Distributed Computing
https://doi.org/10.1007/s00446-021-00389-4

Multi-shot distributed transaction commit

Gregory Chockler[1] · **Alexey Gotsman**[2]

Received: 16 February 2019 / Accepted: 28 January 2021
© The Author(s), under exclusive licence to Springer-Verlag GmbH, DE part of Springer Nature 2021

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.

Keywords Atomic commit problem · Two-phase commit · Concurrency control · Transactions · Data replication · Paxos

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 [8,15,42,51]. 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 ensuring that the global histories comply with the desired isolation level. In a typical implementation, each shard keeps track of the local objects accessed by a transaction, and determines its vote (commit or abort) on the transaction outcome 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 [2,19,26,46].

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 [17] for solving ACP among reliable processes. A transaction processing system typically executes a 2PC instance for each transaction [39,40,42,47]. 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

This article is a revised and expanded version of a paper that received a Best Paper Award at the 32nd International Symposium on Distributed Computing (DISC).

✉ Alexey Gotsman
software@imdea.org

[1] University of Surrey, Guildford GU2 7XH, UK

[2] IMDEA Software Institute, Campus Montegancedo s/n, Pozuelo de Alarcón, 28223 Madrid, Spain

Published online: 24 March 2021

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, Sect. 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* (Sect. 3). It formalises how the classical 2PC interacts with concurrency control in many practical transaction processing systems [39,40,42,47] 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 (Sect. 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 [8,15,19,21,51]. Similarly to recent work [29,50], our implementation optimises the time complexity of this scheme by weaving 2PC and Paxos together. In particular, our protocol avoids consistently replicating the 2PC coordinator in exchange for removing from the coordinator the ability to unilaterally abort the transaction. This, in its turn, allows eliminating the Paxos consensus required in the vanilla fault-tolerant 2PC to persist the final decision on a transaction at a shard: in case of failures decisions can be reconstructed from the current state of the shards. In contrast to previous protocols following the same design, 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 [6,27] and non-blocking atomic commit [13].

A feature of our protocol that makes it particularly subtle to prove its correctness is avoiding the consensus on the final decision on a transaction and instead propagating the decision to the relevant shard replicas asynchronously. This means that different shard replicas may receive the decision 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 [25,37], this *passive replication* approach requires care in the design of recovery from leader failures and arguing its correctness.

2 Transaction certification service

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

In this paper we focus on transactional processing systems using (backward-oriented) 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* $R(t) \subseteq \mathsf{Obj} \times \mathsf{Ver}$: the set of objects with their versions that t read, which contains one version per object.
- *Write set* of $W(t) \subseteq \mathsf{Obj} \times \mathsf{Val}$: the set of objects with their values that t wrote, which contains 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 parameterized by 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 [49]: $f(T, t) = \textsc{commit}$ iff none of the versions read by t have been overwritten by a transaction in T, i.e.,

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

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

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

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

Histories We represent TCS executions using *histories*—finite 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 consecutive pairs of `certify` and matching `decide` actions. A transaction t *commits* in a history h if h contains $\texttt{decide}(t, \textsc{commit})$. 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 [24] is a sequential history such that: (i) h and ℓ contain the same actions; and (ii)

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

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:

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

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.

Using the TCS specification A correct TCS can be readily used in a transaction processing system (TPS). For example, consider the domain of transactions defined earlier. A typical TPS based on optimistic concurrency control will ensure that transactions submitted for certification read versions that already exist in the database. Formally, this means that for every history h induced by the actions of the TCS interface, it holds: if t is a transaction such that $\texttt{certify}(t)$ occurs in h, and $(x, v) \in R(t)$, then there exists a transaction t' such that $(x, \_) \in W(t') \wedge V_c(t') = v$, and h contains $\texttt{decide}(t', \textsc{commit})$ before $\texttt{certify}(t)$. It is easy to see that this implies that if h is correct with respect to the certification function for serializability (2), then ordering committed transactions as prescribed by a legal linearization of h results in a correct serial history: the value of every object read by a transaction t is the one written by the latest preceding transaction writing to that object. Hence, TCS correct with respect to certification function (2) can indeed be used to implement a serializabile TPS.

3 Multi-shot 2PC and shard-local certification functions

We now present a multi-shot version of the classical *two-phase commit (2PC)* protocol [17], 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 there is a one-to-one mapping between the shards and the processes. We sometimes apply proc to a set of shards to get the set of processes managing them.

Protocol: common case We give the pseudocode of the protocol in Fig. 1 and illustrate its message flow in Fig. 2a. Each handler in Fig. 1 is executed atomically.

To certify a transaction t, a client sends it in a PREPARE message to the relevant shards (line 6)[1]. 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, 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 (Sect. 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. Note that we do not allow the coordinator to unilaterally abort a transaction: this allows minimising latency in a fault-tolerant version of the protocol we present in Sect. 4.

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

```
1  next ← −1 ∈ ℤ;
2  txn[ ] ∈ ℕ → 𝒯;
3  vote[ ] ∈ ℕ → {COMMIT, ABORT};
4  dec[ ] ∈ ℕ → {COMMIT, ABORT};
5  phase[ ] ← (λk. START) ∈ ℕ →
     {START, PREPARED, 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_{s0}({txn[k] | k < next ∧ phase[k] =
               DECIDED ∧ dec[k] = COMMIT}, t) ⊓
         g_{s0}({txn[k] | k < next ∧ phase[k] =
               PREPARED ∧ vote[k] = COMMIT}, t);
12     phase[next] ← PREPARED;
13     send PREPARE_ACK(s0, next, t, vote[next]) to
         coord(t);

14 when received PREPARE_ACK(s, pos_s, t, d_s) for
   every s ∈ shards(t)
15     send DECISION(t, ⊓_{s∈shards(t)} d_s) to
         client(t);
16     forall s ∈ shards(t) do
17         send DECISION(pos_s, ⊓_{s∈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 ∈ ℕ
22     pre: phase[k] = DECIDED;
23     phase[k] ← PREPARED;

24 non-deterministically for some k ∈ ℕ
25     pre: phase[k] ≠ START;
26     send PREPARE_ACK(s0, k, txn[t], vote[k]) to
         coord(t);
```

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

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 Sect. 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 Sect. 2 and assume that the set of objects Obj is partitioned among shards: $\mathsf{Obj} = \biguplus_{s \in \mathcal{S}} \mathsf{Obj}_s$. Then one possible way to define the shard-local certification functions for serializabil-

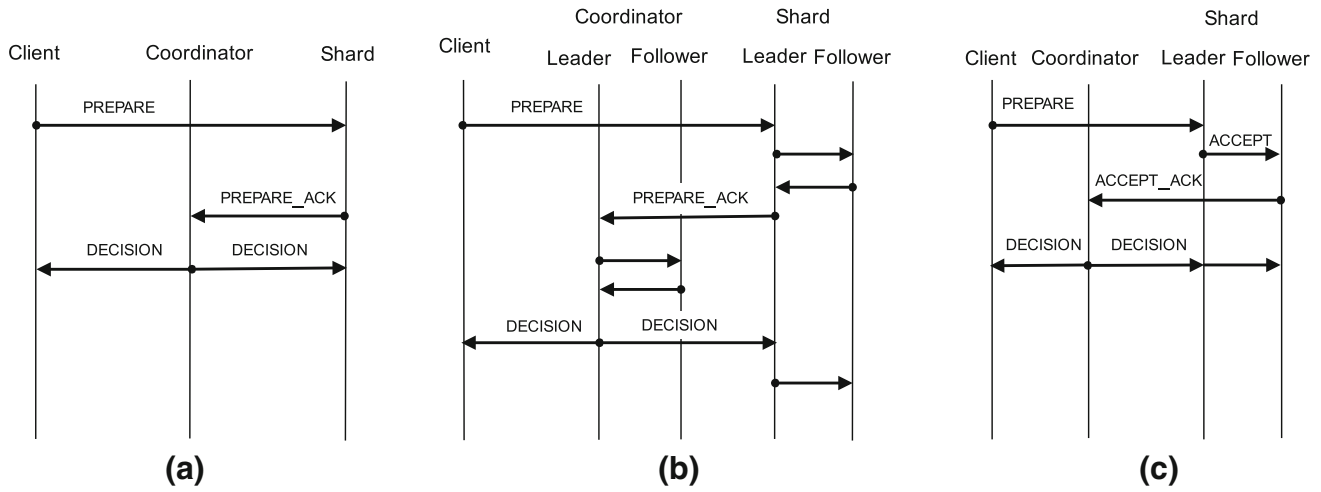

Fig. 2 Message flow diagrams illustrating the behaviour of **a** multi-shot 2PC; **b** multi-shot 2PC with shards replicated using Paxos; **c** optimised protocol weaving together multi-shot 2PC and Paxos

ity is as follows: $f_s(T, t) = \text{COMMIT}$ iff

$$\begin{aligned} &\forall x \in \mathsf{Obj}_s.\, \forall v.\, (x, v) \in R(t) \implies \\ &\quad (\forall t' \in T.\, (x, \_) \in W(t') \implies V_c(t') \leq v), \end{aligned} \tag{4}$$

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

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

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 [45]. Consider a transaction t_1 that reads versions v_1 and v_2 of x and y respectively, and wants to write y with a commit version $v_2' > v_2$, and a transaction t_2 that reads the same versions v_1 and v_2 of x and y, and wants to write x with a commit version $v_1' > v_1$. Suppose that x and y are stored at two distinct shards s_1 and s_2 respectively. 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 [39,40,42,45,47]. 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

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

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.

We next formulate conditions sufficient to ensure that shard-local certification functions correctly approximate a given global function f. 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

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

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

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

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

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 Fig. 1 has two additional handlers at lines 21 and 24, executed non-deterministically. As we show in Sect. 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 establishes the correctness of multi-shot 2PC provided the shard-local concurrency control given by f_s and g_s is related to the shard-agnostic concurrency control given by a global certification function f as prescribed by (6)–(8).

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

To facilitate the proof of Theorem 1, we first introduce a low-level specification TCS-LL1, and prove that it is correctly implemented by multi-shot 2PC (Lemma 1). We then show that every history satisfying TCS-LL1 is correct with respect to f (Lemma 2). The structure of all our proofs is illustrated

$$\forall t.\, d[t] = \bigsqcap \{d_s[t] \mid s \in \mathsf{shards}(t)\} \quad (9)$$

$$\forall t, s.\, (pos_s[t] \text{ is defined}) \implies \forall j < pos_s[t].\, \exists t'.\, pos_s[t'] = j \quad (10)$$

$$\forall t_1, t_2, s.\, t_1 \neq t_2 \implies pos_s[t_1] \neq pos_s[t_2] \quad (11)$$

$$\forall t, s.\, d_s[t] = f_s(T_s[t], t) \sqcap g_s(P_s[t], t) \quad (12)$$

$$\forall t, s.\, T_s[t] = \{t' \mid pos_s[t'] < pos_s[t] \wedge d[t'] = \text{COMMIT}\} \setminus P_s[t] \quad (13)$$

$$\forall t, s.\, P_s[t] \subseteq \{t' \mid pos_s[t'] < pos_s[t] \wedge d_s[t'] = \text{COMMIT}\} \quad (14)$$

$$\forall t, t', s.\, t' \sqsubset_{\text{rt}} t \wedge s \in \mathsf{shards}(t') \cap \mathsf{shards}(t) \implies pos_s[t'] < pos_s[t] \quad (15)$$

$$\sqsubset_{\text{rt}} \cup \sqsubset_{\text{dec}} \text{ is acyclic}, \quad (16)$$

where

$$t' \sqsubset_{\text{rt}} t \iff \texttt{decide}(t', \_) \prec_h \texttt{certify}(t)$$

$$t' \sqsubset_{\text{dec}} t \iff \exists s.\, t' \in T_s[t] \vee (pos_s[t'] < pos_s[t] \wedge\ d_s[t'] = \text{COMMIT} \wedge d[t'] = \text{ABORT} \wedge t' \notin P_s[t])$$

Fig. 3 Constraints on the votes and decisions computed by the multi-shot 2PC protocol

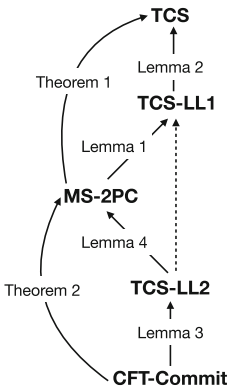

Fig. 4 High-level structure of the correctness proofs. An arrow from X to Y means that X is a refinement of Y. MS-2PC stands for the multi-shot 2PC protocol in Fig. 1, and CFT-Commit for the fault-tolerant commit protocol in Fig. 5

in Fig. 4. The low-level specification TCS-LL1 is defined as follows.

Consider a history h. Let T denote the set of transactions t such that $\texttt{certify}(t)$ is an event in h, and $d[t]$ denote the decision value d of $t \in T$ if $\texttt{decide}(t, d)$ is an event in h. The history h satisfies TCS-LL1 if for some of transactions $t \in T$ and shards $s \in \mathsf{shards}(t)$ there exist $d_s[t] \in \mathcal{D}$, $pos_s[t] \in \mathbb{N}$ and $T_s[t]$, $P_s[t] \in 2^{\mathcal{T}}$ such that all the constraints in Fig. 3 are satisfied. A

protocol is a correct implementation of TCS-LL1 if each of its finite histories satisfies TCS-LL1.

The conditions in Fig. 3 can be intuitively understood from the proof sketch of the following easy lemma, which connects them to the pseudocode in Fig. 1.

Lemma 1 *The multi-shot 2PC protocol in Fig. 1 is a correct implementation of* TCS-LL1.

Proof sketch We informally explain how to select the required $d_s[t] \in \mathcal{D}$, $pos_s[t] \in \mathbb{N}$ and $T_s[t]$, $P_s[t] \in 2^{\mathcal{T}}$. Fix a finite execution of the protocol producing a history h. Let T be the set of transactions t such that $\texttt{certify}(t)$ is an event in h. For some of transactions $t \in T$ and shards $s \in \mathsf{shards}(t)$ we define the certification order position $pos_s[t]$ and the vote $d_s[t]$ computed by the protocol for shard s as follows:

Consider $t \in T$ and $s \in \mathsf{shards}(t)$. Assume the process $\mathsf{proc}(s)$ sent $\texttt{PREPARE\_ACK}(s, k, t, d)$ and right after this it had $\mathsf{txn} = txn$ and $\mathsf{vote} = vote$. Then for every $j \leq k$ we let $pos_s[txn[j]] = j$ and $d_s[txn[j]] = vote[j]$.

It is easy to see that this defines $pos_s[t]$ and $d_s[t]$ uniquely. Furthermore, by the structure of the handler at line 14 in Fig. 1, for each t such that $\texttt{decide}(t, d[t])$ occurs in h, $d_s[t]$ is defined for all $s \in \mathsf{shards}(t)$ and (9) holds. It is also easy to check that the order determined by pos_s is down-closed (10) and distinct transactions are assigned to distinct positions by pos_s (11). By the structure of the vote computation at line 11, for a transaction t and a shard $s \in \mathsf{dest}(t)$, constraints (12)–(14) are satisfied for some sets of transactions $T_s[t]$ and $P_s[t]$, determining transactions respectively considered committed and pending when computing the vote on t at s. The relation $\sqsubset_{\text{rt}}$ lifts the real-time order between certification requests to transactions certified. The relation $t' \sqsubset_{\text{dec}} t$ means that the final decision on transaction t' is taken into account when computing a vote on a transaction t. The relation $\sqsubset_{\text{rt}} \cup \sqsubset_{\text{dec}}$ is acyclic (16), because if $t' \sqsubset_{\text{rt}} t$ or $t' \sqsubset_{\text{dec}} t$, then a $\texttt{DECISION}(t', \_)$ message must be sent before a $\texttt{DECISION}(t, \_)$ message can be sent. □

Theorem 1 follows from Lemma 1 and the following lemma.

Lemma 2 *If shard-local certification functions f_s and g_s satisfy (6)–(8), then every history satisfying* TCS-LL1 *is correct with respect to f.*

Proof We construct a linearization of $h \mid \mathsf{committed}(h)$ by arranging committed certification requests in h in any order consistent with $\sqsubset_{\text{rt}} \cup \sqsubset_{\text{dec}}$. By (16) at such an order exists. To prove the lemma, it is enough to show that the resulting linearization is legal with respect to f. We do this by induction on its length $k \geq 0$. This trivially holds when $k = 0$.

Assume now that the linearization ℓ_k of length k is legal and consider a linearization ℓ_{k+1} of length $k + 1$.

For all $i = 1..(k + 1)$, let $\texttt{decide}(t_i, \text{COMMIT})$ be the i-th $\texttt{decide}$ action in ℓ_k. Let $T_0 = \emptyset$, and for all $j = 1..k$ let $T_j = T_{j-1} \cup \{t_j\}$. By the induction hypothesis, $f(T_{i-1}, t_i) = \text{COMMIT}$ for all $i = 1..k$. We show that $f(T_k, t_{k+1}) = \text{COMMIT}$, which implies that ℓ_{k+1} is legal with respect to f.

Let

$$T = \{t_i \mid \neg(\texttt{decide}(t_i, \_) \prec_h \texttt{certify}(t_{k+1})) \wedge 1 \leq i \leq k\}$$

and $T' = T_k \setminus T$. Fix a shard $s \in \mathsf{shards}(t_{k+1})$. Let

$$U_0 = \{t' \mid pos_s[t'] < pos_s[t_{k+1}]\} \cap (T_k \mid s).$$

Then

$$U_0 \subseteq \{t' \mid pos_s[t'] < pos_s[t_{k+1}] \wedge d[t'] = \text{COMMIT}\}. \quad (17)$$

We have $d[t_{k+1}] = \text{COMMIT}$, which by (9) implies $d_s[t_{k+1}] = \text{COMMIT}$. Then by (12) we have

$$f_s(T_s[t_{k+1}], t_{k+1}) \sqcap g_s(P_s[t_{k+1}], t_{k+1}) = d_s[t_{k+1}] = \text{COMMIT},$$

From (17), (13) and (14), we get $U_0 \subseteq T_s[t_{k+1}] \cup P_s[t_{k+1}]$. Then by (7) and (1), we have

$$f_s(U_0, t_{k+1}) = \text{COMMIT}. \quad (18)$$

By (15), $(T' \mid s) \subseteq U_0$. We now augment U_0 with the transactions in $(T \mid s) \setminus U_0$ as follows. Let $t^1, \ldots, t^m$ where $m = |(T \mid s) \setminus U_0|$ be the set of transactions in $(T \mid s) \setminus U_0$ sorted according to pos_s. Consider a sequence of sets $U_0, \ldots, U_m$ such that $U_i = U_{i-1} \cup \{t^i\}$ for $i = 1..m$. We now prove that $f_s(U_i, t_{k+1}) = \text{COMMIT}$ for $i = 0..m$ by induction on i.

First, note that (18) implies that the claim is true for $i = 0$. Next, assume that $f_s(U_j, t_{k+1}) = \text{COMMIT}$ for all $j = 0..(i - 1)$, and consider $f_s(U_i, t_{k+1})$ where $U_i = U_{i-1} \cup \{t^i\}$. Since t^i comes before t_{k+1} in the linearization, we cannot have $t_{k+1} \sqsubset_{\text{dec}} t^i$. Hence, we must have $t_{k+1} \in P_s[t^i]$ and, by (1), $g_s(\{t_{k+1}\}, t^i) = \text{COMMIT}$. By (8), this implies $f_s(\{t^i\}, t_{k+1}) = \text{COMMIT}$. From this and the induction hypothesis, by (1) we get

$$f_s(U_i, t_{k+1}) = f_s(U_{i-1} \cup \{t^i\}, t_{k+1}) = \text{COMMIT},$$

which concludes the proof of the induction step.

We have established $f_s(U_m, t_{k+1}) = \text{COMMIT}$. Since $(T_k \mid s) = ((T \cup T') \mid s) = U_m$, this implies $f_s((T_k \mid s), t_{k+1}) =$

COMMIT. Thus, for all $s \in \mathsf{shards}(t_{k+1})$ we have $f_s((T_k \mid s), t_{k+1}) = \text{COMMIT}$, which by (6) implies $f(T_k, t_{k+1}) = \text{COMMIT}$, as required. □

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 $2k+1$ processes, out of which at most k can fail. We call a set of $k+1$ 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 [44] to make a shard simulate a reliable process in multi-shot 2PC; this is usually based on a consensus protocol such as Paxos [30]. 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 [8,15,51], multiple researchers have observed that the resulting protocol requires an unnecessarily high number of message delays [29,31,50]. Namely, every action of multi-shot 2PC in Fig. 2a 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 Fig. 2b. 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 Fig. 5 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 Fig. 2c and summarise the key invariants used in its proof of correctness in Fig. 6.

A process maintains the same variables as in the multi-shot 2PC protocol (Fig. 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. We use the whole set of integers as ballots because a given process may play the role of a leader multiple times. At any given time, a process participates in a single ballot, which is stored in a variable $\mathsf{cballot}$ and never decreases. During leader changes we also use an additional ballot variable ballot.

Unlike the vanilla protocol illustrated in Fig. 2b, 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 (Sect. 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 15).

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 17), 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 23). 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 Fig. 6: 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 25). 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

```
1  next ← −1 ∈ ℤ;
2  txn[ ] ∈ ℕ → 𝒯;
3  vote[ ] ∈ ℕ → {COMMIT, ABORT};
4  dec[ ] ∈ ℕ → {COMMIT, ABORT};
5  phase[ ] ← (λk. START) ∈ ℕ →
     {START, PREPARED, DECIDED};
6  status ∈ {LEADER, FOLLOWER, RECOVERING};
7  cballot ← 0 ∈ ℕ;
8  ballot ← 0 ∈ ℕ;

9  function certify(t)
10   send PREPARE(t) to proc(shards(t));

11 when received PREPARE(t) from p_j or a client
12   pre: status = LEADER;
13   if received from a process p_j then p ← p_j;
14   else p ← coord(t);
15   if ∃k. t = txn[k] then
16     send ACCEPT(cballot, k, t, vote[k], p) to
         proc(s_0)
17   else
18     next ← next + 1;
19     txn[next] ← t;
20     vote[next] ←
         f_{s_0}({txn[k] | k < next ∧ phase[k] =
21             DECIDED ∧ dec[k] = COMMIT}, t) ⊓
         g_{s_0}({txn[k] | k < next ∧ phase[k] =
               PREPARED ∧ vote[k] = COMMIT}, t);
22     phase[next] ← PREPARED;
23     send ACCEPT(cballot, next, t, vote[next], p)
         to proc(s_0);

24 when received ACCEPT(b, k, t, d, p)
25   pre: status ∈ {LEADER, FOLLOWER} ∧
           cballot = b;
26   if phase[k] = START then
27     txn[k] ← t;
28     vote[k] ← d;
29     phase[k] ← PREPARED;
30   send ACCEPT_ACK(s_0, b, k, t, d) to p;

31 when for every s ∈ shards(t) received a
   quorum of ACCEPT_ACK(s, b_g, pos_g, t, d_g)
32   send DECISION(t, ⨅_{s∈shards(t)} d_s) to client(t);
33   forall s ∈ shards(t) do
34     send DECISION(b_s, pos_s, ⨅_{s∈shards(t)} d_s) to
         proc(s)

35 when received DECISION(b, k, d)
36   pre: status ∈ {LEADER, FOLLOWER} ∧
           cballot ≥ b ∧ phase[k] = PREPARED;
37   dec[k] ← d;
38   phase[k] ← DECIDED;

39 function recover()
40   send NEW_LEADER(any ballot b such that
       b > ballot ∧ leader(b) = p_i) to proc(s_0);

41 when received NEW_LEADER(b) from p_j
42   pre: b > ballot;
43   status ← RECOVERING;
44   ballot ← b;
45   send NEW_LEADER_ACK(ballot, cballot, txn, vote,
       dec, phase) to p_j;

46 when received {NEW_LEADER_ACK
     (b, cballot_j, txn_j, vote_j, dec_j, phase_j) | p_j ∈ Q}
   from a quorum Q in s_0
47   pre: status = RECOVERING ∧ ballot = b;
48   reinitialise phase, txn, vote, dec;
49   var J ← the set of j with the maximal
           cballot_j;
50   forall k do
51     if ∃j ∈ J. phase_j[k] ≥ PREPARED then
52       txn[k] ← txn_j[k];
53       vote[k] ← vote_j[k];
54       phase[k] ← PREPARED;
55     if ∃j. phase_j[k] = DECIDED then
56       dec ← dec_j[k];
57       phase[k] ← DECIDED;
58   next ← min{k | phase[k] = START} − 1;
59   cballot ← b;
60   status ← LEADER;
61   send NEW_STATE(b, txn, vote, dec, phase) to
       proc(s_0) \ {p_i};

62 when received
   NEW_STATE(b, txn, vote, dec, phase) from p_j
63   pre: b ≥ ballot;
64   status ← FOLLOWER;
65   cballot ← b;
66   txn ← txn;
67   vote ← vote;
68   dec ← dec;
69   phase ← phase;

70 function retry(k)
71   pre: phase[k] = PREPARED;
72   send PREPARE(txn[k]) to proc(shards(txn[k]));
```

Fig. 5 Fault-tolerant commit protocol at a process p_i in a shard s_0

1. If $\texttt{ACCEPT}(b, k, t_1, d_1, \_)$ and $\texttt{ACCEPT}(b, k, t_2, d_2, \_)$ messages are sent to the same shard, then $t_1 = t_2$ and $d_1 = d_2$.
2. After a process receives and acknowledges $\texttt{ACCEPT}(b, k, t, d, \_)$, we have $\mathsf{txn} = txn\downarrow_k$ and $\mathsf{vote} = vote\downarrow_k$, 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 $\texttt{ACCEPT}(b, k, t, d, \_)$ and responded to it with $\texttt{ACCEPT\_ACK}(s, b, k, t, d)$, and at the time $\mathsf{leader}(b)$ sent $\texttt{ACCEPT}(b, k, t, d, \_)$ it had $\mathsf{txn}\downarrow_k = txn$ and $\mathsf{vote}\downarrow_k = vote$. Whenever at a process in s we have $\mathsf{cballot} = b' > b$, we also have $\mathsf{txn}\downarrow_k = txn$ and $\mathsf{vote}\downarrow_k = vote$.
4. At any process, we always have $\mathsf{cballot} \leq \mathsf{ballot}$.
5. If $\texttt{ACCEPT}(b, k_1, t, \_, \_)$ and $\texttt{ACCEPT}(b, k_2, t, \_, \_)$ messages are sent to the same shard, then $k_1 = k_2$.
6. At any process, all transactions in the txn array are distinct.
7. (a) For any messages $\texttt{DECISION}(\_, k, d_1)$ and $\texttt{DECISION}(\_, k, d_2)$ sent to processes in the same shard, we have $d_1 = d_2$.
   (b) For any messages $\texttt{DECISION}(t, d_1)$ and $\texttt{DECISION}(t, d_2)$ sent, we have $d_1 = d_2$.
8. If $\mathsf{txn}[k]$ is defined, then $\mathsf{phase}[k] \geq$ PREPARED.
9. (a) Assume that a quorum of processes in s have sent $\texttt{ACCEPT\_ACK}(s, b_1, k, t_1, d_1)$ and a quorum of processes in s have sent $\texttt{ACCEPT\_ACK}(s, b_2, k, t_2, d_2)$. Then $t_1 = t_2$ and $d_1 = d_2$.
   (b) Assume that a quorum of processes in s have sent $\texttt{ACCEPT\_ACK}(s, b_1, k_1, t, d_1)$ and a quorum of processes in s have sent $\texttt{ACCEPT\_ACK}(s, b_2, k_2, t, d_2)$. Then $k_1 = k_2$ and $d_1 = d_2$.
10. If at a process in a shard s we have $\mathsf{ballot} = b'$, $\mathsf{phase}[k] =$ DECIDED and $\mathsf{dec}[k] = d$, then a $\texttt{DECISION}(\_, k, d)$ message has been sent to s, where $b \leq b'$.
11. If at a process we have $\mathsf{phase}[k] =$ DECIDED and $\mathsf{dec}[k] =$ COMMIT, then $\mathsf{vote}[k] =$ COMMIT.

Fig. 6 Key invariants of the fault-tolerant protocol. We let $\alpha\downarrow_k$ be the prefix of the sequence α of length k

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 31). 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 35), the process stores it and advances the transaction's phase to DECIDED. Note that, as in the multi-shot 2PC protocol in Sect. 3, the coordinator cannot unilaterally abort a transaction.

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 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, a new leader is elected in two stages (similarly to [25,37]). First, processes vote to join the ballot of a prospective leader, which they record in a variable ballot; like cballot, this variable can only increase. Then, before normal processing can resume, followers receive and accept an initial state from the new leader and set cballot to $ballot$. We thus always have $\mathsf{cballot} \leq \mathsf{ballot}$, as stated by Invariant 4.

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 39). The process picks a ballot that it leads higher than the last ballot it joined and sends it in a NEW_LEADER message to the shard members (including itself); this message asks the group members to support the process as the new leader and is analogous to the "1a" message in Paxos. When a process receives a $\texttt{NEW\_LEADER}(b)$ message (line 41), it first checks that the proposed ballot b is higher than the last ballot it joined. 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 (due to the guards at lines 12, 25 and 36). It then replies to the new leader with a NEW_LEADER_ACK message containing all components of its state; this message serves as a vote for the new leader and is analogous to the "1b" message of Paxos.

The new leader waits until it receives NEW_LEADER_ACK messages from a quorum of shard members (line 46). 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 49): if the k-th transac-

tion 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 58-60). It then sends a NEW_STATE message to other shard members, containing the new state (line 61). Upon receiving this message (line 62), 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 70). 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 15). Otherwise, it handles t as before. In the end, a quorum of processes in each shard will reply to the new coordinator (line 30), which will then broadcast the final decision (lines 32–33). Note that the check at line 15 ensures Invariants 5 and 6: 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 Fig. 2b, 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 7: 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.

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 (Fig. 2c). 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 [6,27] 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 70).

Protocol correctness We only establish the safety of the protocol (in the sense of the correctness condition in Sect. 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 Fig. 5 simulates the multi-shot 2PC protocol in Fig. 1.*

We give the proof of the theorem below. 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 Fig. 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 70). Forgetting and recalling the final decisions in the multi-shot 2PC protocol captures how such scenarios affect vote computations.

We first prove the nontrivial Invariants 3 and 7 from Fig. 6. The following proof of Invariant 3 relies on a straighfor-

ward auxiliary Invariant 8, where we order phases as follows: START < ACCEPTED < DECIDED.

Proof of Invariant 3. We prove the invariant by induction on b'. Assume that the invariant holds for all $b' < b''$. We now show it for $b' = b''$. Assume that at some point a quorum Q of processes in s have received $\texttt{ACCEPT}(b, k, t, d, \_)$ and responded to it with $\texttt{ACCEPT\_ACK}(s, b, k, t, d)$, and at the time $\mathsf{leader}(b)$ sent $\texttt{ACCEPT}(b, k, t, d, \_)$ it had $\mathsf{txn}\downarrow_k = txn$ and $\mathsf{vote}\downarrow_k = vote$. We prove by induction on the length of the protocol execution that, whenever at a process $p_i \in s$ we have $\mathsf{cballot} = b'' > b$, we also have $\mathsf{txn}\downarrow_k = txn$ and $\mathsf{vote}\downarrow_k = vote$. The property holds trivially at the start of the execution, since at this time at all processes we have $\mathsf{cballot} = \bot < b$. The only transition that can nontrivially affect the validity of the above property is the transition in line 46. Assume that after this transition $\mathsf{cballot} = b'' > b$ at p_i. We show that we also have $\mathsf{txn}\downarrow_k = txn$ and $\mathsf{vote}\downarrow_k = vote$. Assume that during the transition in line 46 the process p_i received messages

$$\texttt{NEW\_LEADER\_ACK}(b'', cballot_j, txn_j, vote_j, dec_j, phase_j)$$

from a quorum Q' of processes p_j. Let $b_0 = \max\{cballot_j \mid p_j \in Q'\}$ and $J = \{j \mid cballot_j = b_0\}$. Then by the check at line 42 and Invariant 4, we have $b_0 < b''$.

We have $Q \cap Q' \neq \emptyset$. Thus, some process process p_{j_0} must have sent both ACCEPT_ACK and NEW_LEADER_ACK messages. Then p_{j_0} must have sent its ACCEPT_ACK message before the NEW_LEADER_ACK message. By the precondition in line 25, when p_{j_0} sent the ACCEPT_ACK message, it must have had $\mathsf{cballot} = b$. Then when p_{j_0} sent its NEW_LEADER_ACK message, it had $\mathsf{cballot} \geq b$. Hence, $cballot_{j_0} \geq b$ and $b_0 \geq b$, so that $b_0 \neq 0$ and $J \neq \emptyset$.

Consider first the case when $b_0 = b$. Since $p_{j_0} \in Q'$ has received and acknowledged $\texttt{ACCEPT}(b, k, t, d, \_)$, by Invariant 2 we have $txn = txn_{j_0}\downarrow_k$ and $vote = vote_{j_0}\downarrow_k$. Furthermore, for any other process p_j such that $j \in J$, if $phase_j[k] \geq$ PREPARED, then $txn_{j_0}\downarrow_k$ is a prefix of txn and $vote_{j_0}\downarrow_k$ is a prefix of $vote$. By Invariant 8 we also have $\forall l \leq k.\, phase_{j_0}[l] \geq$ PREPARED. Hence, right after p_i executes the transition in line 46, we have $txn = \mathsf{txn}\downarrow_k$ and $vote = \mathsf{vote}\downarrow_k$, as required.

Assume now that $b_0 > b$. Consider an arbitrary $j \in J$, so that $cballot(p_j) = b_0$. Then by induction hypothesis we have $txn = txn_j\downarrow_k$ and $vote = vote_j\downarrow_k$. By Invariant 8 we also have $\forall l \leq k.\, phase_j[l] \geq$ PREPARED. Then right after p_i executes the transition in line 46, we have $txn = \mathsf{txn}\downarrow_k$ and $vote = \mathsf{vote}\downarrow_k$, as required. □

We next prove Invariant 7. By the structure of the handler at line 31, this invariant actually follows from Invariant 9, 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 now prove Invariant 9.

Proof of Invariant 9 (a) Assume that quorums of processes in s have sent $\texttt{ACCEPT\_ACK}(s, b_1, k, t_1, d_1)$ and $\texttt{ACCEPT\_ACK}(s, b_2, k, t_2, d_2)$. Then $\texttt{ACCEPT}(b_1, k, t_1, d_1, \_)$ and $\texttt{ACCEPT}(b_2, k, t_2, d_2, \_)$ 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, right before $\mathsf{leader}(b_2)$ sends the ACCEPT message, it has $\mathsf{txn}[k] = t_1$ and $\mathsf{vote}[k] = d_1$. But then due to the check at line 15, we again must have $t_1 = t_2$ and $d_1 = d_2$.

(b) Assume that quorums of processes in s have sent $\texttt{ACCEPT\_ACK}(s, b_1, k_1, t, d_1)$ and $\texttt{ACCEPT\_ACK}(s, b_2, k_2, t, d_2)$. Then $\texttt{ACCEPT}(b_1, k_1, t, d_1, \_)$ and $\texttt{ACCEPT}(b_2, k_2, t, d_2, \_)$ 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 5. Assume now that $b_1 < b_2$. By Invariant 3, right before $\mathsf{leader}(b_2)$ sends the ACCEPT message, it has $\mathsf{txn}[k_1] = t$. But then due to the check at line 15 and Invariant 6, we again must have $k_1 = k_2$. Hence, $k_1 = k_2$. But then by Invariant 9a we must also have $d_1 = d_2$. □

The proof of Theorem 2 relies on auxiliary Invariants 10 and 11, which we prove next.

Proof of Invariant 10 We prove the invariant by induction on the length of the protocol execution. The transitions that may set $\mathsf{phase}[k] =$ DECIDED and thereby affect the validity of the invariant are those at lines 35, 46 and 62. The transition at line 35 trivially preserves the invariant. If a transition at lines 46 or 62 is executed and sets $\mathsf{phase}[k] =$ DECIDED, then the premiss of the invariant must have held earlier at some process with $\mathsf{ballot} \leq b'$. Then the required follows from the induction hypothesis. □

Proof of Invariant 11 Follows from Invariants 10 and 3. □

To prove Theorem 2, we introduce another low-level TCS specification TCS-LL2 (see Fig. 4), which is a strengthening of TCS-LL1, and prove that it is correctly implemented by the fault-tolerant commit protocol in Fig. 5 (Lemma 3). We then show that for every history h satisfying TCS-LL2 there is an execution of the multi-shot 2PC protocol in Fig. 1 producing a history h' that is a permutation of h preserving its real-time order (Lemma 4), which implies Theorem 2. Note that although by Lemma 2, it would have been sufficient to argue that our fault-tolerant commit protocol implements TCS-LL1 to establish its correctness, a refined specification TCS-LL2 allows us to obtain a stronger result, namely, that it simulates multi-shot 2PC. The low-level specification TCS-LL2 is defined as follows.

Consider a history h. Let T denote the set of transactions t such that $\texttt{certify}(t)$ is an event in h, and $d[t]$ denote the decision value of $t \in T$ if $\texttt{decide}(t, d)$ is an event in h. The history h satisfies TCS-LL2 if for some of transactions $t \in T$ and shards $s \in \mathsf{shards}(t)$ there exist $d_s[t] \in \mathcal{D}$, $pos_s[t] \in \mathbb{N}$ and $T_s[t]$, $P_s[t] \in 2^{\mathcal{T}}$ such that all the constraints in Fig. 3 are satisfied and the relation $<_V$ over $V = \{(s, t) \mid d_s[t] \text{ is defined}\}$ is acyclic, where $(s', t') <_V (s, t)$ iff one of the following holds:

(i) $s' = s \wedge pos_s[t'] < pos_s[t]$;
(ii) $t' \sqsubset_{\text{rt}} t$; or
(iii) $t' \in T_s[t] \vee (pos_s[t'] < pos_s[t] \wedge d_s[t'] = \text{COMMIT} \wedge d[t'] = \text{ABORT} \wedge t' \notin P_s[t])$.

A protocol is a correct implementation of TCS-LL2 if each of its finite histories satisfies TCS-LL2.

Note that item (iii) above mirrors the definition of $\sqsubset_{\text{dec}}$ in Fig. 3. The relation $<_V$ determines an order on vote computations that must be respected by the execution of multi-shot 2PC we construct to produce the desired decisions.

Lemma 3 *The fault-tolerant commit protocol in Fig. 5 is a correct implementation of* TCS-LL2.

Proof Fix a finite execution of the fault-tolerant commit protocol producing a history h. Let T be the set of transactions t such that $\texttt{certify}(t)$ is an event in h. For some of transactions $t \in T$ and shards $s \in \mathsf{shards}(t)$ we define the certification order position $pos_s[t]$ and the vote $d_s[t]$ computed by the protocol as follows:

Consider $t \in T$ and $s \in \mathsf{shards}(t)$. Assume that a quorum of processes in s received $\texttt{ACCEPT}(b, k, t, d, \_)$ and responded to it with $\texttt{ACCEPT\_ACK}(s, b, k, t, d)$, and at the time $\mathsf{leader}(b)$ sent $\texttt{ACCEPT}(b, k, t, d, \_)$ it had $\mathsf{txn}\downarrow_k = txn$ and $\mathsf{vote}\downarrow_k = vote$. Then for every $j \leq k$ we let $pos_s[txn[j]] = j$ and $d_s[txn[j]] = vote[j]$.

According to Invariants 3 and 6, this defines $pos_s[t]$ and $d_s[t]$ uniquely and (11) in Fig. 3 holds. Furthermore, by the structure of the handler at line 31, for each t such that $\texttt{decide}(t, d[t])$ occurs in h, $d_s[t]$ is defined for all $s \in \mathsf{shards}(t)$ and (9) holds. By construction, (10) holds. We prove (12)–(14) using the following proposition.

Proposition 1 *The following always holds at any process in a shard s:*

$$
\begin{aligned}
&\forall k.\ (\mathsf{vote}[k] \textit{ is defined}) \implies \\
&\quad \exists T, P.\ \mathsf{vote}[k] = f_s(T, \mathsf{txn}[k]) \sqcap g_s(P, \mathsf{txn}[k]) \wedge \\
&\quad T = \{\mathsf{txn}[k'] \mid k' < k \wedge \mathsf{vote}[k'] = \text{COMMIT} \wedge \\
&\qquad\qquad d[\mathsf{txn}[k']] = \text{COMMIT}\} \setminus P \wedge \\
&\quad P \subseteq \{\mathsf{txn}[k'] \mid k' < k \wedge \mathsf{vote}[k'] = \text{COMMIT}\} \wedge \\
&\quad (\forall t \in T.\ (\texttt{DECISION}(t, \text{COMMIT}) \textit{ has been sent})) \wedge \\
&\quad (\forall k' < k.\ \mathsf{vote}[k'] = \text{COMMIT} \wedge \mathsf{txn}[k'] \notin T \cup P \\
&\qquad \implies (\texttt{DECISION}(t, \text{ABORT}) \textit{ has been sent})).
\end{aligned}
$$

Proof We prove this by induction on the length of the protocol execution. The validity of the above property can be nontrivially affected only by the transitions at lines 21, 28, 46 and 62.

First consider a transition at line 21 that computes $\mathsf{vote}[k]$ as follows:

$$
\begin{aligned}
&\mathsf{vote}[k] = f_s(T, \mathsf{txn}[k]) \sqcap g_s(P, \mathsf{txn}[k]); \\
&T = \{\mathsf{txn}[k'] \mid k' < k \wedge \mathsf{phase}[k'] = \text{DECIDED} \wedge \\
&\qquad\qquad \mathsf{dec}[k'] = \text{COMMIT}\}; \\
&P = \{\mathsf{txn}[k'] \mid k' < k \wedge \mathsf{phase}[k'] = \text{PREPARED} \wedge \\
&\qquad\qquad \mathsf{vote}[k'] = \text{COMMIT}\}.
\end{aligned}
$$

Then by Invariants 10 and 11,

$$
\begin{aligned}
&T = \{\mathsf{txn}[k'] \mid k' < k \wedge \mathsf{vote}[k'] = \text{COMMIT} \wedge \\
&\qquad\qquad d[\mathsf{txn}[k']] = \text{COMMIT}\} \setminus P \wedge \\
&(\forall t \in T.\ (\texttt{DECISION}(t, \text{COMMIT}) \text{ has been sent})) \wedge \\
&(\forall k' < k.\ \mathsf{vote}[k'] = \text{COMMIT} \wedge \mathsf{txn}[k'] \notin T \cup P \\
&\implies (\texttt{DECISION}(t, \text{ABORT}) \text{ has been sent}))
\end{aligned}
$$

which implies the required.

We next consider the transition at line 46 by a process p_i. Then p_i has received messages

$$\texttt{NEW\_LEADER\_ACK}(b'', cballot_j, txn_j, vote_j, dec_j, phase_j)$$

from a quorum of processes p_j. Let $b_0 = \max\{cballot_j \mid p_j \in Q'\}$ and $J = \{j \mid cballot_j = b_0\}$.

Consider k such that $\mathsf{vote}[k]$ is defined at p_i after the transition. By induction hypothesis, we have:

$$
\begin{aligned}
&(vote_j[k] \text{ is defined}) \implies \\
&\quad \exists T, P.\ vote_j[k] = f_s(T, txn_j[k]) \sqcap g_s(P, txn_j[k]) \wedge \\
&\quad T = \{txn_j[k'] \mid k' < k \wedge vote_j[k'] = \text{COMMIT} \wedge \\
&\qquad\qquad d[t'] = \text{COMMIT}\} \setminus P \wedge \\
&\quad P \subseteq \{txn_j[k'] \mid k' < k \wedge vote_j[k'] = \text{COMMIT}\} \wedge \\
&\quad (\forall t \in T.\ (\texttt{DECISION}(t, \text{COMMIT}) \text{ has been sent})) \wedge \\
&\quad (\forall k' < k.\ vote_j[k'] = \text{COMMIT} \wedge txn_j[k'] \notin T \cup P \\
&\qquad \implies (\texttt{DECISION}(t, \text{ABORT}) \text{ has been sent})).
\end{aligned}
$$

From Invariant 2 it follows that

$$
\begin{aligned}
&\forall j_1, j_2 \in J.\ (vote_{j_1}[k'] \text{ is defined}) \wedge \\
&\quad (vote_{j_2}[k'] \text{ is defined}) \implies vote_{j_1}[k'] = vote_{j_2}[k'].
\end{aligned}
$$

Hence,

$$\exists j_0 \in J.\, \forall k' \leq k.\, \mathsf{vote}[k'] = \mathit{vote}_{j_0}[k'].$$

Then the induction hypothesis implies

$$\begin{aligned}
&\exists T, P.\\
&\quad \mathsf{vote}[k] = f_s(T, \mathsf{txn}[k]) \sqcap g_s(P, \mathsf{txn}[k]) \wedge\\
&\quad T = \{\mathsf{txn}[k'] \mid k' < k \wedge \mathsf{vote}[k'] = \text{COMMIT} \wedge\\
&\qquad\qquad d[t'] = \text{COMMIT}\} \setminus P \wedge\\
&\quad P \subseteq \{\mathsf{txn}[k'] \mid k' < k \wedge \mathsf{vote}[k'] = \text{COMMIT}\} \wedge\\
&\quad (\forall t \in T.\, (\texttt{DECISION}(t, \text{COMMIT}) \text{ has been sent})) \wedge\\
&\quad (\forall k' < k.\, \mathsf{vote}[k'] = \text{COMMIT} \wedge \mathsf{txn}[k'] \notin T \cup P\\
&\quad \Longrightarrow (\texttt{DECISION}(t, \text{ABORT}) \text{ has been sent}))
\end{aligned}$$

as required.

Finally, the cases of the transitions at line 62 and 28 trivially follow from the induction hypothesis. □

We now prove (12)–(14). Take the earliest point in the execution where we can define $d_s[t]$ as per the definition given earlier. Let b be the ballot used in this definition. Then by Proposition 1 at this point at the leader of b for some $T_s[t]$, $P_s[t]$ we have

$$\begin{aligned}
&d_s[t] = f_s(T_s[t], t) \sqcap g_s(P_s[t], t) \wedge\\
&T_s[t] = \{\mathsf{txn}[k'] \mid k' < k \wedge \mathsf{vote}[k'] = \text{COMMIT} \wedge\\
&\qquad\qquad d[t'] = \text{COMMIT}\} \setminus P_s[t] \wedge\\
&P_s[t] \subseteq \{\mathsf{txn}[k'] \mid k' < k \wedge \mathsf{vote}[k'] = \text{COMMIT}\} \wedge\\
&(\forall t' \in T_s[t].\, (\texttt{DECISION}(t', \text{COMMIT}) \text{ has been sent})) \wedge\\
&(\forall k' < k.\, \mathsf{vote}[k'] = \text{COMMIT} \wedge \mathsf{txn}[k'] \notin T_s[t] \cup P_s[t]\\
&\Longrightarrow (\texttt{DECISION}(t', \text{ABORT}) \text{ has been sent})).
\end{aligned} \tag{19}$$

But then from the first three conjuncts we get

$$\begin{aligned}
&d_s[t] = f_s(T_s[t], t) \sqcap g_s(P_s[t], t) \wedge\\
&T_s[t] = \{t' \mid \mathit{pos}_s[t'] < \mathit{pos}_s[t] \wedge d[t'] = \text{COMMIT}\} \setminus P_s[t] \wedge\\
&P_s[t] \subseteq \{t' \mid \mathit{pos}_s[t'] < \mathit{pos}_s[t] \wedge d_s[t'] = \text{COMMIT}\},
\end{aligned}$$

which establishes (12)–(14) for the $T_s[t]$, $P_s[t]$ fixed above.

We next prove (15). Consider t, t', s such that

$$\begin{aligned}
&\texttt{decide}(t, _) \prec_h \texttt{certify}(t') \wedge\\
&s \in \mathsf{shards}(t) \cap \mathsf{shards}(t').
\end{aligned}$$

Let $\texttt{DECISION}(b, \mathit{pos}_s[t], \_)$ be the message send to the shard s when the $\texttt{decide}(t, \_)$ action was generated. Let b' be some ballot at which $\mathit{pos}_s[t']$ is defined according to the above definition. Assume first that $b' < b$. Then by Invariant 3 when the leader of b starts operating, it has $\mathsf{txn}[\mathit{pos}_s[t']] = t'$. But then $\texttt{certify}(t')$ must have occurred before $\texttt{decide}(t, \_)$. Hence, $b \leq b'$. By Invariant 3 when the leader of b' receives $\texttt{PREPARE}(t')$, it has $\mathsf{txn}[\mathit{pos}_s[t]] = t$. But then $\mathit{pos}_s[t] < \mathit{pos}_s[t']$, which proves (15).

Finally, we prove that $<_V$ is acyclic; it is easy to check that then (16) also holds. Let us denote by $<_{\text{i}}$, $<_{\text{ii}}$ and $<_{\text{iii}}$ the relations on V defined by the corresponding items in the definition of $<_V$, but the latter two relating only pairs with distinct shards. Consider a path π in $<_V$ starting from an edge of type (ii) or (iii). Due to (15), if $(s, t') <_V (s, t)$, then $\mathit{pos}_s[t'] < \mathit{pos}_s[t]$. Hence, we can assume that any edge on π of type (ii) or (iii) is between pairs with distinct shards and thus belongs to $<_{\text{ii}}$ or $<_{\text{iii}}$, respectively. Then without loss of generality we can assume that the path π is of the form defined by the regular expression $((<_{\text{ii}} \cup <_{\text{iii}}); <_{\text{i}}^?)^+$, where $^?$ denotes reflexive closure. We now prove that if

$$((s', t'), (s, t)) \in (<_{\text{ii}} \cup <_{\text{iii}}); <_{\text{i}}^?.$$

then: (*) a message $\texttt{DECISION}(t', d[t'])$ was sent in the execution, and this had happened before any message $\texttt{DECISION}(t, \_)$ was sent. This implies that π cannot form a cycle. Then, since $<_{\text{i}}$ is acyclic, so is $<_V$.

We first show the property (*) in the case when $((s', t'), (s, t)) \in <_{\text{ii}}; <_{\text{i}}^?$. Then for some transaction t'' we have $t' \sqsubset_{\text{rt}} t''$ and $\mathit{pos}_s[t''] \leq \mathit{pos}_s[t]$. Take the earliest point in the execution where we can define $\mathit{pos}_s[t'']$. Since $\mathit{pos}_s[t''] \leq \mathit{pos}_s[t]$, this point is no later than the earliest point at which we can define $\mathit{pos}_s[t]$. Then a message $\texttt{DECISION}(t, \_)$ could not have been sent by this point. Furthermore, $\texttt{certify}(t'')$ must have occurred before the point where we define $\mathit{pos}_s[t'']$. Since $t' \sqsubset_{\text{rt}} t''$, a message $\texttt{DECISION}(t', d[t'])$ must have also been sent before. But then $\texttt{DECISION}(t', d[t'])$ is sent before $\texttt{DECISION}(t, \_)$, as required.

We now show the property (*) in the case when $((s', t'), (s, t)) \in <_{\text{iii}}; <_{\text{i}}^?$. Then for some transaction t'' we have

$$\begin{aligned}
&(t' \in T_s[t''] \vee (\mathit{pos}_s[t'] < \mathit{pos}_s[t''] \wedge d_s[t'] = \text{COMMIT}\\
&\wedge d[t'] = \text{ABORT} \wedge t' \notin P_s[t'']))
\end{aligned}$$

and $\mathit{pos}_s[t''] \leq \mathit{pos}_s[t]$. Again, take the earliest point in the execution where we can define $\mathit{pos}_s[t'']$ and $d_s[t'']$, and hence, $T_s[t'']$ and $P_s[t'']$ (by (19)). Since $\mathit{pos}_s[t''] \leq \mathit{pos}_s[t]$, this point is no later than the earliest point at which we can define $\mathit{pos}_s[t]$. Then a message $\texttt{DECISION}(t, \_)$ could not have been sent by this point. If $t' \in T_s[t'']$, then by (19) a message $\texttt{DECISION}(t', \text{COMMIT})$ has been sent earlier. If

$$\begin{aligned}
&t' \notin T_s[t''] \wedge \mathit{pos}_s[t'] < \mathit{pos}_s[t''] \wedge d_s[t'] = \text{COMMIT}\\
&\wedge d[t'] = \text{ABORT} \wedge t' \notin P_s[t''],
\end{aligned}$$

then at the point when we define $pos_s[t'']$ we have $\mathsf{txn}[pos_s[t']] = t'$ and $\mathsf{vote}[pos_s[t']] = \text{COMMIT}$, so that by (19) a message DECISION(t', ABORT) has been sent earlier. Thus, at the point when we define $pos_s[t'']$, DECISION(t', _) message has been sent, and a message DECISION(t, _) has not been sent, which implies the required. □

Lemmas 2 and 3 already imply that the fault-tolerant protocol in Fig. 5 correctly implements a certification service. Theorem 2, showing that this protocol simulates multi-shot 2PC, follows from Lemma 3 and the following lemma: the relationship between histories established by the lemma implies refinement [14].

Lemma 4 *If a history h satisfies* TCS-LL2*, then there exists an execution of the multi-shot 2PC protocol in Fig. 1 producing a history h' that is a permutation of h preserving its real-time order.*

Proof For each shard s, let us define arrays $txn_s[\,]$ and $vote_s[\,]$ as follows: if $pos_s[t] = k$, then $txn_s[k] = t$ and $vote_s[k] = d_s[t]$. Given (10) and (11), this definition is well-formed. We construct the desired execution of the multi-shot 2PC protocol so that at its end, the arrays txn and vote at a shard s contain txn_s and $vote_s$, respectively. Let $<_P$ be any total order containing $<_V$, which exists because $<_V$ is acyclic. We construct the execution according to the following rules:

- For each vote $d_s[t]$ we execute the handler at line 8 for receiving a PREPARE(t) message at shard s in the order given by $<_P$. Since $<_{\text{i}} \subseteq <_V \subseteq <_P$, the position k assigned to the transaction t by this handler is $pos_s[t]$.
- Right before executing the handler at line 8 at shard s processing PREPARE(t), we execute the handler at line 21 to forget the decisions on all transactions that are currently DECIDED at the shard, but are in $P_s[t]$. Furthermore, for all transactions t' that are PREPARED at the shard and are such that

$$t' \in T_s[t] \vee (pos_s[t'] < pos_s[t''] \wedge d_s[t'] = \text{COMMIT} \wedge d[t'] = \text{ABORT} \wedge t' \notin P_s[t'']),$$

  we execute the handler at line 24 at all shards in $\mathsf{dest}(t')$ to resend PREPARE_ACK messages, the handler at line 14 at $\mathsf{coord}(t)$ to process them and send DECISION messages, and the handler at line 18 at shard s to process the DECISION message. Since $<_{\text{iii}} \subseteq <_V$, all such transactions t' have already been decided and, hence, the handler at line 24 can indeed be executed for them. By (12)–(14), the vote on t computed by the handler at line 8 is equal to $d_s[t]$.
- For each transaction t decided in h, we execute the handler at line 14 for t and, hence, generate event decide(t, _), immediately after the last shard processes a PREPARE(t) message for the first time. By (9), the decision computed by this handler is equal to $d[t]$.
- For each transaction t such that $d_s[t]$ is defined for some s, we execute the handler at line 6 for t immediately before a PREPARE(t) is processed for the first time. For each transaction t such that certify(t) occurs in h, but $d_s[t]$ is not defined for any s, we execute the handler at line 6 for t at the end of the execution. Since $<_{\text{ii}} \subseteq <_V$, this implies that the history h' of the execution we construct preserves the real-time order of h.

□

5 Related work

The notion of atomicity was coined by Gray as an informal requirement that a transaction "either happens or it does not" [18]. The Atomic Commit Problem (ACP) was later introduced to formalise atomicity as a variant of a distributed agreement problem. The existing work on 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 [17], which blocks in the event of the coordinator failure. The *non-blocking* variant of ACP known as Non-Blocking Atomic Commit (NBAC) [46] has been extensively studied in both the distributed computing and database communities [13,19–21,23,26,41,46]. The Three-Phase Commit (3PC) family of protocols [2,3,13,26,46] solve NBAC by augmenting 2PC with an extra message exchange round in the failure-free case. Paxos Commit [19] and Guerraoui et al. [21] 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 [22] 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 [22], 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 [8,15,39,40,42,45,47]. 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 [38] used non-fault-tolerant 2PC for transaction commit [39,40,42,47]. Our formalisation of the TCS problem should allow making them fault-tolerant using protocols of the kind we presented in Sect. 4.

Multiple researchers have observed that implementing transaction commit by layering 2PC on top of Paxos is suboptimal and proposed possible solutions [11,29,31,45,50]. In comparison to our work, they did not formulate a standalone 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. [29] and Zhang et al. [50] 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) and without rigorous proofs of correctness. Our fault-tolerant commit protocol borrows some of the key design decisions from the above systems, such as not replicating the 2PC coordinator. Our contribution is to provide a simple, generic and provably correct protocol that can serve as a reference solution for future distributed transaction commit implementations.

Ding et al. proposed a transaction processing system with optimistic concurrency control called Centiman [10]. The flow of transaction certification in Centiman is similar to our multi-shot 2PC (Sect. 3), with the transaction coordinator gathering votes from processes responsible for validating transactions in each shard. However, Centiman ensures fault-tolerance differently from our protocol in Sect. 4, by largely outsourcing it to an external storage service. We believe that our method of proving correctness of transaction certification can be applied to systems like Centiman, by establishing a simulation relation that maps steps of the protocol to the corresponding steps of multi-shot 2PC.

Sciascia et al. proposed Scalable Deferred Update Replication [45] for implementing serializable transactions in sharded systems. Like the vanilla protocol in Sect. 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 designed differently than in our protocol, relying on an external reconfiguration engine. Since the conference publication of this work, the approach we propose has also been applied to the class of systems similar to FaRM [4].

Mahmoud et al. proposed Replicated Commit [31], which reduces the latency of transaction commit by layering Paxos on top of 2PC, instead of the other way round. However, like the original 2PC, this protocol may block in case of data centre failures.

In [33], Maiyya et al. proposed the Consensus and Commitment (C&C) framework integrating the consensus and atomic commitment protocols into a generic template, which the authors then used to instantiate three transaction commit protocols for sharded and replicated data. Unlike our TCS framework, C&C is formulated in terms of the concrete protocols (Paxos and 2PC), rather than an abstract problem, and therefore, cannot capture a full variety of possible implementations. It also assumes that the transaction commits are requested one at a time, and is not generic in the isolation level. As a result, the protocols derived from C&C cannot easily accommodate common-case optimisations, resulting in suboptimal performance. For example, the most optimised protocol, G-PAC, requires 6 message delays to commit a single transaction in a failure-free run, whereas our fault-tolerant protocol requires only 4. In addition, both C&C and its derived algorithms were not formally specified and proven correct, which resulted in the discovery of several nontrivial safety issues in them [34,35].

MaaT, an optimistic concurrency control (OCC) scheme introduced in [32], aims to reduce the number of concurrency conflicts in a sharded cloud database by associating each transaction with a pair of timestamps whose values are dynamically adjusted based on the versions of the objects accessed by the transaction. Although at present, MaaT does not fit our proof methodology, which stipulates a standard OCC based on read and write sets, its high-level properties can still be captured through our TCS framework. Also the current implementation of MaaT has limited fault-tolerance as the shards are assumed to be non-replicated.

Schiper et al. proposed an alternative approach to implementing deferred update replication in sharded systems [43]. This distributes transactions to shards for certification using genuine atomic multicast [9], 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 mes-

sage, in addition to a varying convoy effect among different transactions [7]. The resulting overall latency of certification is 5 message delays plus the convoy effect.

An alternative approach to transaction processing avoids aborts by pre-determining the order of transaction execution [36,48], thus implementing a form of atomic multicast [9]. The techniques described in this paper can also be applied to minimise the latency of this primitive [16].

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 [25] and Viewstamped Replication [37]. Kokocinski et al. [28] 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.

For conciseness, in this paper we focussed on transaction processing systems using optimistic concurrency control. But we believe that our framework can also be generalised to systems that employ pessimistic concurrency control, following the analogy between shard-local certification functions and object-level locks we pointed out in Sect. 3. In particular, our fault-tolerant protocol could be adjusted so that the locks protecting objects are solely managed by shard leaders, and acquiring locks during transaction execution ensures that shard-local certification functions evaluate to COMMIT when the transaction is submitted for certification. We plan to report in detail on extensions of our framework to pessimistic concurrency control in future papers.

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.

Acknowledgements Alexey Gotsman was supported by a Starting Grant RACCOON from the European Research Council.

References

1. Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., O'Neil, P.: A critique of ANSI SQL isolation levels. In: Conference on Management of Data (SIGMOD) (1995)
2. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison-Wesley Longman Publishing Co., Inc., Boston (1986)
3. Borr, A.J.: Transaction monitoring in ENCOMPASS: reliable distributed transaction processing. In: International Conference on Very Large Data Bases (VLDB) (1981)
4. Bravo, M., Gotsman, A.: Reconfigurable atomic transaction commit. In: Symposium on Principles of Distributed Computing (PODC) (2019)
5. Chandra, T.D., Hadzilacos, V., Toueg, S.: The weakest failure detector for solving consensus. J. ACM **43**(4), 685–722 (1996)
6. Charron-Bost, B., Schiper, A.: Uniform consensus is harder than consensus. J. Algorithms **51**(1), 15–37 (2004)
7. Coelho, P.R., Schiper, N., Pedone, F.: Fast atomic multicast. In: Conference on Dependable Systems and Networks (DSN) (2017)
8. Corbett, J.C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J.J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh, W.C., Kanthak, S., Kogan, E., Li, H., Lloyd, A., Melnik, S., Mwaura, D., Nagle, D., Quinlan, S., Rao, R., Rolig, L., Saito, Y., Szymaniak, M., Taylor, C., Wang, R., Woodford, D.: Spanner: Google's globally-distributed database. In: Symposium on Operating Systems Design and Implementation (OSDI) (2012)
9. Défago, X., Schiper, A., Urbán, P.: Total order broadcast and multicast algorithms: taxonomy and survey. ACM Comput. Surv. **36**(4), 372–421 (2004)
10. Ding, B., Kot, L., Demers, A., Gehrke, J.: Centiman: elastic, high performance optimistic concurrency control by watermarking. In: Symposium on Cloud Computing (SoCC) (2015)
11. Dragojević, A., Narayanan, D., Nightingale, E.B., Renzelmann, M., Shamis, A., Badam, A., Castro, M.: No compromises: distributed transactions with consistency, availability, and performance. In: Symposium on Operating Systems Principles (SOSP) (2015)
12. Dwork, C., Lynch, N., Stockmeyer, L.: Consensus in the presence of partial synchrony. J. ACM **35**(2), 288–323 (1988)
13. Dwork, C., Skeen, D.: The inherent cost of nonblocking commitment. In: Symposium on Principles of Distributed Computing (PODC) (1983)
14. Filipovic, I., O'Hearn, P.W., Rinetzky, N., Yang, H.: Abstraction for concurrent objects. Theor. Comput. Sci. **411**(51–52), 4379–4398 (2010)
15. Glendenning, L., Beschastnikh, I., Krishnamurthy, A., Anderson, T.: Scalable consistency in Scatter. In: Symposium on Operating Systems Principles (SOSP) (2011)
16. Gotsman, A., Lefort, A., Chockler, G.V.: White-box atomic multicast. In: Conference on Dependable Systems and Networks (DSN) (2019)
17. Gray, J.: Notes on data base operating systems. In: International Conference on Very Large Databases (1978)
18. Gray, J.: The transaction concept: virtues and limitations (invited paper). In: International Conference on Very Large Data Bases (VLDB) (1981)
19. Gray, J., Lamport, L.: Consensus on transaction commit. ACM Trans. Database Syst. **31**(1), 133–160 (2006)
20. Guerraoui, R.: Revisiting the relationship between non-blocking atomic commitment and consensus. In: Workshop on Distributed Algorithms (WDAG) (1995)
21. Guerraoui, R., Larrea, M., Schiper, A.: Reducing the cost for non-blocking in atomic commitment. In: International Conference on Distributed Computing Systems (ICDCS) (1996)
22. Guerraoui, R., Wang, J.: How fast can a distributed transaction commit? In: Symposium on Principles of Database Systems (PODS) (2017)
23. Hadzilacos, V.: On the relationship between the atomic commitment and consensus problems. In: Asilomar Workshop on Fault-Tolerant Distributed Computing (1990)

24. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3), 463–492 (1990)
25. Junqueira, F.P., Reed, B.C., Serafini, M.: Zab: high-performance broadcast for primary-backup systems. In: Conference on Dependable Systems and Networks (DSN) (2011)
26. Keidar, I., Dolev, D.: Increasing the resilience of atomic commit at no additional cost. In: Symposium on Principles of Database Systems (PODS) (1995)
27. Keidar, I., Rajsbaum, S.: A simple proof of the uniform consensus synchronous lower bound. Inf. Process. Lett. **85**(1), 47–52 (2003)
28. Kokocinski, M., Kobus, T., Wojciechowski, P.T.: Make the leader work: executive deferred update replication. In: Symposium on Reliable Distributed Systems (SRDS) (2014)
29. Kraska, T., Pang, G., Franklin, M.J., Madden, S., Fekete, A.: MDCC: multi-data center consistency. In: European Conference on Computer Systems (EuroSys) (2013)
30. Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. **16**(2), 133–169 (1998)
31. Mahmoud, H., Nawab, F., Pucher, A., Agrawal, D., El Abbadi, A.: Low-latency multi-datacenter databases using replicated commit. Proc. VLDB Endow. **6**(9), 661–672 (2013)
32. Mahmoud, H.A., Arora, V., Nawab, F., Agrawal, D., El Abbadi, A.: Maat: effective and scalable coordination of distributed transactions in the cloud. Proc. VLDB Endow. **7**(5), 329–340 (2014)
33. Maiyya, S., Nawab, F., Agrawal, D., El Abbadi, A.: Unifying consensus and atomic commitment for effective cloud data management. Proc. VLDB Endow. **12**(5), 611–623 (2019)
34. Maiyya, S., Nawab, F., Agrawal, D., El Abbadi, A.: Private communication. 2020–2021
35. Maiyya, S., Nawab, F., Agrawal, D., El Abbadi, A.: Errata for "Unifying consensus and atomic commitment for effective cloud data management". Submitted to VLDB'21 (2021)
36. Mu, S., Nelson, L., Lloyd, W., Li, J.: Consolidating concurrency control and consensus for commits under conflicts. In: Symposium on Operating Systems Design and Implementation (OSDI) (2016)
37. Oki, B.M., Liskov, B.H.: Viewstamped replication: a new primary copy method to support highly-available distributed systems. In: Symposium on Principles of Distributed Computing (PODC) (1988)
38. Pedone, F., Guerraoui, R., Schiper, A.: The database state machine approach. Distrib. Parallel Databases **14**(1), 71–98 (2003)
39. Peluso, S., Romano, P., Quaglia, F.: Score: a scalable one-copy serializable partial replication protocol. In: International Middleware Conference (Middleware) (2012)
40. Peluso, S., Ruivo, P., Romano, P., Quaglia, F., Rodrigues, L.E.T.: GMU: genuine multiversion update-serializable partial data replication. IEEE Trans. Parallel Distrib. Syst. **27**(10), 71–98 (2016)
41. Ramarao, K.V.S.: Complexity of distributed commit protocols. Acta Inform. **26**(6), 577–595 (1989)
42. Saeida Ardekani, M., Sutra, P., Shapiro, M.: G-DUR: a middleware for assembling, analyzing, and improving transactional protocols. In: International Middleware Conference (Middleware) (2014)
43. Schiper, N., Sutra, P., Pedone, F.: P-store: genuine partial replication in wide area networks. In: Symposium on Reliable Distributed Systems (SRDS) (2010)
44. Schneider, F.B.: Implementing fault-tolerant services using the state machine approach: a tutorial. ACM Comput. Surv. **22**(4), 299–319 (1990)
45. Sciascia, D., Pedone, F., Junqueira, F.: Scalable deferred update replication. In: Conference on Dependable Systems and Networks (DSN) (2012)
46. Skeen, D.: Nonblocking commit protocols. In: Conference on Management of Data (SIGMOD) (1981)
47. Sovran, Y., Power, R., Aguilera, M.K., Li, J.: Transactional storage for geo-replicated systems. In: Symposium on Operating Systems Principles (SOSP) (2011)
48. Thomson, A., Diamond, T., Weng, S., Ren, K., Shao, P., Abadi, D.J.: Calvin: fast distributed transactions for partitioned database systems. In: Conference on Management of Data (SIGMOD) (2012)
49. Weikum, G., Vossen, G.: Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery. Morgan Kaufmann Publishers Inc., Burlington (2001)
50. Zhang, I., Sharma, N.K., Szekeres, A., Krishnamurthy, A., Ports, D.R.K.: Building consistent transactions with inconsistent replication. In: Symposium on Operating Systems Principles (SOSP) (2015)
51. Zhang, I., Sharma, N.K., Szekeres, A., Krishnamurthy, A., Ports, D.R.K.: When is operation ordering required in replicated transactional storage? IEEE Data Eng. Bull. **39**(1), 27–38 (2016)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.