in both the distributed computing and database communities [38, 13, 33, 20, 23, 17, 18, 16]. The Three-Phase Commit (3PC) family of protocols [38, 3, 13, 2, 23] solve NBAC by augmenting 2PC with an extra message exchange round in the failure-free case. Paxos Commit [16] and Guerraoui et al. [18] avoid extra message delays by instead replicating the 2PC participants through consensus instances. While our fault-tolerant protocol builds upon similar ideas to optimise the number of failure-free message delays, it nonetheless solves a more general problem (TCS) by requiring the output decisions to be compatible with the given isolation level.

Recently, Guerraoui and Wang [19] have systematically studied the failure-free complexity of NBAC (in terms of both message delays and number of messages) for various combinations of the correctness properties and failure models. The complexity of certifying a transaction in the failure-free runs of our crash fault-tolerant TCS implementation (provided the coordinator is replaced with all-to-all communication) matches the tight bounds for the most robust version of NBAC considered in [19], which suggests it is optimal. A comprehensive study of the TCS complexity in the absence of failures is the subject of future work.

Our multi-shot 2PC protocol is inspired by how 2PC is used in a number of systems [32, 31, 34, 39, 37, 9, 14]. Unlike prior works, we formalise how 2PC interacts with concurrency control in such systems in a way that is parametric in the isolation level provided and give conditions for its correctness, i.e., (6)-(8). A number of systems based on deferred update replication [30] used non-fault-tolerant 2PC for transaction commit [32, 31, 34, 39]. Our formalisation of the TCS problem should allow making them fault-tolerant using protocols of the kind we presented in §4.

Multiple researchers have observed that implementing transaction commit by layering 2PC on top of Paxos is suboptimal and proposed possible solutions [41, 26, 28, 37, 11]. In comparison to our work, they did not formulate a stand-alone certification problem, but integrated certification with the overall transaction processing protocol for a particular isolation level and corresponding optimisations.

In more detail, Kraska et al. [26] and Zhang et al. [41] presented sharded transaction processing systems, respectively called MDCC and TAPIR, that aim to minimise the latency of transaction commit in a geo-distributed setting. The protocols used are leaderless: to compute the vote, the coordinator of a transaction contacts processes in each relevant shard directly; if there is a disagreement between the votes computed by different processes, additional message exchanges are needed to resolve it. This makes the worse-case failure-free time complexity of the protocols higher than that of our fault-tolerant protocol. The protocols were formulated for particular isolation levels (a variant of Read Committed in MDCC and serializability in TAPIR). Both MDCC and TAPIR are significantly more complex than our fault-tolerant commit protocol and lack rigorous proofs of correctness.

Sciascia et al. proposed Scalable Deferred Update Replication [37] for implementing serializable transactions in sharded systems. Like the vanilla protocol in §4, their protocol keeps shards consistent using black-box consensus. It avoids executing consensus to persist a final decision by just not taking final decisions into account in vote computations. This solution, specific to their conflict check for serializability, is suboptimal: if a prepared transaction t aborts, it will still cause conflicting transactions to abort until their read timestamp goes above the write timestamp of t.

Dragojević et al. presented a FaRM transactional processing system based on RDMA [11]. Like in our fault-tolerant protocol, in the FaRM atomic commit protocol only shard leaders compute certification votes. However, recovery in FaRM is simplified by the use of leases and an external reconfiguration engine.

Mahmoud et al. proposed Replicated Commit [28], which reduces the latency of transaction commit by layering Paxos on top of 2PC, instead of the other way round. This approach relies on 2PC deciding ABORT only in case of failures, but not because of concurrency control. This requires integrating the transaction commit protocol with two-phase locking and does not allow using it with optimistic concurrency control.

Schiper et al. proposed an alternative approach to implementing deferred update replication in sharded systems [35]. This distributes transactions to shards for certification using genuine atomic multicast [10], which avoids the need for a separate fault-tolerant commit protocol. However, atomic multicast is more expensive than consensus: the best known implementation requires 4 message delays to deliver a message, in addition to a varying convoy effect among different transactions [8]. The resulting overall latency of certification is 5 message delays plus the convoy effect.

Our fault-tolerant protocol follows the primary/backup state machine replication approach in imposing the leader order on transactions certified within each shard. This is inspired by the design of some total order broadcast protocols, such as Zab [22] and Viewstamped Replication [29]. Kokocinski et al. [25] have previously explored the idea of delegating the certification decision to a single leader in the context of deferred update replication. However, they only considered a non-sharded setting, and did not provide full implementation details and a correctness proof. In particular, it is unclear how correctness is maintained under leader changes in their protocol.

6 Conclusion

In this paper we have made the first step towards building a theory of distributed transaction commit in modern transaction processing systems, which captures interactions between atomic commit and concurrency control. We proposed a new problem of transaction certification service and an abstract protocol solving it among reliable processes. From this, we have systematically derived a provably correct optimised fault-tolerant protocol.

For conciseness, in this paper we focussed on transaction processing systems using optimistic concurrency control. We hope that, in the future, our framework can be generalised to systems that employ pessimistic concurrency control or a mixture of the two. The simple and leader-driven nature of our optimised protocol should also allow porting it to the Byzantine fault-tolerant setting by integrating ideas from consensus protocols such as PBFT [4].

References

1 Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ANSI SQL isolation levels. In *Conference on Management of Data (SIGMOD)*, 1995.

2 Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., 1986.

3 Andrea J. Borr. Transaction monitoring in ENCOMPASS: reliable distributed transaction processing. In *International Conference on Very Large Data Bases (VLDB)*, 1981.

4 Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 1999.

5 Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4), 1996.

6 Bernadette Charron-Bost and André Schiper. Uniform consensus is harder than consensus. *J. Algorithms*, 51(1), 2004.

7 Gregory Chockler and Alexey Gotsman. Multi-shot distributed transaction commit (extended version). *arXiv CoRR*, 1808.00688, 2018. Available from http://arxiv.org/abs/1808.00688.

8 Paulo R. Coelho, Nicolas Schiper, and Fernando Pedone. Fast atomic multicast. In *Conference on Dependable Systems and Networks (DSN)*, 2017.

9 James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

10 Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4), 2004.

11 Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *Symposium on Operating Systems Principles (SOSP)*, 2015.

12 Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2), 1988.

13 Cynthia Dwork and Dale Skeen. The inherent cost of nonblocking commitment. In *Symposium on Principles of Distributed Computing (PODC)*, 1983.

14 Lisa Glendenning, Ivan Beschastnikh, Arvind Krishnamurthy, and Thomas Anderson. Scalable consistency in Scatter. In *Symposium on Operating Systems Principles (SOSP)*, 2011.

15 Jim Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, 1978.

16 Jim Gray and Leslie Lamport. Consensus on transaction commit. *ACM Trans. Database Syst.*, 31(1), 2006.

17 Rachid Guerraoui. Revisiting the relationship between non-blocking atomic commitment and consensus. In *Workshop on Distributed Algorithms (WDAG)*, 1995.

18 Rachid Guerraoui, Mikel Larrea, and André Schiper. Reducing the cost for non-blocking in atomic commitment. In *International Conference on Distributed Computing Systems (ICDCS)*, 1996.

19 Rachid Guerraoui and Jingjing Wang. How fast can a distributed transaction commit? In *Symposium on Principles of Database Systems (PODS)*, 2017.

20 V. Hadzilacos. On the relationship between the atomic commitment and consensus problems. In *Asilomar Workshop on Fault-Tolerant Distributed Computing*, 1990.

21 Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3), 1990.

22 Flavio Paiva Junqueira, Benjamin C. Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *Conference on Dependable Systems and Networks (DSN)*, 2011.

23 Idit Keidar and Danny Dolev. Increasing the resilience of atomic commit at no additional cost. In *Symposium on Principles of Database Systems (PODS)*, 1995.

24 Idit Keidar and Sergio Rajsbaum. A simple proof of the uniform consensus synchronous lower bound. *Inf. Process. Lett.*, 85(1), 2003.

25 Maciej Kokocinski, Tadeusz Kobus, and Pawel T. Wojciechowski. Make the leader work: Executive deferred update replication. In *Symposium on Reliable Distributed Systems (SRDS)*, 2014.

26 Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. MDCC: Multi-data center consistency. In *European Conference on Computer Systems (EuroSys)*, 2013.

27 Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2), 1998.

28 Hatem Mahmoud, Faisal Nawab, Alexander Pucher, Divyakant Agrawal, and Amr El Abbadi. Low-latency multi-datacenter databases using replicated commit. *Proc. VLDB Endow.*, 6(9), 2013.

29 Brian M. Oki and Barbara H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Symposium on Principles of Distributed Computing (PODC)*, 1988.

30 Fernando Pedone, Rachid Guerraoui, and André Schiper. The database state machine approach. *Distributed and Parallel Databases*, 14(1), 2003.

31 Sebastiano Peluso, Paolo Romano, and Francesco Quaglia. Score: A scalable one-copy serializable partial replication protocol. In *International Middleware Conference (Middleware)*, 2012.

32 Sebastiano Peluso, Pedro Ruivo, Paolo Romano, Francesco Quaglia, and Luís E. T. Rodrigues. GMU: genuine multiversion update-serializable partial data replication. *IEEE Trans. Parallel Distrib. Syst.*, 27(10), 2016.

33 K. V. S. Ramarao. Complexity of distributed commit protocols. *Acta Informatica*, 26(6), 1989.

34 Masoud Saeida Ardekani, Pierre Sutra, and Marc Shapiro. G-DUR: A middleware for assembling, analyzing, and improving transactional protocols. In *International Middleware Conference (Middleware)*, 2014.

35 Nicolas Schiper, Pierre Sutra, and Fernando Pedone. P-store: Genuine partial replication in wide area networks. In *Symposium on Reliable Distributed Systems (SRDS)*, 2010.

36 Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4), 1990.

37 Daniele Sciascia, Fernando Pedone, and Flavio Junqueira. Scalable deferred update replication. In *Conference on Dependable Systems and Networks (DSN)*, 2012.

38 Dale Skeen. Nonblocking commit protocols. In *Conference on Management of Data (SIGMOD)*, 1981.

39 Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Symposium on Operating Systems Principles (SOSP)*, 2011.

40 Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery.* Morgan Kaufmann Publishers Inc., 2001.

41 Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. Building consistent transactions with inconsistent replication. In *Symposium on Operating Systems Principles (SOSP)*, 2015.

42 Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. When is operation ordering required in replicated transactional storage? *IEEE Data Eng. Bull.*, 39(1), 2016.