Reconfigurable Atomic Transaction Commit

Manuel Bravo
IMDEA Software Institute

Alexey Gotsman
IMDEA Software Institute

ABSTRACT
Modern data stores achieve scalability by partitioning data into shards and fault-tolerance by replicating each shard across several servers. A key component of such systems is a Transaction Certification Service (TCS), which atomically commits a transaction spanning multiple shards. Existing TCS protocols require \(2f + 1\) crash-stop replicas per shard to tolerate \(f\) failures. In this paper we present atomic commit protocols that require only \(f + 1\) replicas and reconfigure the system upon failures using an external reconfiguration service. We furthermore rigorously prove that these protocols correctly implement a recently proposed TCS specification. We present protocols in two different models—the standard asynchronous message-passing model and a model with Remote Direct Memory Access (RDMA), which allows a machine to access the memory of another machine over the network without involving the latter’s CPU. Our protocols are inspired by a recent FARM system for RDMA-based transaction processing. Our work codifies the core ideas of FARM as distributed TCS protocols, rigorously proves them correct and highlights the trade-offs required by the use of RDMA.

CSCS CONCEPTS
- Theory of computation → Distributed algorithms; Distributed computing models;

KEYWORDS
Atomic commit, vertical Paxos, RDMA.

ACM Reference Format:

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 [7, 11, 23]. Such data stores often use optimistic concurrency control [27], where a transaction is first executed speculatively, and the results (e.g., read and write sets) are then certified to determine whether the transaction can commit or must abort because of a conflict with concurrent transactions. The certification is implemented by a Transaction Certification Service (TCS), which accepts a stream of transactions and outputs decisions based on a given certification function, defining the concurrency-control check for the desired isolation level. TCS is the most challenging part of transaction processing in systems with the above architecture, since it requires solving a distributed agreement problem among the replicated shards participating in the transaction. This agreement problem has been recently formalized as the multi-shot commit problem [6], generalizing the classical atomic commit problem [10] to more faithfully reflect the requirements of modern transaction processing systems (we review the new problem statement in §2).

Most existing solutions to the TCS problem require replicating each shard among \(2f + 1\) replicas to tolerate \(f\) crash-stop failures within each shard [7, 11, 16, 28], which allows using a replication protocol such as Paxos [17]. This is expensive: if transaction data are written to all replicas of the shard, only \(f + 1\) replicas are needed for the data to survive failures. Since, in this case even a single replica failure will block transaction processing, to recover we need to reconfigure the system, i.e., change its membership to replace failed replicas with fresh ones. Unfortunately, processes concurrently deciding to reconfigure the system need to be able to agree on the next configuration; this reduces to solving consensus, which again requires \(2f + 1\) replicas [18]. The way out of this conundrum is to use a separate configuration service with \(2f + 1\) replicas to perform consensus on the configuration. In this way, we use \(2f + 1\) replicas only to store the small amount of information about the configuration and \(f + 1\) replicas to store the actual data. This vertical approach [19], which layers replication on top of a configuration service, has been used by a number of practical systems [1, 5, 9]. It is particularly suitable for deployment in local-area networks, where the configuration service can be reached quickly.

In this paper we propose the first rigorously proven protocols for implementing a TCS in a vertical system, with \(f + 1\) replicas per shard and an external configuration service. We present protocols in two different models—the standard asynchronous message-passing model (§3) and a model with Remote Direct Memory Access (RDMA), which allows a machine to access the memory of another machine over the network without involving the latter’s CPU (§5). Our protocols are parametric in the isolation level provided, and we prove that they correctly implement the TCS specification from the multi-shot commit problem [6] (§4).

Our work complements and takes its inspiration from a recent FARM system [9, 10]—a transaction processing system that achieves impressive scalability and availability by exploiting RDMA and the vertical approach. FARM currently forms the core of a graph database used to serve some of search queries in Microsoft Bing. It is a complex system that includes a number of optimizations, both specific to RDMA and not. FARM’s design was presented without a rigorous proof of correctness, and it did not highlight which features
are motivated by the use of RDMA and which are inherent to the vertical approach. Our work provides a theoretical complement to FARM: we codify its core ideas as distributed transaction commit protocols and rigorously prove them correct with respect to the TCS specification. By basing our protocols on a principled footing, we are also able to provide better fault-tolerance guarantees than FARM. Finally, by also implementing two related protocols using message passing and RDMA, we determine the trade-offs required by the use of RDMA.

In more detail, a straightforward way to implement TCS is using the classical two-phase commit (2PC) protocol [12]. Since 2PC is not fault-tolerant, we can make each shard simulate a reliable process in 2PC using a replication protocol such as Paxos [7, 11]. This vanilla approach requires every 2PC action to be replicated using Paxos, which results in a high latency (7 message delays to learn a decision on a transaction [22]) and a high load on Paxos leaders. To improve on this, our protocol combines 2PC and Vertical Paxos [19] into one coherent protocol, thereby minimizing the latency and load on Paxos leaders. Upon a failure inside a shard, we use the reconfiguration service to replace the failed replicas, as in Vertical Paxos. This reconfiguration interacts nontrivially with the 2PC part of the protocol: e.g., reconfiguration may lead to losing undecided transactions that affected 2PC computations of decisions on other transactions—a behavior that we nevertheless show to be correct. Finally, we show that the price of exploiting RDMA to efficiently write transaction data to replicas is that reconfiguration has to be performed globally, instead of per-shard: when reconfiguring a shard, we have to ensure that the whole system is aware of the configuration before activating it.

2 TRANSACTION CERTIFICATION SERVICE

Service interface and certification functions. A Transaction Certification Service (TCS) is meant to be used in the context of transactional processing systems with optimistic concurrency control [27], where transactions are first executed speculatively, and the results are submitted for certification to the TCS. We start by reviewing its specification proposed in [6]. Clients invoke the TCS using requests of the form certify(t, l), where t ∈ T is a unique transaction identifier and l ∈ L is the transaction payload, which carries the results of the optimistic execution of the transaction (e.g., read and write sets). Responses of the service are of the form decide(t, d), where d ∈ D = {ABORT, COMMIT}. A TCS is specified using a certification function f : 2^L × L → D, which encapsulates the concurrency-control policy for the desired isolation level. The result f(L, l) is the decision for the transaction with payload l given the set of payloads L of the previously committed transactions. We require f to be distributive in the following sense:

\[ \forall L_1, L_2, l. f(L_1 ∪ L_2, l) = f(L_1, l) \cap f(L_2, l), \]

where \( \cap \) is such that COMMIT ∩ COMMIT = COMMIT and d ∩ ABORT = ABORT for any d. This requirement is justified by the fact that common definitions of f(L,l) check l for conflicts against each transaction in L separately.

As an example, consider a transactional system managing objects from Obj with values from Val, where transactions can execute reads and writes on the objects. The objects are associated with a totally ordered set Ver of versions. Then the payload of a transaction t is a triple \((R, W, V_t)\). Here the read set \( R \subseteq \text{Obj} \times \text{Ver} \) is the set of objects with their versions that t read, which contains one version per object. The write set \( W \subseteq \text{Obj} \times \text{Val} \) is 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: \( \forall (x, \_v) \in W, (x, \_v) \in R \). Finally, the commit version \( V_t \in \text{Ver} \) is the version to be assigned to the writes of t. We require this version to be higher than any of the versions read: \( \forall (\_, v) \in R, V_t > v \). Given this domain of transactions, the following certification function encapsulates the classical concurrency-control policy for serializability [27]: \( f(L, (R, W, V_t)) = \text{COMMIT} \) iff none of the versions in R have been overwritten by a transaction in L, i.e.,

\[ \forall x, v. (x, v) \in R \implies (\forall L', V_{t}' \subseteq W', (x, \_v) \in W' \implies V'_{t} \leq v) \quad (2) \]

TCS specification. We represent TCS executions using histories—sequences of certify and decide actions such that every transaction appears at most once in certify, and each decide is a response to exactly one preceding certify. For a history \( h \) we let act(h) be the set of actions in \( h \). For actions \( a, a' \in \text{act}(h) \), we write \( a \prec_h a' \) when \( a \) occurs before \( a' \) in \( h \). A history \( h \) is complete if every certify action in it has a matching decide action. A complete history is sequential if it consists of pairs of certify and matching decide actions. A transaction \( t \) commits in a history \( h \) if \( h \) contains decide(t, COMMIT). We denote by 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 \( h \) [14] is a sequential history such that \( h \) and \( l \) contain the same actions and

\[ \forall t, t'. \text{decide}(t, \_l) \prec_h \text{certify}(t', \_l) \implies \text{decide}(t, \_l) \prec_l \text{certify}(t', \_l). \]

A complete sequential history \( h \) is legal with respect to a certification function \( f \), if its decisions are computed according to \( f \):

\[ \forall t, l, d. \text{certify}(t, l), \text{decide}(t, d) \in \text{act}(h) \implies d = f(l' \mid \text{certify}(t', l'), l' \in \text{act}(h) \wedge \text{decide}(t', \text{COMMIT}) \prec_h \text{decide}(t, d), l). \]

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

A TCS implementation satisfying the above specification can be readily used in a transaction processing system. For example, consider the domain of transactions defined earlier. A typical system based on optimistic concurrency control will ensure that transactions submitted for certification only read versions written by previously committed transactions. A history produced by such a system that is correct with respect to certification function (2) is also serializable [6]. Hence, a TCS correct with respect to this certification function can indeed be used to implement serializability.

Shard-local certification functions. We are interested in TCS implementations in systems where the data are partitioned into shards from a set \( S \). In such systems TCS is usually implemented using a variant of the classical two-phase commit protocol (2PC) [12].
In this protocol each shard \( s \) receiving a transaction for certification first prepares it, i.e., performs a local concurrency-control check and accordingly votes to commit or abort the transaction. The votes on the transaction by different shards are aggregated, and the final decision is then distributed to all shards: the transaction can commit only if all votes are commit. When a shard \( s \) votes on a transaction, it does not have information about all transactions in the system, but only those that concern it. Hence, the votes are computed using not the global certification function \( f \), but shard-local certification functions [6], which check for conflicts only on objects managed by the shard and corresponding to names as parameters only the parts of the transaction payloads relevant to the shard: for a payload \( l \) we denote this by \( l | s \). For example, let \( \text{Obj}_s \) be the set of objects managed by a shard \( s \). For a payload \( l = \langle R, W, V_c \rangle \) of the form given above, we let \( l | s = \langle R^s, W^s, V_c \rangle \), where \( R^s = \{(x, _) \in R \mid x \in \text{Obj}_s\} \) and \( W^s = \{(x, _) \in W \mid x \in \text{Obj}_s\} \). There are two shard-local functions, \( f_l : 2^L \times L \rightarrow D \) and \( g_l : 2^L \times L \rightarrow D \). As its first argument \( f_l \) takes the set of shard-relevant payloads of transactions that previously committed at the shard, and \( g_l \) the set of such payloads for transactions that have been prepared to commit. As their second argument, the functions take the part of the payload of the transaction being certified relevant to the shard. We require that these functions are distributive, similarly to (1).

For example, the shard-local certification functions for serializability are defined as follows: 
\[
\forall x \in \text{Obj}_s, \forall v. (x, v) \in R \implies (\forall \langle _, W', V_c' \rangle \in L, (x, _) \in W' \implies V_c' \leq v),
\]
and 
\[
\forall x \in \text{Obj}_s, \forall v. ((x, _) \in R \implies (\forall \langle _, W', V_c' \rangle \in L, (x, _) \notin W' \wedge (x, _) \in W \implies (\forall \langle R', _, V_c' \rangle \in L, (x, _) \notin R'))).
\]

The function \( f_l \) certifies a transaction \( t \) against previously committed transactions similarly to the certification function (2) for serializability, but taking into account only the objects managed by the shard \( s \). The function \( g_l \) certifies \( t \) against transactions prepared to commit, and its check is stricter than that of \( f_l \). In our example, the function \( g_l \) aborts a transaction \( t \) if: (i) it read an object written by a transaction \( t' \) prepared to commit; or (ii) it writes to an object read by a transaction \( t' \) prepared to commit. This reflects the behaviour of typical implementations, which upon preparing a transaction acquire read locks on its read set and write locks on its write set, and abort the transaction if the locks cannot be acquired.

For a sharded TCS implementation to be correct, shard-local functions have to match the global certification function, i.e., perform similar conflict checks. We formalize the required conditions as follows. Assume a function shard : \( T \rightarrow 2^S \) that determines the shards that need to certify a transaction with a given identifier, which are usually the shards storing the data the transaction accesses. We also assume a distinguished empty payload \( e \in L \) such that \( \forall s, L, f_s(L, e) = \text{commit} \). For example, for a payload \( l = \langle R, W, V_c \rangle \) of the form given above, \( l = e \) is such that \( R = \emptyset \) and \( W = \emptyset \). We require that for a transaction \( t \in T \) with payload \( l \in L \), for each shard \( s \not\in \text{shards}(t) \), we have \( l | s = e \). We further lift the \( | \) operator to sets of payloads: for any \( L \subseteq L \) we let \( (L | s) = \{(l | s) \mid l \in L\} \). Then we require that global and local certification functions match as follows:

\[
\forall l \in L, \forall l \subseteq L. f_s(l, l) = \text{commit} \iff \forall s \in S. f_s((l | s), (l | s)) = \text{commit}. \tag{3}
\]

Finally, for each shard \( s \in S \), the two functions \( f_s \) and \( g_s \) are required to be related to each other as follows [6]:

\[
\forall l \in L, l \subseteq L. g_s(l, l) = \text{commit} \implies f_s(l, l) = \text{commit}; \tag{4}
\]

\[
\forall l, l' \in L. g_s(l, l') = \text{commit} \implies f_s(l', l) = \text{commit}. \tag{5}
\]

Property (4) requires the conflict check performed by \( g_s \) to be no weaker than the one performed by \( f_s \). Property (5) requires a form of commutativity: if a transaction with payload \( l' \) is allowed to commit after a still-pending transaction with payload \( l \), then the latter would be allowed to commit after the former.

### 3 ATOMIC COMMIT PROTOCOL

#### System model

We consider an asynchronous message-passing system consisting of a set of processes \( P \) which may fail by crashing, i.e., permanently stopping execution. We assume that processes are connected by reliable FIFO channels: messages are delivered in FIFO order, and messages between non-faulty processes are guaranteed to be eventually delivered. A function client : \( T \rightarrow P \) determines the client process that issued a given transaction.

Each shard \( s \in S \) is managed by a group of replica processes, whose membership can change over time. For simplicity, we assume that the groups of replica processes managing different shards are disjoint. Each shard moves through a sequence of configurations, determining its membership. Reconfiguration is the process of changing the configuration of a shard. In our protocols reconfiguration is initiated by a replica when it suspects another replica of failing: for simplicity we do not expose it in the TCS interface. Every member of a shard in a given configuration is either the leader of the shard or a follower. A configuration of a shard \( s \) is then a tuple \((e, M, p_l)\) where \( e \) is the epoch identifying the configuration, \( M \in 2^P \) is the set of processes that manage \( s \) at \( e \), and \( p_l \in M \) is the leader of \( s \) at \( e \).

Configurations are stored in an external configuration service (CS), which for simplicity we assume to be a reliable process. In practice, this service may be implemented using Paxos-like replication over \( 2f + 1 \) processes out of which at most \( f \) can fail (as done in systems such as Zookeeper [15]). The configuration service stores the configurations of all shards and provides three operations. An operation compare_and_swap(s, e, (e', M, p_l)) succeeds if the epoch of the last stored configuration of \( s \) is \( e \); in this case it stores the provided configuration with a higher epoch \( e' > e \). Operations get_last(s) and get(s, e) respectively return the last configuration of \( s \) and the configuration of \( s \) associated with a given epoch \( e \).

#### Protocol preliminaries

We give the pseudocode of our protocol in Figure 1. illustrate its message flow in Figure 2 and summarize the key invariants used in its proof of correctness in Figure 3. The protocol weaves together the two-phase commit protocol across shards [12] and a Vertical Paxos-based reconfiguration protocol within each shard [19]. At any given time, a process participates in a single configuration of the shard it belongs to. The process stores the information about this configuration as well as those of other...
shards in several arrays: configuration epochs are stored in an array $e \in \mathcal{E} \rightarrow \mathcal{N}$, the current members in members $m \in \mathcal{M} \rightarrow \mathcal{P}$, and the current leader in leader $l \in \mathcal{L} \rightarrow \mathcal{P}$. The entries for the shard the process belongs to configure the process is in; the other entries maintain information about the configurations of the other shards. A status variable at a process records whether it is a leader, a follower or is in a special reconfiguring state used during reconfiguration. Each process keeps track of the status of transactions in an array phase, whose entries initially store store.

The transaction status changes to PREPARED when the shard determines its vote and to DECIDED when a final decision on the transaction is reached.

**Failure-free case.** A client submits a transaction for certification by calling the certify function at any replica process, which will serve as the coordinator of the transaction (line 1). The function takes as arguments the transaction’s identifier and its payload. The transaction coordinator first sends a PREPARE message to the leader of the relevant shards, which includes the payload part for each shard (line 3). The leader of a shard arranges all transactions it receives into a total certification order, which the leader stores in an array $t \in \mathcal{T} \rightarrow \mathcal{N}$; a next $e \in \mathcal{Z}$ variable points to the last filled slot in the array. When the leader receives a PREPARE message for a transaction for the first time (line 8), it appends the transaction to the certification order, stores the transaction’s payload in an array $p \in \mathcal{P} \rightarrow \mathcal{L}$, and sets the transaction’s phase to PREPARED. It then computes a vote on the transaction and stores it in an array vote $v \in \mathcal{V} \rightarrow \{\text{commit, abort}\}$ (line 12). The vote is computed using the shard-local certification functions $f_{s0}$ and $g_{s0}$ to check for conflicts against transactions that have been previously committed or prepared to commit; the results are combined using the $\cap$ operator, so that the transaction can commit only if both functions say so. We defer the description of the cases when the leader has previously received the transaction in the PREPARE message (line 6) and when the payload in the message is an undefined value $\bot$ (line 14).

Our protocol next replicates the leader’s decision and the transaction payload at the followers. Instead of having the leader to do this directly, the protocol delegates this task to the coordinator of the transaction. This design is used by practical systems, such as Corfu [1] and FARM [9], since it minimizes the load on the leaders, which are the main potential performance bottleneck. Instead, the network-intensive task of persisting transactions at multiple followers is spread among a number of different transaction coordinators. As we explain in the following, this optimization interacts in a nontrivial way with transaction certification. In more detail, after preparing a transaction the leader sends a PREPARE_ACK message to the coordinator of the transaction, which carries the leader’s epoch, the transaction identifier, its position in the certification order, the payload, and the vote (line 20). Upon receiving the PREPARE_ACK message (line 18), the coordinator forwards the data from the PREPARE_ACK message to the followers in an ACCEPT message.

A process handles an ACCEPT message only if it participates in the corresponding epoch (line 22). The process stores the transaction identifier, its payload and vote, and advances the transaction’s phase to PREPARED. It then sends an ACCEPT_ACK message to the coordinator of the transaction, confirming that the process has accepted the transaction and the vote. The certification order at a follower is always a prefix with zero or more holes of the certification order at the leader of the epoch the follower is in, as formalized by Invariant 1 (Figure 3). The holes in the prefix arise from the lack of FIFO ordering in the communication between the leader of a given epoch and its followers, as the ACCEPT message for a given transaction is sent to the followers by the coordinator of the transaction and not directly by the leader.

The coordinator of a transaction $t$ acts once it receives ACCEPT_ACK messages for $t$ from every follower of its shards $s \in \mathcal{S}(t)$ (line 26); it determines this using the configuration information it stores for every shard. The coordinator computes the final decision on $t$ using the $\cap$ operator on the votes of each involved shard: the transaction can commit if all votes are commit. The coordinator then sends the final decision in DECISION messages to the client and to each of the relevant shards. When a process receives a decision for a transaction (line 30), it stores the decision and advances the transaction’s phase to DECIDED. In a realistic implementation, at this point the process would also upcall into the transaction processing system running at its server, to inform it about the decision and allow it to apply the transaction’s writes to the database if the decision is to commit.

In the absence of failures, our protocol allows the client to learn a decision on a transaction in 5 message delays, instead of 7 required by vanilla protocols that use Paxos as a black-box [7, 11]. We can further reduce this to 4 by co-locating the client with the transaction coordinator. The protocol also minimizes the load on Paxos leaders, which are the main potential bottleneck: each involved leader only has to receive one PREPARE and one DECISION message, and send one PREPARE_ACK message.

**Reconfiguration.** When a failure is suspected in a shard $s$, any process can initiate a reconfiguration of the shard to replace failed replicas. Reconfiguration is done only in the affected shard, without disrupting others. It aims to preserve Invariant 2, which is key in proving the correctness of the protocol. This assumes that all followers in $s$ at an epoch $e$ have received ACCEPT($e, k, t, l, d$) and responded to it with ACCEPT_ACK; in this case we say that the transaction $t$ has been accepted at shard $s$. The invariant guarantees that the accepted transaction $t$ will persist in epochs higher than $e$; this is used to prove that the protocol computes a unique decision on each transaction. The invariant also guarantees that the entries preceding $t$ in the certification order in epochs higher than $e$ may only contain the votes that the leader of $s$ at epoch $e$ took into account when computing the vote $d$ on $t$ (some of these votes may be missing due to the lack of FIFO order in the communication between the leader and its followers). This property is necessary to guarantee that the protocol computes decisions according to a single global certification order, as required by the TCS specification.

To ensure Invariant 2, a process performing reconfiguration first probes previous configurations to determine which processes are still alive and to find a process whose state contains all transactions previously accepted at the shard, which will serve the new leader. The new leader then transfers its state to the members of the new configuration, thereby initializing them. A variable initialized $\in \{\text{true, false}\}$ at a process records whether it has
function certify(t, l)
    for all s ∈ shards(t) do
        send PREPARE(t, l, s) to leader[s];

when received PREPARE(t, l) from p_j
    pre: status = LEADER;
    if \exists k, t = txn[k] then
        send PREPARE_ACK(epoch[s_0], s_0, k, txn[k], payload[k], vote[k]) to p_j
    else
        next ← next + 1;
        (txn, phase)[next] ← (t, PREPARED);
        if l ≠ phase then
            vote[next] ← f_{s_0}(L_1, l) \cap g_{s_0}(L_2, l);
        payload[next] ← l;
        else
            vote[next] ← ABORT;
            payload[next] ← e;
        end
        send PREPARE_ACK(epoch[s_0], s_0, next, t, payload[next], vote[next]) to p_j;

when received PREPARE_ACK(e, s, k, t, l, d)
    pre: epoch[s] = e;
    send ACCEPT(e, s, k, t, l, d) to members[s] \ leader[s];

when received ACCEPT(e, s, k, t, l, d) from p_j
    pre: status = FOLLOWER ∧ epoch[s_0] = e;
    if phase[k] = \_START then
        (txn, payload, vote, phase)[next] ← (t, l, d, PREPARED);
    end
    send ACCEPT_ACK(s_0, e, k, t, l, d) to p_j;

when for every s ∈ shards(t) received an ACCEPT_ACK(s, epoch[s], k_s, t, d_s)
    for every p_j ∈ members[s] \ leader[s]
        send DECISION(t, \bigcap_{s \in shards(t)} d_s) to client(t);
    end

when received DECISION(e, k, d)
    pre: status ∈ \{LEADER, FOLLOWER\} ∧ epoch[s_0] ≥ e;
    (dec, phase)[k] ← (d, DECIDED);

function reconfigure(s)
    pre: probing = FALSE;
    probing ← TRUE;
    (probed_epoch, probed_members, _) ← get_last(s) at CS;
    recon_epoch ← probed_epoch + 1;
    recon_shard ← s;
    send PROBE(recon_epoch) to probed_members;

when received PROBE(e) from p_j
    pre: e ≥ new_epoch;
    status = RECONFIGURING;
    new_epoch ← e;
    send PROBE_ACK(initialized, e, s_0) to p_j;

when received PROBE_ACK(true, recon_epoch, recon_shard) from p_j
    pre: probing = TRUE;
    probing ← FALSE;
    M ← compute_membership();
    var r ← compare_and_swap(recon_shard, recon_epoch - 1, (recon_epoch, M, p_j)) at CS;
    if r then send NEW_CONFIG(recon_epoch, M) to p_j;

non-deterministically when received
    PROBE_ACK(false, recon_epoch, recon_shard) from p_j ∈ probed_members and no
    PROBE_ACK(true, recon_epoch, recon_shard)
    pre: probing = TRUE;
    probed_epoch ← probed_epoch - 1;
    probed_members ← get(recon_shard, probed_epoch) at CS;
    send PROBE(recon_epoch) to probed_members;

when received NEW_CONFIG(new_epoch, M) from p_j
    status = LEADER;
    (epoch, members, leader)[s_0] ← (e, M, p_j);
    next ← max{k | phase[k] ≠ \_START};
    send NEW_STATE(new_epoch, M, txn, payload, vote, dec, phase) to M \ p_j;

when received NEW_STATE(e, M, txn, payload, vote, dec, phase) from p_j
    pre: e ≥ new_epoch;
    initialized ← TRUE;
    status = FOLLOWER;
    (epoch, members, leader)[s_0] ← (e, M, p_j);
    (txn, payload, vote, dec, phase) ←
        (txn, payload, vote, dec, phase);

when received CONFIG_CHANGE(s, e, M, p_j) from CS
    pre: epoch[s] < e ∧ s ≠ s_0;
    (epoch, members, leader)[s] ← (e, M, p_j);

function retry(k)
    pre: phase[k] = PREPARED;
    all s ∈ shards(txn[k]) do
        send PREPARE(txn[k], l) to leader[s];

Figure 1: Atomic commit protocol at a process p_j in a shard s_0. At line 12 we let L_1 = \{payload[k] | k < next ∧ phase[k] = DECIDED ∧ dec[k] = COMMIT\}; L_2 = \{payload[k] | k < next ∧ phase[k] = PREPARED ∧ vote[k] = COMMIT\}. 
1. If a process \( p_1 \) receives \( \text{ACCEPT}(e, k, t, l, d) \) and replies with \( \text{ACCEPT_ACK} \), then after this and while \( \text{epoch}[s] = e \) at \( p_1 \), we have \( \text{txn}_k \prec \text{txn}_l \), \( \text{vote}_k \prec \text{vote}_l \) and \( \text{payload}_k \prec \text{payload}_l \), where \( \text{txn}_k \), \( \text{vote}_k \) and \( \text{payload}_k \) were the values of the arrays \( \text{txn}, \text{vote} \) and \( \text{payload} \) at the leader of \( s \) at \( e \). Hence, probing requires traversing all the members that have been initialized.

2. Assume that all followers in \( s \) at \( e \) received \( \text{ACCEPT}(e, k, t, l, d) \) and responded to it with \( \text{ACCEPT_ACK} \), and at the time the leader of \( s \) at \( e \) sent the corresponding message \( \text{PREPARE_ACK}(e, s, k, t, l, d) \) it had \( \text{txn}_k = \text{txn} \), \( \text{vote}_k = \text{vote} \) and \( \text{payload}_k = \text{payload} \). Whenever at a process in \( s \) we have \( \text{epoch}[s] = e' \prec e \), we also have \( \text{txn}_k \prec \text{txn} \), \( \text{vote}_k \prec \text{vote} \) and \( \text{payload}_k \prec \text{payload} \).

3. After a process receives \( \text{PROBE}(e) \) and replies with \( \text{PROBE_ACK}(e, s, k, t, l, d) \), it will never send \( \text{ACCEPT_ACK}(e', s, k, t, l, d) \) with \( e' \prec e \).

4. (a) For any messages \( \text{DECISION}(\_, k, d_1) \) and \( \text{DECISION}(\_, k, d_2) \) sent to processes in the same shard, we have \( d_1 = d_2 \).

(b) For any messages \( \text{DECISION}(t, d_1) \) and \( \text{DECISION}(t, d_2) \) sent, we have \( d_1 = d_2 \).

5. Assume that all followers in \( s \) at \( e \) received \( \text{ACCEPT}(e, k, t, l, d) \) and replied with \( \text{ACCEPT_ACK} \). Let the process \( p_j \) be a member of \( s \) at \( e' \) such that \( e' < e \). If \( p_1 \) is not a member of \( s \) at \( e' \) then \( p_1 \) cannot be member of \( s \) at any \( e'' > e' \).

Figure 3: Key invariants of the atomic commit protocol. Given a sequence \( \alpha \), we let \( \text{length}(\alpha) = \max\{k \mid \alpha[k] \neq \bot\} \) and let \( \alpha[k] \) be the prefix of \( \alpha \) of length \( k \). Given a second sequence \( \beta \), we let \( \beta < \alpha \) if \( \text{length}(\beta) \leq \text{length}(\alpha) \) and \( \forall k, \beta[k] \neq \bot \Rightarrow \beta[k] = \alpha[k] \).

ever been initialized. Our protocol guarantees that a shard can become operational, i.e., start accepting transactions, only after all its members have been initialized.

The probing phase is complicated by the fact that there may be a series of failed reconfiguration attempts, where the new leader fails before initializing all its followers. Hence, probing requires traversing epochs from the current one down, skipping epochs that are not operational. Probing selects as the new leader the first initialized process it encounters during this traversal; we can show that this process is guaranteed to know about all transactions accepted at the shard, and thus making it the new leader will preserve Invariant 2 ($\S 4$).

In more detail, a process \( p_r \) initiates a reconfiguration of a shard \( s \) by calling \( \text{reconfigure}(s') \) (line 33). The process picks an epoch number \( \text{recon_epoch} \) higher than the epoch of \( s \) stored in the configuration service and then starts the probing phase, as marked by the flag \( \text{probing} \). The process \( p_r \) keeps track of the shard being reconfigured in \( \text{recon_shard} \), the epoch being probed in \( \text{probed_epoch} \) and the membership of this epoch in \( \text{probed_members} \). The process initializes these variables when it first reads the current configuration from the configuration service (line 36). It then sends a \( \text{PROBE} \) message to the members of the current configuration, asking them to join the new epoch \( \text{recon_epoch} \). Upon receiving a \( \text{PROBE}(e) \) message (line 40), a process first checks that the proposed epoch is equal or higher than the highest epoch it has ever been asked to join, which the process stores in \( \text{new_epoch} \) (we always have \( \text{epoch}[s] \leq \text{new_epoch} \) at a process in \( s \)). In this case, the process sets \( \text{new_epoch} \) to \( e \) and changes its status to \( \text{reconfiguring} \), which causes it to stop transaction processing. It then replies to \( p_r \) with a \( \text{ACCEPT_ACK} \) message, which indicates whether it has been previously initialized or not. If \( p_r \) finds a process that has previously been initialized, and hence can serve as the new leader, \( p_r \) ends probing (line 45). If \( p_r \) does not find such a process in the epoch \( \text{probed_epoch} \) and receives at least one reply \( \text{PROBE_ACK} \) from a process that has not been initialized (line 51), \( p_r \) can conclude that the epoch \( \text{probed_epoch} \) is not operational and will never become such, because it has convinced at least one of its members to join the new epoch; this is formalized by Invariant 3. In this case \( p_r \) starts probing the preceding epoch. Since no transactions could have been accepted at the epoch \( \text{probed_epoch} \), picking a new leader from an earlier epoch will not lose any accepted transactions and thus will not violate Invariant 2.

Once the probing finds a new leader \( p_j \) for the shard \( s \) (line 45), the process \( p_r \) computes the membership of the new configuration using a function \( \text{compute_membership} \) (line 48). We do not prescribe a particular implementation of this function, except that the new membership must contain the new leader \( p_j \) and may only contain the processes that replied to probing or fresh processes. The latter can be added to reach the desired level of fault tolerance. Once the new configuration is computed, \( p_r \) attempts to store it in the
configuration service using a compare-and-swap operation. This succeeds only if the current epoch is still the epoch from which \(p_r\) started probing, which means that no concurrent reconfiguration occurred while \(p_r\) was probing. In this case, \(p_r\) sends a NEW_CONFIG message with the new configuration to the new leader of \(s\).

When the new leader of \(s\) receives the NEW_CONFIG message (line 56), it sets next to the length of its sequence of transactions, epoch[\(s\)] to the new epoch and status to LEADER, which allows it to start processing new transactions. It then sends a NEW_STATE message to its followers, containing its state. Upon receiving this message (line 61), a process overwrites its state with the one provided, changes its status to FOLLOWER, and sets initialized to true. As part of the state update, the process also updates its epoch epoch[\(s\)] to the new one. Hence, the process will not accept transactions from the new leader until it receives the NEW_STATE message.

When a new configuration of a shard \(s\) is persisted in the configuration service, the service sends it in a CONFIG_CHANGE message to the members of shards other than \(s\). A process updates the locally stored configuration upon receiving this message (line 67).

**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 a retry function (line 70). For this, the process just sends a PREPARE(\(t\), \(_\perp\)) message to the leaders of the shards of \(t\), carrying a special undefined value \(_\perp\) as the payload. If a leader receiving PREPARE(\(t\), \(_\perp\)) has already certified \(t\), it re-sends the corresponding PREPARE_ACK message to the new coordinator, including the transaction payload and vote (line 6). Otherwise, if the leader does not have the payload of \(t\), it prepares the transaction as aborted and with an empty payload \(e\) (line 15). In either case, the new coordinator will finish processing the transaction as usual. The above case when the transaction is aborted because the leader of a shard does not know its payload may arise when the old coordinator crashed in between sending PREPARE messages to different shards. Note that if the old coordinator was suspected spuriously and will try later to submit the transaction to a shard where it was aborted, it will just get a PREPARE_ACK message with an ABORT vote.

Our protocol allows any number of processes to become coordinators of a transaction at the same time. Nevertheless, the protocol ensures that they will all reach the same decision, even in case of reconfigurations. We formalize this by Invariant 4: 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 protocol computes a unique decision on each transaction. Invariant 4 is proved as a corollary of Invariant 2.

**Losing undecided transactions.** Recall that our protocol uses the optimization that delegates persisting transactions at followers to coordinators \([1, 9]\). We now highlight how this optimization interacts with transaction certification. Because of the optimization, transactions prepared by a leader of a shard \(s\) can be persisted at followers out of order. For example, \(t_2\) may follow \(t_1\) in the certification order at the leader, but may be persisted at followers first. If now the leader of \(s\) and the coordinator of \(t_1\) crashes before \(t_1\) is persisted at followers, \(t_1\) will be lost forever, something that is allowed by Invariant 2 (due to the use of \(<\)). In this case we lose a transaction \(t_1\) on the basis of which the vote on the transaction \(t_2\) was computed (e.g., the payload \(l_1\) of \(t_1\) was in \(L_2\) when the vote on \(t_2\) was computed at line 12). This does not violate correctness, since the vote on \(t_2\) makes sense also in the context excluding \(t_1\): due to distributivity of certification functions (§2), if \(t_2\) was allowed to commit in the presence of \(t_1\) \(f_p((l_1, l_2) = \text{commit})\), it can also commit in its absence \(f_p((0, l_2) = \text{commit})\). Note that in this case a decision on \(t_1\) could not have been exposed to the client: otherwise \(t_1\) could not get lost due to Invariant 2. Also note that, since we assume the transaction execution component produces payloads with read-sets containing only values written by committed transactions (§2), in the above case \(t_2\) could not have read a value written by \(t_1\).

**4 CORRECTNESS**

The next theorem states the safety of our protocol, showing that it implements the TCS specification.

**Theorem 4.1.** A transaction certification service implemented using the protocol in Figure 1 is correct with respect to a certification function \(f\) matching the shard-local certification functions \(f_s\) and \(g_s\).

We defer the proof to [4] and only sketch the proof of the key Invariant 2. This relies on auxiliary Invariant 5, which we prove first.

**Proof sketch for Invariant 5.** We prove the invariant by induction on \(e''\). Assume that the invariant holds for all \(e'' < e'\). We now show it for \(e'' = e'\). The members of \(s\) at \(e'\) are computed at line 48 by a reconfiguring process \(p_r\) using the compute_membership function, which returns either fresh processes or processes that responded to \(p_r\)’s probing. Since \(p_r\) was a member of \(s\) at \(e' < e'\), it is not fresh; then by assumptions on compute_membership \(p_r\) must have received PROBE(e's) from \(p_r\) and replied with PROBE_ACK(\(e\), \(e'\), \(s\)). The process \(p_r\) starts probing at epoch \(e'' = 1\) and ends it upon receiving a PROBE_ACK(\(true\), \(e'\), \(s\)) message. By the induction hypothesis, \(p_r\) is not a member of \(s\) at any epoch from \(e'' = 1\) down to \(e + 1\). Hence, if the probing stops before reaching \(e\), then \(p_r\) will not be a member of \(s\) at \(e''\), as required. Assume now that the probing reaches \(e\). By Invariant 3, each follower in \(s\) at \(e\) must have sent ACCEPT_ACK(\(s\), \(e\), \(t\)) before receiving PROBE(e'). Then any member of \(s\) at \(e\) receiving PROBE(e') will have initialized = \(true\). Hence, if any member of \(s\) at \(e\) replies with PROBE_ACK(\(initialized\), \(e'\), \(s\)), we have that initialized = \(true\). Since the process \(p_r\) will not move to the preceding epoch until at least one process replies with PROBE_ACK, this means that the probing can never go beyond \(e\). Since the process \(p_r\) is not a member of \(e\), it cannot be included as a member of \(s\) in \(e'\), as required. □

**Proof sketch for Invariant 2.** We prove the invariant by induction on \(e'\). Assume that the invariant holds for all \(e' < e'\). We now show it for \(e' = e''\) by induction on the length of the protocol execution. We only consider the most interesting transition in line 56, when a process \(p_t\) becomes a leader of \(s\) at an epoch \(e''\). We show that after this transition at \(p_t\) we have txnistk < txn, votek < vote and payloadk < payload.
Since $p_l$ was chosen as the leader of $s$ at $e''$, this process replied
with PROBE_ACK(true, $e''$, s) to a PROBE($e''$). Therefore, $p_l$ was a
member of $s$ at an epoch $e'' < e''$ that was being probed. Probing
ends when at least one process sends a PROBE_ACK(true, $e''$, s).
From Invariant 3 and the assumption that all followers in $e$ replied
with ACCEPT_ACK to ACCEPT($e$, $k$, t, l, d), we can conclude that probing
could no have gone further than $e$. Hence, $e < e'' < e''$.

Let $e_0$ be the value of epoch at $p_l$ right before the transition
at line 56. We have $e_0 \geq e$, as otherwise $p_l$ would not be a member
of $s$ at $e$ and by Invariant 5 could not be picked as the leader of $s$ at $e''$. It is also easy to show that $e_0 < e''$. Hence, $e < e_0 < e''$.

If $e < e_0$, then by the induction hypothesis, we have $txn|_k < txn,$
vote|_k < vote and payload|_k < payload right after the transition
in line 56, as required. Assume now that $e_0 = e$. If $p_l$ was the leader
of $s$ at $e$, then we trivially have $txn|_k < txn$, vote|_k < vote and
payload|_k < payload right after the transition in line 56, as required.
Otherwise, by Invariant 3, $p_l$ must have received ACCEPT_ACK($e$, $k$, t, l, d)
and responded to it with ACCEPT_ACK($s$, $e$, $t$) before the transition
in line 56. Then the required follows from Invariant 1. □

We next state liveness properties of our protocol (we again defer
dependence proofs to [4]). The reconfiguration procedure in the protocol will
get stuck if it cannot find an initialized process, which may happen
if enough processes crash, so that all shard data is lost. We now
state conditions under which this cannot happen. We associate
two events with each configuration $e$ of a shard $s$: introduction and
activation. Introduction indicates that the configuration comes into
existence and is triggered when the configuration is successfully
persisted in the configuration service (line 49). Activation indicates
that the configuration becomes operational and is triggered when all
the followers of the configuration have processed the NEW_STATE
messages sent by its leader (line 61).

Once a configuration has been activated, we say that it is active.
We define its lifetime as the time interval between its introduction
and when a succeeding configuration becomes active. Note that not
every introduced configuration necessarily becomes active, since
its leader may never complete the data transfer to the followers.
To ensure our protocol is live we make the following assumption,
similar to the ones made by other protocols with changing mem-
bership [3, 26].

**Assumption 1.** At least one member in each configuration is non-
faulty throughout the lifetime of a configuration.

The following two theorems show that, under this assumption,
a single reconfiguration makes progress.

**Theorem 4.2.** If a process $p_r$ attempts to reconfigure a shard $s$
and no other process attempts to reconfigure $s$ simultaneously, then if
$p_r$ is non-faulty for long enough, it will eventually introduce a new
configuration.

**Theorem 4.3.** If a configuration of a shard $s$ is introduced by a
process $p_r$, then it will eventually be activated, provided no process
attempts to reconfigure $s$ simultaneously, and $p_r$ and all the members
of the configuration are non-faulty for long enough.

Finally, the following theorem shows that in the absence of fail-
ures or reconstructions, transaction certification makes progress.

**Theorem 4.4.** Assume that the current configuration of each shard
is active, all processes are aware of the current configuration of each
shard, and no reconfiguration is in progress. If a transaction is sub-
mited for certification, then it will eventually be decided, provided
no reconfiguration is attempted and all the processes belonging to the
current configuration of each shard are non-faulty for long enough.

5 EXPLOITING RDMA

We now present a variant of our protocol that uses Remote Direct
Memory Access (RDMA), which follows the design of the FARM
system [9, 24]. By comparing this protocol with that of §3 we high-
light the trade-offs required by the use of RDMA. Due to space
constraints, we defer the pseudocode of our protocol to [4] and
describe the required changes in the protocol of §3 only informally.

We assume the same system model as in §2, except that pro-
desses can communicate using RDMA. This allows a machine to
access the memory of another machine over the network without
involving the latter’s CPU, thus lowering latency. Like FARM, our
protocol uses RDMA to implement a primitive for point-to-point
communication between processes with the following interface.
The primitive allows a sender process to reliably send a message
$m$ to a receiver process $p_j$ (send-rdma($m$, $p_j$)) by remotely writing
into a specific memory region of $p_j$. The receiver then gets an
acknowledgement when the message reaches the receiver’s memory
(ack-rdma($m$, $p_j$)), sent by the receiver’s network interface card
(NIC) without interrupting its CPU. The receiver is notified at a
later point that a new message is available (deliver-rdma($m$, $p_j$)).

Hence, the guarantee provided by ack-rdma($m$, $p_j$) is that the re-
receiver will eventually deliver the message $m$, even if the sender
causes, since the message is already in the receiver’s memory. The
operation open($p_j$) grants $p_j$ access to a region of the caller’s mem-
ory, and close($p_j$) revokes it. Once the latter operation completes,$p_j$
cannot send any message to the caller using send-rdma. Finally,
we assume that the communication primitive includes another op-
eration: flush. This operation blocks the caller until it has delivered
all messages addressed to it that have been acknowledged by its
NIC through an ack-rdma.

To implement the above primitive, the receiver usually keeps a
circular buffer in memory for each process that may send it a mes-
sage [8, 21]. The operation send-rdma($m$, $p_j$) issued by a process
$p_l$ appends a message to the corresponding buffer at the receiver
using RDMA writes. Receivers periodically pull messages from the
buffers and deliver them to the application via deliver-rdma. If a
buffer at a process $p_j$ gets full, the associated sender process will not
be able to send a message to $p_j$ until the latter pulls some messages.

Following FARM, we use the above RDMA-based communication
primitive in our protocol to persist votes and decisions (steps 2
and 3 of Figure 2a). This requires the following changes to
the protocol in Figure 1. First, ACCEPT and DECISION messages are sent
using send-rdma instead of send (lines 20 and 29). Second, the
followers do not send explicit ACCEPT_ACK messages to transaction
coordinators (line 25); instead, the latter act once they receive an
RDMA acknowledgement ack-rdma. This makes the checks at
lines 22 and 31 redundant, as followers cannot reject ACCEPT or
DECISION messages under any circumstance. The practical rationale
for these changes is that persisting a transaction $t$ at followers using
RDMA minimizes the time during which the transaction is prepared at leaders, which requires them to vote abort on all transactions conflicting with \( t \) (via the certification function \( q_t \), §2); this results in lower abort rates [2, 9]. Transaction processing at followers (e.g., adding them to the local copy of the certification order, line 21) is done off the critical path of certification.

Unfortunately, the above changes to the failure-free path of the protocol do not preserve correctness without changes to reconfiguration, as illustrated by an example execution in Figure 4a. In this execution, two shards \( s_1 \) and \( s_2 \) are involved in the certification of a transaction \( t \), coordinated by a process \( p_c \) from a third shard. The transaction is prepared to commit at the leaders \( p_1 \) and \( p_3 \) of both shards (step 1), and the \texttt{commit} vote from the leader of \( s_1 \) (\( p_1 \)) is persisted at the follower \( p_2 \) using RDMA (step 2). Before the coordinator \( p_c \) persists the vote from the leader \( p_3 \) of \( s_2 \) at the follower \( p_4 \), the leader \( p_3 \) is suspected of failure and a reconfiguration is triggered at shard \( s_2 \). This promotes the follower \( p_4 \) to a new leader and brings online a fresh follower \( p_5 \). Next, the leader \( p_1 \) of \( s_1 \) suspects the coordinator \( p_c \) of failure and triggers a reconfiguration to remove it. Once \( p_c \) is removed from its shard, \( p_1 \) retries the processing of \( t \) (step 3, line 70 in Figure 1). The new leader \( p_4 \) of \( s_2 \) does not know about \( t \), so this results in the transaction being aborted, because its payload at shard \( s_2 \) is thought to be lost (steps 4 and 5). But now the coordinator \( p_c \), who did not actually fail and still believes \( s_2 \) is in the old configuration, finishes its processing by persisting the \texttt{commit} vote of the old leader \( p_3 \) of \( s_2 \) at the old follower \( p_4 \), which is now the new leader of \( s_2 \) (step 6). Since this is done via RDMA, \( p_4 \) cannot reject the vote and, thus, \( p_c \) commits the transaction (step 7). This violates safety, as two contradictory results have been externalized. The protocol in §3 is not subject to this problem, because in that protocol the new leader \( p_4 \) of the shard \( s_2 \) would reject the \texttt{ACCEPT} message due to the failure of the check at line 22.

To make the RDMA-based protocol correct, we need to change the reconfiguration protocol so that the whole system participates in reconfiguration instead of just the affected shard. Figure 4b illustrates the message flow of the redesigned reconfiguration protocol.
We now show how the revised reconfiguration protocol prevents the bug in Figure 4a. In this protocol, when $p_c$ attempts to persist the commit vote at $p_1$ (step 3), the latter will be already aware that $p_c$ has been removed from the system and will close the RDMA connection to it. Thus, $p_c$ will be unable to persist the vote at $p_4$ (this would violate the property 1) and will never gather enough acknowledgements to decide the transaction. Hence, no contradictory results will be externalized. We state and prove the correctness of the RDMA-based protocol in [4].

6 RELATED WORK AND DISCUSSION

Our protocols are inspired by the recent FARM system for transaction processing, which also uses $f + 1$ replicas per shard and deals with failures using reconfiguration [9, 24]. FARM was presented as a complete database system with a number of optimizations, including the use of RDMA. In contrast, our work distills the core ideas of FARM into protocols solving the well-defined transaction certification problem, parametric in the isolation level provided and rigorously proven correct. This allows us to simplify some aspects of the FARM design. In particular, FARM has a more complex way of determining the state of the new leader upon a reconfiguration, which merges the states from all surviving replicas of the previous configuration. In contrast, our protocols take the state of any single initialized replica. Our reconfiguration protocols also provide better fault-tolerance guarantees on a par with those of existing ones [3, 26]. This is because, like Vertical Paxos [19], our protocols look through a sequence of configurations to find the new leader, whereas FARM only considers the previous configuration. Hence, FARM reconfiguration can get stuck even when there exists a non-faulty replica with the necessary data. Finally, by presenting two related protocols using message passing and RDMA, we are able to identify the price of exploiting RDMA—having to reconfigure the whole system instead of a single shard.

There have been a number of protocols for solving the atomic commit problem, which requires reaching a decision on a single transaction [10, 12, 13, 25]. In contrast to these works, our protocol solves the more general problem of implementing a Transaction Certification Service, which requires reaching decisions on a stream of transactions. This problem more faithfully reflects the requirements of modern transaction processing systems [6].

Our protocol weaves together two-phase commit (2PC) [12] and Vertical Paxos [19], instead of using Paxos replication as a black box. This is similar to several existing shared systems for transaction processing, which integrate protocols for distribution and replication [6, 16, 22, 28]. However, these systems considered a static set of $2f + 1$ processes per shard, whereas we assume $f + 1$ processes and allow the system to be reconfigured. Achieving this correctly is nontrivial and requires a subtle interplay between the reconfigurable replication mechanism and cross-shard coordination. For example, as we explained in §3, on failures our protocol may lose information about transactions that influenced votes on other transactions, but this does not violate correctness. As is well-known [20], using $f + 1$ instead of $2f + 1$ replicas results in somewhat weaker availability guarantees: upon a single failure, our protocols have to stop processing transactions while the system is reconfigured.

Acknowledgments. We thank Dushyanth Narayanan for discussions about FARM. This research was supported by an ERC grant RACCOON.

REFERENCES


