Basil: Breaking up BFT with ACID (transactions)

Florian Suri-Payer*, Matthew Burke*, Zheng Wang*, Yunhao Zhang*, Lorenzo Alvisi*, Natacha Crooks†

*Cornell University, †UC Berkeley

Abstract

This paper presents Basil, the first transactional, leaderless Byzantine Fault Tolerant key-value store. Basil leverages ACID transactions to *scalably* implement the abstraction of a trusted shared log in the presence of Byzantine actors. Unlike traditional BFT approaches, Basil executes non-conflicting operations in parallel and commits transactions in a single round-trip during fault-free executions. Basil improves throughput over traditional BFT systems by four to five times, and is only four times slower than TAPIR, a non-Byzantine replicated system. Basil's novel recovery mechanism further minimizes the impact of failures: with 30% Byzantine clients, throughput drops by less than 25% in the worst-case.

1   Introduction

This paper presents Basil, a leaderless transactional key-value store that scales the abstraction of a Byzantine-fault tolerant shared log.

Byzantine fault-tolerance (BFT) systems enable safe online data sharing among mutually distrustful parties as they guarantee correctness in the presence of malicious (Byzantine) actors. These platforms offer exciting opportunities for a variety of applications, including healthcare [93], financial services [5, 6, 9], and supply chain management [7, 34]. Consider the iPhone supply chain: it spans three continents and hundreds of contractors [4] who may not trust each other, yet must be willing to share product information. BFT replicated state machines [24, 28, 43, 50, 101] and permissioned blockchains [8, 13, 17, 23, 41, 49, 92] are at the core of these new services: they ensure that mutually distrustful parties produce the same totally ordered log of operations.

The abstraction of a totally ordered log is appealingly simple. A *scalable* totally ordered log, however, is not only hard to implement (processing all requests sequentially can become a bottleneck), but also often unnecessary. Most distributed applications primarily consist of logically concurrent operations; supply chains for instance, despite their name, are actually complex networks of independent transactions.

Some BFT systems use sharding to try to tap into this parallelism. Transactions that access disjoint shards can execute concurrently but operations within each shard are still totally ordered. Transactions involving multiple shards are instead executed by running cross-shard atomic commit protocols, which are layered above these totally ordered shards [13, 49, 78, 79, 102]. The drawbacks of these systems are known: (*i*) they pay the performance penalty of redundant coordination—both across shards (to commit distributed transactions) and among the replicas within each shard (to totally order in-shard operations) [75, 103, 104]; (*ii*) within each shard, they give a leader replica undue control over the total order ultimately agreed upon, raising fairness concerns [46, 101, 105]; (*iii*) and often they restrict the expressiveness of the transactions they support [78, 79] by requiring that the read and write set be known in advance.

In this paper, we advocate a more principled, performant, and expressive approach to supporting the abstraction of a totally ordered log at the core of all permissioned blockchain systems.

We make our own the lesson of distributed databases, which successfully leverage generic, interactive transactions to implement the abstraction of a sequential, general-purpose log. These systems specifically design highly concurrent protocols that are *equivalent* to a serial schedule [21, 80]. Byzantine data processing systems need be no different: rather than aiming to sequence all operations, they should decouple the *abstraction* of a totally ordered sequence of transactions from its *implementation*. Thus, we flip the conventional approach: instead of building database-like transactions on top of a sharded, totally ordered BFT log, we directly build out this log abstraction above a partially-ordered distributed database, where total order is demanded only for conflicting operations.

To this effect, we design Basil, a serializable BFT key-value store that implements the abstraction of a trusted shared log, whose novel design addresses each of the drawbacks of traditional BFT systems: (*i*) it borrows databases' ability to leverage concurrency control to support highly concurrent but serializable transactions, thereby adding parallelism to the log; (*ii*) it sidesteps concerns about the fairness of leader-based systems by giving clients the responsibility of driving the execution of their own transactions; (*iii*) it eliminates redundant coordination by integrating distributed commit with replication [75, 104] so that, in the absence of faults and contention, transactions can return to clients in a single round trip; and (*iv*) it improves the programming API, offering support for general interactive transactions that do not require a-priori knowledge of reads and writes.

We lay the foundations for Basil by introducing two complementary notions of correctness. *Byzantine isolation* focuses on safety: it ensures that correct clients observe a state of the database that could have been produced by correct clients alone. *Byzantine independence* instead safeguards liveness: it limits the influence of Byzantine actors in determining whether a transaction commits or aborts. To help enforce these two notions, and disentangle correct clients from the maneuvering of Byzantine actors, Basil's design follows the

1

principle of *independent operability*: it enforces safety and liveness through mechanisms that operate on a per-client and per-transaction basis. Thus, Basil avoids mechanisms that enforce isolation through pessimistic locks (which would allow a Byzantine lock holder to prevent the progress of other transactions), adopting instead an optimistic approach to concurrency control.

Embracing optimism in a Byzantine setting comes with its own risks. Optimistic concurrency control protocols [19, 53, 86, 97, 104] are intrinsically vulnerable to aborting transactions if they interleave unfavorably during validation, and Byzantine faults can compound this vulnerability. Byzantine actors may, for instance, intentionally return stale data, or collude to sabotage the commit chances of correct clients' transactions. Consider multiversioned timestamp ordering (MVTSO) [19, 86], which allows writes to become visible to other operations before a transaction commits. While this choice helps reduce abort rates for contended workloads, it can cause transactions to stall on uncommitted operations.

Basil's ethos of independent operability is key to mitigating this issue. The system implements a variant of MVTSO that prevents Byzantine participants from unilaterally aborting correct clients' transactions and includes a novel *fallback* mechanism that empowers clients to finish pending transactions issued by others, while preventing Byzantine actors from dictating their outcome. Importantly, this fallback is a per-transaction recovery mechanism: thus, unlike traditional BFT view-changes, which completely suspend the normal processing of all operations, it can take place without blocking non-conflicting transactions.

Our results are promising: on TPC-C [94], Retwis [3] and Smallbank [35]), Basil's throughput is 3.5-5 times higher than layering distributed commit over totally ordered shards running BFT-SMaRt, a state-of-art PBFT implementation [22] and HotStuff [101] (Facebook's Libra's core consensus protocol). BFT's cryptographic demands, however, still cause Basil to be 2-4 times slower than TAPIR, a recent non-Byzantine distributed database [104]. In the presence of Byzantine clients, Basil's performance degrades gracefully: with 30% Byzantine clients, Basil's throughput drops by less than 25% in the worst-case. In summary, this paper makes the following three contributions:

- It introduces the complementary correctness notions of Byzantine isolation and Byzantine independence.
- It presents novel concurrency control, agreement, and fallback protocols that balance the desire for high-throughput in the common case with resilience to Byzantine attacks.
- It describes Basil, a BFT database that guarantees Byzserializability while preserving Byzantine independence. Basil supports interactive transactions, is leaderless, and achieves linear communication complexity.

2 Model and Definitions

We introduce the complementary and system-independent notions of Byzantine isolation and Byzantine independence, which, jointly formalize the degree to which a Byzantine actor can affect transaction progress and safety.

2.1 System Model

Basil inherits the standard assumptions of prior BFT work. It assumes that the number of faulty replicas within a shard does not exceed a threshold f and that an arbitrary number of clients may be faulty. But for this bound, we make no further assumption about the pattern of failures across shards. We assume that applications authenticate clients and can subsequently audit their actions. Faulty clients and replicas may deviate arbitrarily from their specification; a strong but static adversary can coordinate their actions but cannot break standard cryptographic primitives. Similar to other BFT systems [23, 24, 28, 39, 50], Basil makes no synchrony assumption for safety but for liveness [38] depends on partial synchrony [37]. Basil also inherits some of the limitations of prior BFT systems: it cannot prevent authenticated Byzantine clients, who otherwise follow the protocol, from overwriting correct clients' data. It additionally assumes that, collectively, Byzantine and correct clients have similar processing capabilities, and thus Byzantine clients cannot cause a denial of service attack by flooding the system.

2.2 System Properties

To express Basil's correctness guarantees, we introduce the notion of *Byzantine isolation*. Database isolation (serializability, snapshot isolation, etc.) traditionally regulates the interaction between concurrently executing transactions; Byzantine isolation ensures that, even though Byzantine actors may choose to violate ACID semantics, the state observed by correct clients will always be ACID compliant.

We start from the standard notions of transactions and histories introduced by Bernstein et al. [20]. We summarize them here and defer a more formal treatment to the Appendix B. A transaction T contains a sequence of read and write operations terminating with a commit or an abort. A history H is a partial order of operations representing the interleaving of concurrently executing transactions, such that all conflicting operations are ordered with respect to one another. Additionally, let C be the set of all clients in the system; $Crct \subseteq C$ be the set of all correct clients; and $Byz \subseteq C$ be the set of all Byzantine clients. A projection $H|_\mathscr{C}$ is the subset of the partial order of operations in H that were issued by the set of clients $\mathscr{C}$. We further adopt standard definitions of database isolation: a history satisfies an isolation level I if the set of operation interleavings in H is allowed by I. Drawing from the notions of BFT linearizability [65] and view serializability [20], we then define the following properties:

Legitimate History History H is *legitimate* if it was generated by correct participants, *i.e.*, $H = H_{Crct}$.

Correct-View Equivalent History H is *correct-view* equivalent to a history H' if all operation results, commit decisions, and final database values in $H|_{Crct}$ match those in H'.

Byz-I Given an isolation level I, a history H is *Byz-I* if there exists a legitimate history H' such that H is correct-view equivalent to H' and H' satisfies I.

This definition is not Basil-specific, but captures what it means, for any Byzantine-tolerant database, to enforce the guarantees offered by a given isolation level I. Informally, it requires that the states observed by correct clients be explainable by a history that satisfies I and involves only correct participants. It intentionally makes no assumptions on the states that Byzantine clients choose to observe.

Basil specifically guarantees Byz-serializability: correct clients will observe a sequence of states that is consistent with a sequential execution of concurrent transactions. This is a strong safety guarantee, but it does not enforce application progress; a Byz-serializable system could still allow Byzantine actors to systematically abort all transactions. We thus define the notion of *Byzantine independence*, a general system property that bounds the influence of Byzantine participants on the outcomes of correct clients' operations.

Byzantine Independence For every operation o issued by a correct client c, no group of participants containing solely Byzantine actors can unilaterally dictate the result of o.

In a context where clients issue transaction operations, Byzantine independence implies, for instance, that Byzantine actors cannot collude to single-handedly abort a correct client's transaction. This is a challenging property to enforce. It cannot be attained in a leader-based system: if the leader and a client are both Byzantine, they can collude to prevent a transaction from committing by strategically generating conflicting requests. In contrast, Basil can enforce Byzantine independence as long as Byzantine actors do not have full control of the network, a requirement that is in any case a precondition for any BFT protocol that relies on partial synchrony [39, 72]. We prove in Appendix B that:

Theorem 1. *Basil maintains Byz-serializability.*

Theorem 2. *Basil maintains Byzantine independence in the absence of a strong network adversary.*

Basil is designed for settings where Byzantine attacks can occur, but are infrequent, consistent with the prevalent assumption for permissioned blockchains today; namely, that to maintain standing in a permissioned system, clients are unlikely to engage in actively detectable Byzantine behavior [44] and, if they cannot break safety undetected, it is preferable for them to be live [67]. We design Basil to be particularly efficient during gracious executions [28] (*i.e.*, synchronous and fault-free) while bounding overheads when misbehavior does occur. In particular, we design aggressive concurrency control mechanisms that maximize common case

performance by optimistically exposing uncommitted operations, but ensure that these protocols preserve independent operability, so that Basil can guarantee continued progress under Byzantine attacks [28]. We confirm this experimentally in Section 6.
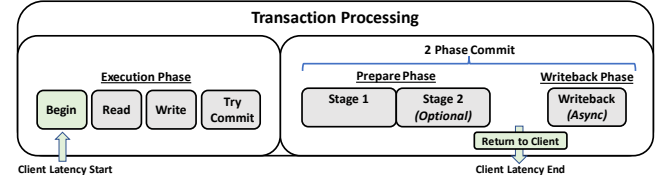
3 System Overview

Figure 1. Basil Transaction Processing Overview

Basil is a transactional key-value store designed to be scalable and leaderless. Our architecture reflects this ethos.

Transaction Processing Transaction processing is driven by clients (avoiding costly all-to-all communications amongst replicas) and consists of three phases (Figure 1). First, in an *Execution phase*, clients execute individual transactional operations. As is standard in optimistic databases, reads are submitted to remote replicas while writes are buffered locally. Basil supports interactive and cross-shard transactions: clients can issue new operations based on the results of past operations to any shard in the system. In a second *Prepare phase*, individual shards are asked to vote on whether committing the transaction would violate serializability. For performance, Basil allows individual replicas within a shard to process such requests out of order. Finally, the client aggregates each shard vote to determine the outcome of the transaction, notifies the application of the final decision, and forwards the decision to the participating replicas in an asynchronous *Writeback phase*. Importantly, the decision of whether each transaction commits or aborts must be preserved across both benign and Byzantine failures. We describe the protocol in Section 4.

Transaction Recovery A Byzantine actor could begin executing a transaction, run the prepare phase, but intentionally never reveal its decision. Such behavior could prevent other transactions from making progress. Basil thus implements a *fallback recovery mechanism* (§5) that can terminate stalled transactions while preserving Byz-serializability. This protocol, in the common case, allows clients to terminate stalled transactions in a single additional round-trip.

Replication Basil uses $n = 5f + 1$ replicas for each shard. This choice allows Basil to (*i*) preserve Byzantine independence (*ii*) commit transactions in a single round-trip in the absence of contention, and (*iii*) reduces the message complexity of transaction recovery by a factor of n, all features which would not be possible with a lower replication factor.

4 Transaction Processing

Basil takes as its starting point MVTSO [19], an aggressive multiversioned concurrency control, and modifies it in three ways: (*i*) in the spirit of independent operability, it has clients drive the protocol execution; (*ii*) it merges concurrency control with replication; and finally (*iii*) it hardens the protocol against Byzantine attacks to guarantee Byz-serializability while preserving Byzantine independence.

Traditional MVTSO works as follows. A transaction T is assigned (usually by a transaction manager or scheduler) a unique timestamp ts_T that determines its serialization order. As MVTSO is multiversioned, writes in T create new versions of the objects they touch, tagged with ts_T. Reads instead return the version of the read object with the largest timestamp smaller than ts_T and update that object's *read timestamp* (RTS) to ts_T. Read timestamps are key to preserving serializability: to guarantee that no read will miss a write from a transaction that precedes it in the serialization order, MVTSO aborts all writes to an object from transactions whose timestamp is lower than the object's RTS.

MVTSO is an optimistic protocol, and, as such, much of its performance depends on whether its optimistic assumptions are met. For example, it uses timestamps to assign transactions a serialization order a-priori, under the assumption that those timestamps will not be manipulated; further, it allows read operations to become *dependent* on values written by ongoing transactions under the expectation that they will commit. This sunny disposition can make MVTSO particularly susceptible to Byzantine attacks. Byzantine clients could use artificially high timestamps to make lower-timestamped transactions less likely to commit; or they could simply start transactions that write to large numbers of keys and never commit them: any transaction dependent on those writes would be blocked too. At the same time, by blocking on dependencies (rather than summarily aborting, as OCC would do) MVTSO leaves open the possibility that blocked transactions may be rescued and brought to commit. In the remainder of this section, we describe how Basil, capitalizing on this possibility, modifies MVTSO to harden it against Byzantine faults.

4.1 Execution Phase

Begin() A client begins a transaction T by optimistically choosing a timestamp $ts := (Time, ClientID)$ that defines a total serialization order across all clients. Allowing clients to choose their own timestamps removes the need for a centralized scheduler, but makes it possible for Byzantine clients to create transactions with arbitrarily high timestamps: objects read by those transactions would cause conflicting transactions with lower timestamps to abort. To defend against this attack, replicas accept transaction operations if and only if their timestamp is no greater than $R_{Time}+\delta$, where R_{Time} is the replica's own local clock. Neither Basil's safety nor its liveness depend on the specific value of δ, though a well-chosen

value will improve the system's throughput. In practice we choose δ based on the skew of NTP's clock.

Write(key,value) Writes from uncommitted transactions raise a dilemma. Making them readable empowers Byzantine clients to stall all transactions that come to depend on them. Waiting to disclose them only when the transaction commits, however, increases the likelihood that concurrent transactions will abort. We adopt a middle ground: we buffer writes locally until the transaction has finished execution, and make them visible during the protocol's Prepare phase (we call such writes *prepared*). This approach allows us to preserve much of the performance benefits of early write disclosure while enforcing independent operability (§4.2).

Read(key) In traditional MVTSO, a read for transaction T returns the version of the read object with the highest timestamp smaller than ts_T. When replicas process requests independently, this guarantee no longer holds, as the write with the largest timestamp smaller than ts_T may have been made visible at replica R, but not yet at R': reading from the latter may result in a stale value. Hence, to ensure serializability, transactions in Basil go through a concurrency control check at each replica as part of their Prepare phase (§ 4.2). Further care is required, as Byzantine replicas could intentionally return stale (or imaginary!) values that would cause transactions to abort, violating Byzantine independence. These considerations lead us to the following read logic:

> **1: C → R**: Client C sends read request to replicas.

C broadcasts an authenticated read request $m = \langle \text{READ}, key, ts_T \rangle$ to $\geq 2f + 1$ replicas for shard S.

> **2: R → C**: Replica processes client read and replies.

Each replica R verifies that the request's timestamp is smaller than $R_{Time} + \delta$. If not, it ignores the request; otherwise, it updates *key*'s RTS to ts_T. Basil may evict clients with a history of reading but never committing the transaction. Then R returns a signed message $\langle Committed, Prepared \rangle_{\sigma_R}$ that contains, respectively, the latest committed and prepared versions of *key* at R with timestamps smaller than ts_T. *Committed* ≡ (*version*, C-CERT) includes a *commit certificate* C-CERT (§ 4.3) proving that *version* has committed, while *Prepared* ≡ (*version*, $id_{T'}$, $Dep_{T'}$) includes a digest identifier for T' (§ 4.2) and the write-read dependencies $Dep_{T'}$ of the transaction T' that created *version*. T' cannot commit unless all the transactions in $Dep_{T'}$ commit first.

> **3: C ← R**: Client receives read replies.

A client waits for at least $f + 1$ replies (to ensure that at least one comes from a correct replica) and chooses the *highest-timestamped* version that is *valid*. For committed versions, the criterion for validity is straightforward: a committed version must contain a valid C-CERT. For prepared versions instead, we require that the same version be returned by at least $f + 1$ replicas. Both the validity and timestamp requirement are

important for Byzantine independence. Message validity protects the client's transaction from becoming dependent on a version fabricated by Byzantine replicas; and, by choosing the valid reply with the highest-timestamp, the client is certain to never read a version staler than what it could have read by accessing a single correct replica.

The client then adds the selected (*key, version*) to $ReadSet_T$. If *version* was prepared but not committed, it adds a new write-read dependency to the dependency set Dep_T. Specifically, the client adds to Dep_T a tuple (*version*, $id_{T'}$), which will be used during T's Prepare phase to validate that T is claiming a legitimate dependency.

After T has completed execution, the application tells the client whether it should abort T or instead try to commit it:

Abort() The client asks replicas to remove its read timestamps from all keys in $ReadSet_T$. No actions need to be taken for writes, as Basil buffers writes during execution.

Commit() The client initiates the Prepare phase, discussed next, which performs the first phase of the multi-shard two-phase commit (2PC) protocol that Basil uses to commit T.

Algorithm 1 MVTSO-Check(T)

1: **if** $ts_T > localClock + \delta$ **then**
2:     **return** Vote-Abort
3: **if** $\exists$ invalid $d \in Dep_T$ **then**
4:     **return** Vote-Abort
5: **for** $\forall key, version \in ReadSet_T$ **do**
6:     **if** $version > ts_T$ **then return** MisbehaviorProof
7:     **if** $\exists T' \in Committed \cup Prepared : key \in WriteSet_{T'}$
          $\wedge\, version < ts_{T'} < ts_T$ **then**
8:         **return** Vote-Abort, *optional: (T', T'.C-CERT)*
9: **for** $\forall key \in WriteSet_T$ **do**
10:     **if** $\exists T' \in Committed \cup Prepared :$
          $ReadSet_{T'}[key].version < ts_T < ts_{T'}$ **then**
11:         **return** Vote-Abort, *optional: (T', T'.C-CERT)*
12:     **if** $\exists RTS \in key.RTS : RTS > ts_T$ **then**
13:         **return** Vote-Abort
14:    $Prepared.add(T)$
15:    **wait** *for all pending dependencies*
16: **if** $\exists\, d \in Dep_T : d.decision = Abort$ **then**
17:    $Prepared.remove(T)$
18:     **return** Vote-Abort
19: **return** Vote-Commit

4.2 Prepare Phase

To preserve independent operability, Basil delegates the responsibility for coordinating the 2PC protocol to clients. For a given transaction T, the protocols begins with a Prepare phase, which consists of two stages (Figure 1).

In stage ST1, the client collects *commit or abort votes* from each shard that T accesses. Determining the vote of a shard in turn requires collecting votes from all the shard's replicas.

To avoid the overhead of coordinating replicas within a shard, Basil lets each replica make its determination independently by running a local *concurrency control check*. The flip side is that, since transactions may reach replicas in different orders, even correct replicas within the same shard may not necessarily reach the same conclusion about T. Client C tallies replica votes to learn the vote of each shard and based on how shards voted, determines whether T will commit or abort.

Stage ST2 ensures that C's decision is made durable (or *logged*) across failures. *C logs* the evidence on one shard only. In the absence of contention or failures, Basil's *fast path* guarantees that T's decision is already durable and this explicit logging step can be omitted.

Stage 1: Aggregating votes

> **1: C → R**: Client sends an authenticated ST1 request to all replicas in S.

The message format is ST1 $:= \langle$PREPARE, $T\rangle$, where T consists of the transaction's *metadata* $:= ts_T$, $ReadSet_T$, $WriteSet_T$, Dep_T, and of its identifier id_T. To ensure Byzantine clients neither spoof the list of involved shards nor equivocate T's contents, id_T is a cryptographic hash of T's *metadata*.

> **2: R ← C**: Replica receives a ST1 request and executes the concurrency control check.

Traditional, non-replicated, MVTSO does not require any validation as transactions are guaranteed to observe all the writes that precede them in the serialization order (any "late" write is detected by read timestamps and the corresponding transaction is aborted). This is no longer true in a replicated system: reads could have failed to observe a write performed on a different replica. Basil thus runs an additional concurrency control check to determine whether a transaction T should commit and preserve serializability (Algorithm 1). It consists of seven steps:

① T's timestamp is within the R's time bound (Lines 1-2).
② T's dependencies are valid: R has either prepared or committed every transaction identified by T's dependencies, and the versions that caused the dependencies were produced by said transactions (Lines 3-4).
③ Reads in T did not miss any writes. Specifically, the algorithm (Lines 7-8) checks that there does not exist a write from a committed or prepared transaction T' that (*i*) is more recent than the version that T's read and (*ii*) has a timestamp smaller than ts_T (implying that T should have observed it).
④ Writes in T do not cause reads in other *prepared or committed* transactions to miss a write (Lines 9-11).
⑤ Writes in T do not cause reads in *ongoing* transactions to miss a write: T is aborted if there exists an RTS greater than ts_T (Lines 12-13).
⑥ T is prepared and made visible to future reads (Line 14).
⑦ All transactions responsible for T's dependencies have

reached a decision. R votes to commit T only if all of its dependencies commit; otherwise it votes to abort (Lines 15-19).

> **3: $R \to C$:** Replica returns its vote in a ST1R message.

After executing the concurrency control check, each replica returns to C a Stage1 Reply $\text{ST1R} := \langle T, vote \rangle_{\sigma_R}$. A correct replica executes this check **at most once** per transaction and stores its vote to answer future duplicate requests (§5).

> **4: $C \leftarrow R$:** The client receives replicas' votes.

C waits for ST1R messages from the replicas of each shard S touched by T. Based on these replies, C determines (*i*) whether S voted to commit or abort; and (*ii*) whether the received ST1R messages constitute a *vote certificate* (V-CERT) that proves S's vote to be *durable*. A shard's vote is durable if its original outcome can be independently retrieved and verified at any time by any correct client, independent of Byzantine failures or attempts at equivocation. If so, we dub shard S *fast*; otherwise, we call it *slow*. Votes from a slow shard do not amount to a vote certificate, but simply to a *vote tally*: to be certified, they need to be made explicitly durable during stage ST2 of the Prepare Phase. Specifically, C proceeds as follows, depending on the set of ST1R messages it receives:

(1) Commit Slow Path ($3f + 1 \le$ Commit votes $< 5f + 1$): The client has received a $CommitQuorum(CQ) := \frac{n+f+1}{2} = 3f+1$ of votes in favor of committing T. Intuitively, the size of CQ guarantees that two conflicting transactions cannot both commit, since the correct replica that is guaranteed to exist in the overlap of their *CommitQuorums* will enforce isolation. However, C receiving a CQ of Commit votes is not enough to guarantee that another client C', verifying S's vote, would see the same number of Commit votes: after all, f of the replicas in the CQ could be Byzantine, and provide a different vote if later queried by C'. C thus adds S to the set of slow shards, and records the votes it received in the following *vote tally*: $\langle id_T, S, Commit, \{\text{ST1R}\} \rangle$. Note that, though a vote tally has the same structure of a V-CERT, the information it contains is not sufficient to guarantee that S's vote is durable.

(2) Abort Slow Path ($f+1 \le$ Abort votes $< 3f+1$): A collection of $f + 1$ Abort votes constitutes the minimum *AbortQuorum (AQ)*, i.e., the minimal evidence sufficient for the client to count S's vote as Abort in the absence of a conflicting C-CERT. Requiring an *AbortQuorum* of at least $f + 1$ preserves Byzantine independence: Byzantine replicas alone cannot cause a transaction to abort as a correct replica must have found T to be conflicting with a prepared transaction. The client records the votes collected from S in the following *vote tally*: $(id_T, S, Abort, \{\text{ST1R}\}$ and adds S to the slow set for T.

(3) Commit Fast Path ($5f + 1$ Commit votes): No replica reports a conflict. Furthermore, a unanimous vote ensures that, since correct replicas never change their vote, if some client C' were to step in for C, it would be guaranteed to observe at least a CQ of $3f + 1$ Commit votes. C records the votes collected

from S in the following V-CERT: $\langle id_T, S, Commit, \{\text{ST1R}\} \rangle$ and dubs S fast.

(4) Abort Fast Path ($3f + 1 \le$ Abort votes): T conflicts with a prepared, but potentially not yet committed transaction. S's Abort vote is already durable: since a shard votes to commit only when at least $3f + 1$ of its replicas are in favor of it, once C observes $3f + 1$ replica votes for Abort from S, it is certain that S will never be able to produce $3f+1$ Commit votes, since that would require a correct replica to change its ST1R vote or equivocate. C then creates V-CERT $\langle id_T, S, Abort, \{\text{ST1R}\} \rangle$, where $\{\text{ST1R}\}$ is the set of matching ST1R replies, and adds S to the set of fast shards.

(5) Abort Fast Path (One Abort with a C-CERT for a conflicting transaction T'): C validates the integrity of the C-CERT and creates the following V-CERT for S: $\langle id_T, S, Abort, id_{T'}, \text{C-CERT} \rangle$. It indicates that S voted to abort T because T conflicts with T', which, as C-CERT proves, is a committed transaction. Since C-CERT is durable, C knows that the conflict can never be overlooked and that S's vote cannot change; thus, it adds S to the set of fast shards.

After all shards have cast their vote, C decides whether to commit (if all shards voted to commit) or abort (otherwise). Either way, it must make durable the evidence on which its decision is based. As we discussed above, the votes of fast shards already are; if there are no slow shards, then, C can move directly to the Writeback Phase (§4.3): this is Basil's fast path, which allows C to return a decision for T after a single message round trip. If some shards are in the slow set, however, C needs to take an additional step to make its tentative 2PC decision durable in a second phase (ST2).

Discussion When 2PC is layered above shards that already order transactions internally using state machine replication, then *within every shard* every correct replica has logged the vote of every other correct replica. Basil's design avoids this indiscriminate cost: if all shards are fast, then their votes are already durable without requiring replicas to run any coordination protocol; and if some shards are slow, as we discuss below, only the replicas of a single shard need to durably log the decision.

Stage 2: Making the decision durable

> **5: $C \to R$:** The client attempts to make its tentative 2PC decision durable.

C makes its decision durable by storing an (authenticated) message $\text{ST2} := \langle id_T, decision, \{\text{SHARDVOTES}\}, view = 0 \rangle$ on *one* of the shards that voted in Stage 1 of the Prepare phase; we henceforth refer to this shard, chosen deterministically depending on T's id, as S_{log}. The set $\{\text{SHARDVOTES}\}$ includes the vote tallies of all shards; the value of *view* indicates whether this ST2 message was issued by the client that initated T (*view*= 0) or it is part of a fallback protocol. We discuss *view*'s role in detail in §5.

> **6: R → C**: Replicas in S_{log} receive the ST2 message and return ST2R responses.

Each replica validates that C's 2PC decision is justified by the corresponding vote tallies; if so, the replica logs the decision and acknowledges its success. Specifically, it replies to C with a message of the form ST2R:$\langle id_T, decision, view_{decision}, view_{current} \rangle_{\sigma_R}$. We once again defer discussion of what a view is to §5.

> **7: C ← R**: The client receives a sufficient number of matching replies to confirm a decision was logged.

C waits for $n - f$ ST2R messages whose *decision* and $view_{decision}$ match, and creates a single shard certificate V-CERT$_{S_{log}}$: $\langle id_T, S, decision, \{ST2R\} \rangle$ for the logging shard.

4.3 Writeback Phase

C notifies its local application of whether T will commit or abort, and asynchronously broadcasts to all shards that participated in the Prepare phase a corresponding decision certificate (C-CERT for commit; A-CERT for abort).

> **1: C → R**: The client asynchronously forwards decision certificates to all participating shards.

C sends to all involved shards a decision certificate (C-CERT:$\langle id_T,$ *Commit*, $\{$V-CERT$_S\} \rangle$ for a Commit decision, A-CERT: $\langle id_T,$ *Abort*, $\{$V-CERT$_S\} \rangle$ otherwise). We distinguish between the fast, and slow path: On the fast path, C-CERT consists of the full set of Commit V-CERT votes from all involved shards, while an A-CERT need only contain one V-CERT vote for Abort. On the slow path, both C-CERT and A-CERT simply include V-CERT$_{S_{log}}$.

> **2: R ← C**: Replica validates C-CERT or A-CERT and updates store accordingly.

Replicas update all local data structures, including applying writes to the datastore on commit and notifying pending dependencies.

4.4 Further Optimizations

To amortize the cost of signature generation and verification, Basil batches messages. Unlike leader-based systems, Basil has no central sequencer through which to batch requests. Basil instead implements batching at the replica *after* processing messages. To amortize signature generation for replies, Basil replicas create batches of b request replies, generate a Merkle tree [70] for each batch, and sign the root hash. They then send, to each client, the signed root hash, along with its request reply and all intermediate nodes necessary to reconstruct the hash (reducing signature generation by a factor of b at the cost of $log(b)$ additional messages). To amortize signature verification, Basil uses caching. When a replica successfully verifies the root hash signature in a client message m, it caches a map between the corresponding root hash value and the signature. If the replica later receives a message m' carrying the same root hash and signature as m (indicating that m and m' refer to the same batch of replies), it can, upon verifying the correctness of the root hash, immediately declare the corresponding signature valid.

5 Transaction Recovery

For performance, Basil optimistically allows transactions to acquire dependencies on uncommitted operations. Without care, Byzantine clients could leverage this optimism to cause transactions issued by correct clients to stall indefinitely. To preserve Byzantine independence, transactions must be able to eventually commit even if they conflict with, or acquire dependencies on, stalled Byzantine transactions. To this effect, Basil enforces the following invariant: if a transaction acquires a dependency on some other transaction T, or is aborted because of a conflict with T, then a correct participant (client or replica) has enough information to successfully complete T.

Specifically, Basil clients whose transactions are blocked or aborted by a stalled transaction T try to finish T by triggering a *fallback* protocol. To this end, Basil modifies MVTSO to make visible the operations of transactions that have *prepared* only. As T's ST1 messages contain all of T's planned writes, any client or replica can use this information to take it upon itself to finish T. A correct client is guaranteed to be able to retrieve the ST1 for any of its dependencies, since $f + 1$ replicas (*i.e.*, at least one correct) must have vouched for that ST1 during T's read phase. Likewise, a correct client's transaction only aborts if at least $f+1$ replicas report a conflict.

Basil's fallback protocol starts with clients: any client blocked by a stalled transaction T can try to finish it. In the **common case**, it will succeed by simply re-executing the previously described Prepare phase; success is guaranteed as long as replicas within the shard S_{log} that logged shard votes in Stage 2 of T's Prepare phase store the same decision for T. The **divergent case**, in which they do not, can occur in one of two ways: (*i*) a Byzantine client issued T and sent deliberately conflicting ST2 messages to S_{log}; or (*ii*) multiple correct clients tried to finish T concurrently, and collected Prepare phase votes that led them to reach (and try to store at S_{log}) different decisions. Fortunately, in Basil a Byzantine client cannot generate conflicting ST2 messages at will: its ability to do so depends on the odds (which §6 suggests are low) of receiving, from the replicas of at least one shard, votes that constitute *both* a CQ and an AQ (i.e., *3f+1* Commit votes and *f+1* Abort votes). Whatever the cause, if a client trying to finish T observes that replicas in S_{log} store different decisions, it proceeds to elect a *fallback leader*, chosen deterministically among the replicas in S_{log}. Through this process, Basil guarantees that clients are always able to finish dependent transactions after at most $f + 1$ fallback leader elections (since one of them must elect a correct leader).
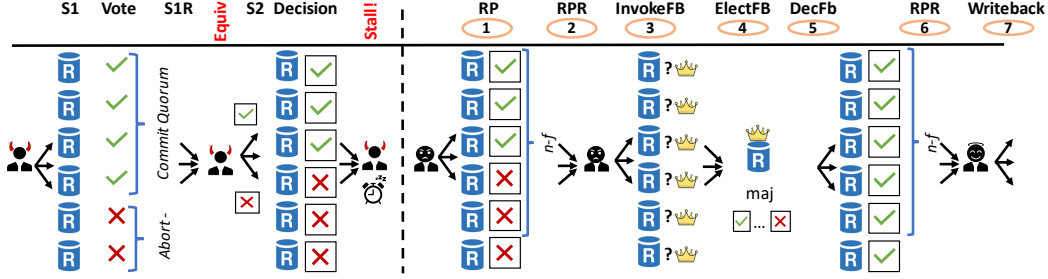
Figure 2. Fallback Scenario. A Byzantine client equivocates ST2R decisions and stalls. An interested client invokes the FB

Though Basil's fallback protocol is reminiscent of the traditional view-change protocols used to evict faulty leaders, it differs in three significant ways. First, it requires no leader in the common case; further, if electing a fallback leader becomes necessary, communication costs can be made linear in the number of replicas using threshold signatures [43], as in HotStuff [101] . Second, the fallback election is local, and affects only transactions that access the same operations as the stalled transaction: when a fallback leader is elected for T, the scope of its leadership is limited to finishing T. In contrast, a standard view-change prevents the system from processing *any* operation and the leader, once elected, lords over all consensus operations during its tenure. Finally, Basil' fallback leaders have no say on the ordering of transactions or on what they decide [105].

As in traditional view-change protocols, each leader operates in a *view*. For independent operability, views are defined on a per-transaction basis. Transactions start in *view* = 0; transactions in that view can be brought to a decision by any client. A replica increases its view number for T each time it votes to elect a new fallback leader.

We now describe the steps of the fallback protocol triggered by a client C wishing to finish a transaction T, distinguishing between the aforementioned common and divergent cases.

Common case In the common case, the client simply re-sends a ST1 message (renamed for clarity *Recovery Prepare* (RP)) in this context) to all the replicas in shards accessed by T. Replicas reply with an RPR message which, depending the progress of previous attempts (if any) at completing T corresponds to either (*i*) a ST1R message; (*ii*) a ST2R message; or (*iii*) a C-CERT or A-CERT certificate. Based on these replies, the client can fast-forward to the corresponding next step in the Prepare or Writeback protocol. In the common phase, stalled dependencies thus cause correct clients to experience only a single additional round-trip .

Divergent case If, however, the client only receives non-matching ST2R replies, more complex remedial steps are needed. ST2R can differ (*i*) in their decision value and (*ii*) in their view number $view_{decision}$. The former, as we saw, is the result of either explicit Byzantine equivocation or of multiple

clients attempting to concurrently terminate T. The latter indicates the existence of prior fallback invocations: a Byzantine fallback leader, for instance, could have intentionally left the fallback process hanging. In both scenarios, the client elects a fallback leader. The steps outlined below ensure that, once a correct fallback leader is elected, replicas can be reconciled without introducing live-lock.

> **(1: C → R):** Upon receiving non-matching ST2R responses, a client starts the fallback process.

The client sends $InvokeFB := \langle id_T, views \rangle$, where *views* is the set of signed current views associated with the RPR responses received by the client.

> **(2: R → R_{FL}):** Replicas receive fallback invocation *InvokeFB* and start election of a fallback leader R_{FL} for the current view.

R takes two steps. First, it determines the most up-to-date view v'' held by correct replicas in S_{log} and adopts it as its current view $view_{current}$. Second, R sends message ELECTFB: $\langle id_T, decision, view_{current}\rangle_{\sigma_R}$ to the replica with id $v_{current} + (id_T \mod n)$ to inform it that R now considers it to be T's fallback leader.

R determines its current view as follows: If a view v appears at least $3f + 1$ times in the *current views* received in *InvokeFB*, then R updates its $view_{current}$ to $max(v + 1, view_{current})$; otherwise, it sets its $view_{current}$ to the largest view v greater than its own that appears at least $f + 1$ times in *current views*. When counting how frequently a view is present in the received *current views*, R uses vote subsumption: the presence of view v counts as a vote also for all $v' \leq v$.

The thresholds Basil adopts to update a replica's current view are chosen to ensure that all $4f + 1$ correct replicas in S_{log} quickly catch up to the same view, and thus agree on the identity of the fallback leader. Specifically, by requiring $3f + 1$ matching views to advance to a new view v, Basil ensures that at least $2f + 1$ correct replicas are at most one view behind v at any given time. In turn, this threshold guarantees that (*i*) a correct client will receive at least $f + 1$ matching views for $v' \geq v - 1$ in response to its RP message and (*ii*) will include them in its *InvokeFB*. These $f + 1$ matching views are sufficient for all $4f + 1$ correct replicas to catch-up to view v',

then (if necessary) jointly move to view v, and send election messages to the fallback leader of v.

> **(3: $R_{FL} \rightarrow R$)**: Fallback leader R_{FL} aggregates election messages and sends decisions to replicas.

R_{FL} considers itself elected upon receiving $4f + 1$ ELECTFB messages with matching views $view_{elect}$. It proposes a new decision $dec_{new} = majority(\{decision\})$ and broadcasts message DECFB:$\langle (id_T, dec_{new}, view_{elect})_{\sigma_{R_{FL}}}, \{ELECTFB\} \rangle$, which includes the ELECTFB messages as proof of the sender's leadership. As in prior work [43, 101], Basil could use threshold signatures to aggregate ELECTFB messages into a single signature, keeping the logic linear in both message size and cryptographic costs.

> **(4: $R \rightarrow C$)**: Replicas sends a ST2R message to interested clients.

Replicas receive a DECFB message and adopt the message's decision (and $view_{decision}$) as their own if their current view is smaller or equal to $view_{elect}$. If so, replicas update their current view to $view_{elect}$ and forward the decision to all interested clients in a ST2R message: $\langle id_T, decision, view_{decision}, view_{current} \rangle_{\sigma_R}$.

> **(5: C**: A client creates a V-CERT or restarts fallback.

If the client receives $n - f$ ST2R with matching decision and decision views, she creates a V-CERT$_{S_{log}}$ and proceeds to the Commit phase. Otherwise, it restarts the fallback using the newly received $view_{current}$ messages to propose a new view.

We illustrate the full divergent case algorithm in Figure 2, which for simplicity considers a transaction T involving a single shard. To begin the process of committing T, a Byzantine client sends message ST1 and waits for all ST1R messages. Since the replies it receives allow it to generate both a Commit and Abort quorum, the client chooses to equivocate, sending ST2 messages for both Commit and Abort. It then stalls. A second correct client who acquired a dependency on T attempts to finish it. It sends RP messages (1) and receives non-matching RPR messages (three Commit and two Abort decisions, all from view 0) (2). To redress this inconsistency, the correct client invokes a Fallback with view 0 (3b). Upon receiving this message, replicas transition to view 1 and send their own decision to view 1's leader in an ELECTFB message (4). In our example, having received a majority of Commit decisions, the leader chooses to commit and broadcasts its decision to all other replicas in an DECFB message (5). Finally, replicas send the transaction's outcome to the interested client (6), who then proceeds to the Writeback phase (7).

6 Evaluation

Our evaluation seeks to answer the following questions:
- How does Basil perform on realistic applications? (§6.1)
- Where do Basil's overheads come from? (§6.2)
- What are the impacts of our optimizations in Basil? (§6.3)
- How does Basil perform under failures? (§6.4)

Baselines We compare against three baselines: (*i*) TAPIR [104], a recent distributed database that combines replication and cross-shard coordination for greater performance but does not support Byzantine faults; (*ii*) TxHotstuff, a distributed transaction layer built on top of the standard C++ implementation [100] of HotStuff, a recent consensus protocol that forms the basis of several commercial systems [15, 17, 25, 33, 71], most notably Facebook's Libra Blockchain; and (*iii*) TXBFT-SMaRt, a distributed transaction layer built on top of BFT-SMaRt [1, 22], a state-of-the-art PBFT-based implementation of Byzantine state machine replication (SMR). HotStuff and BFT-SMaRt support general-purpose SMR, and are not fully-fledged transactional systems; we thus supplement their core consensus logic with a coordination layer for sharding (running 2PC) and an execution layer that implements a standard optimistic concurrency control serializability check [53] and maintains the underlying key-value store. Additionally, we augmented both BFT baselines to also profit from Basil' reply batching scheme. This architecture follows the standard approach to designing distributed databases (e.g. Google Spanner [30], Hyperledger Fabric [92] or Callinicos [78, 79]) where concurrency control and 2PC are layered on top of the consensus mechanism. Spanner and Hyperledger (built on Paxos and Raft, respectively) are not Byzantine-tolerant, while Callinicos does not support interactive transactions. [1]

Experimental Setup We use CloudLab [2] m510 machines (8-core 2.0 GHz CPU, 64 GB RAM, 10 GB NIC, 0.15ms ping latency) and run experiments for 90 seconds (30s warm-up/cool-down). Each system tolerates $f = 1$ faults ($N = 2f+1$ for TAPIR, $3f + 1$ for HotStuff and BFT-SMaRt).

6.1 High-level Performance

We first evaluate Basil against three popular benchmark OLTP applications: TPC-C [94], Smallbank [35], and the Retwis-based transactional workload used to evaluate TAPIR [104] . TPC-C simulates the business logic of an e-commerce framework. We configure it to run with 20 warehouses. As we do not support secondary indices, we create a separate table to (*i*) locate a customer's latest order in the `order status` transaction and (*ii*) lookup customers by last name in the `order status` and `payment` transactions [32, 90]. We configure Smallbank, a simple banking application benchmark, with one million accounts. Access is skewed, with 1,000 accounts being accessed 90% of the time. Users in Retwis, which emulates a simple social network, similarly follow a moderately skewed Zipfian distribution (0.75). Figures 3a and 3b reports results for the three applications.

---

[1] We discussed extensively our setup and implementation with the authors of TAPIR and HotStuff. We corresponded with the authors of Callinicos who were unfortunately unable to locate a fully functional version of their system.
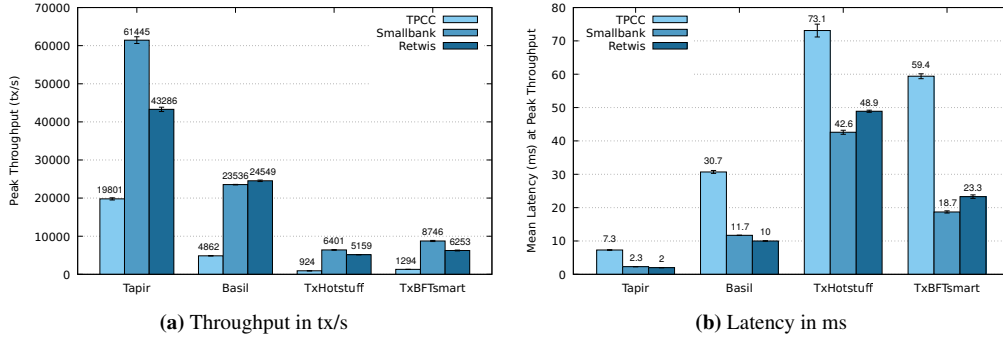
(a) Throughput in tx/s

(b) Latency in ms

Figure 3. Application High-level Performance

(a) Impact of signatures

(b) Impact of read quorums
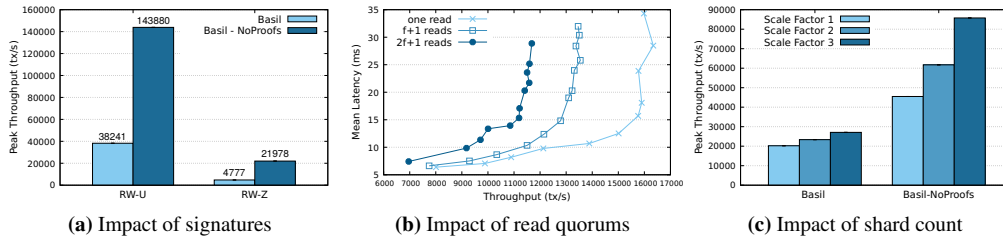
(c) Impact of shard count

Figure 4. BFT Overheads

TPC-C Basil's TPC-C throughput is 5.2x higher than Tx-Hotstuff's and 3.8x higher than TxBFT-SMaRt's – but 4.1x lower than TAPIR's. All these systems are contention-bottlenecked on the read-write conflict between `payment` and `new-order`. Basil has 4.2x higher latency than TAPIR: this increases the conflict window of contending transactions, and thus the probability of aborts. Basil's higher latency stems from (*i*) its replicas need for signing and verifying signatures; (*ii*) its larger quorum sizes for both read and prepare phases; and (*iii*) its need to validate read/prepare replies at clients.

Throughput in Basil is higher than in TxHotStuff and TxBFT-SMaRt. Basil's superior performance is directly linked to its lower latency (2.4x lower than TxHotstuff, 1.2x lower than TxBFT-SMaRt). By merging 2PC with agreement, Basil allows transactions to decide in a single round-trip 96% of the time (through its fast path to commit or abort. TxHotstuff and TxBFT-SMaRt, which layer a 2PC protocol over a black-box consensus instance, must instead process and order two requests for each decision (one to Prepare, and one to Commit/Abort), each requiring multiple roundtrips. In particular, Hotstuff and BFT-SMaRT incur respectively nine and five message delays before returning the Prepare result to clients. In a contention-heavy application like TPC-C, this higher latency translates directly into lower throughput, since it significantly increases the chances that transactions will conflict. Indeed, for these applications layering transaction processing on top of state machine replication actually turns a classic performance booster for state machine replication—running agreement on large batches—into a liability, as large batches increase latency and encourage clients to operate in lock-step, increasing contention artificially. In practice, we find that TxHotstuff and TxBFT-SMaRt perform best with comparatively small batches (four transactions for TxHotStuff, 16 for TxBFT-SMaRT).

Smallbank and Retwis Basil is only 1.8/2.6x slower than TAPIR for these workloads, which are resource bottlenecked for both systems. The lower contention in Smallbank and Retwis (due to the relatively small transactions) allows Basil to use a batch size of 16 for signature generation (up from 4 in TPC-C), thus lowering the cryptographic overhead that Basil pays over TAPIR. With this larger batch, both TAPIR and Basil are bottlenecked on message serialization/deserialization and networking overheads. Because of their higher latency, however, TxHotStuff and TxBFT-SMaRt continue to be contention bottlenecked: Basil's commit rates for Smallbank and Retwis are respectively 93% and 98%, but for TxHotStuff they drop to 75% and 85% and for TxBFT-SMaRt to 85% for both benchmarks. Even on their best configuration (batch size of 16 for TxHotStuff and 64 for TxBFT-SMaRt), Basil outperforms them, respectively, by 3.7x and 2.7x on Smallbank, and by 4.8x and 3.9x on Retwis.

6.2 BFT Overheads

Besides additional replicas (from $2f + 1$ to $5f + 1$), tolerance to Byzantine faults requires both additional cryptography to preserve Byz-serializability and more expensive reads to preserve Byzantine independence. To evaluate these overheads,

we configure the YCSB-T microbenchmark suite [29] to implement a simple workload of identical transactions over ten million keys. We distinguish between a uniform workload (*RW-U*) and a Zipfian workload (*RW-Z*) with coefficient 0.9.
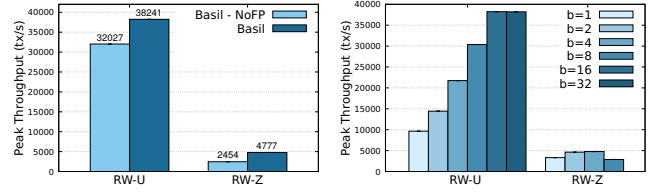
We first quantify the cost of cryptography. To do so, we measure the relative throughput of Basil with and without signatures. Transactions consist of two reads and two writes. Figure 4a describes our results. We find that Basil without cryptography performs 3.7x better than Basil with cryptography on the uniform workload, and up to 4.6x better on the skewed workload. Without cryptography, Basil can use cores that would have been dedicated for signing/ signature verification for regular operation processing. This effect is more pronounced on the skewed workload as reducing latency (through increased operation parallelism, lack of batching, and absence of signing/verification latency) reduces contention, and thus further increases throughput.

In all sharded BFT systems, the number of signatures necessary per transaction grows linearly with the number of shards: each replica must verify that other shards also voted to commit/abort a transaction before finalizing the transaction decision locally. This requires a signature per shard. In Figure 4c, we quantify this cost by increasing the number of shards from one to three on the CPU-bottlenecked *RW-U* workload (three reads/writes). Basil without cryptography increases by a factor of 1.9 (on average, transactions with three read operations will touch two distinct shards). In contrast, Basil's throughput increases by only 1.3x.

To guarantee Byzantine independence, individual clients must receive responses from $f + 1$ replicas instead of a single replica. Reading from $2f + 1$ replicas (thus sending to 3f+1) increases the chances of a transaction acquiring a valid dependency over reading outdated data. We measure the relative cost of these different read quorum sizes in Figure 4b. We use a simple read-only workload of 24 operations per transaction, and a batch size of 16. Unsurprisingly, increasing the number of read operations increases the load on each replica due to the (*i*) additional signature generations that must be performed, and (*ii*) the additional messages that must be processed. Throughput decreases by 20% when reading from $f + 1$ replicas (instead of one), and a further 16% when reading from $2f + 1$.

6.3 Basil Optimizations

We measure how Basil's performance benefits from batching and from its fast path option. We report results for YCBS-T with and without fast path (NoFP) on a workload of two reads and two writes (Figure 5a). For the uniform workload, enabling fast paths leads to a 19% performance increase; the ST2R messages that fast paths save contain a signature that must be verified, but require little additional processing. For a contended Zipfian workload, however, the additional phase incurred by the slow path increases contention (as it increases latency): adding the fast path increases throughput by 49%.

(a) Throughput with/without fast path   **(b)** Throughput vs. batch size

Figure 5. Basil Optimizations

Note that Byzantine replicas, by refusing to vote or voting abort, can effectively disable the fast path option; Basil can prevent this by removing consistently uncooperative replicas.

Next, we quantify the effects of batching. We report the throughput for both workloads (transactions consist of two reads and two writes) while changing the batch size from 1 to 32 ((Figure 5b). As expected, on the resource-bottlenecked uniform workload, throughput increases linearly with increased batch size until peaking at 16 (a 4x throughput increase) – at which point additional hashing costs of the batching Merkle tree nullify any further reduction in signature costs. On the Zipfian workload instead, throughput only increases by up 1.4x, peaking at a small batch size of 4, and degrading afterwards as higher wait times and batch-induced client lock step increase contention (thus reducing throughput).

6.4 Basil Under Failures

Basil can experience Byzantine failures from both replicas and clients. We have already quantified the effect of Byzantine replicas preventing fast paths (Figure 5a) and, by being unresponsive, forcing $(2f + 1)$-sized read quorums (Figure 4b). We then focus here on quantifying the effects of clients failing.

Byzantine clients can stall after sending ST1 messages (*stall-early*), or before sending vote certificates V-CERT$_S$ (*stall-late*). To equivocate, they must instead receive votes that allow them to generate, and send to replicas, conflicting V-CERT certificates. We evaluate two scenarios: a worst-case, in which we artificially allow clients to always equivocate (*equiv-forced*), and a realistic setup where clients only equivocate when the set of messages received allows them to (*equiv-real*). For both scenarios, we report the throughput of correct clients (measured in $tx/s/client_{correct}$); we keep a constant number of clients, a fraction of which exhibits Byzantine behavior (every n^{th} transaction - we refer to those transactions as faulty). Figures 6a and 6b report our results.

For the RW-U workload, the additional CPU cost of fallback invocations on the CPU-bottlenecked servers causes correct clients' throughput to decrease slowly and linearly. Clients invoke fallbacks only rarely, as there is no contention. Moreover, stalled transactions can be finished in a single round-trip (a pair of RP, RPR) messages thanks to the fallback's common case and fast path. The small throughput drop over *stall-late* is an artefact of Byzantine clients directly starting a new transaction before finishing the old one, increasing the throughput
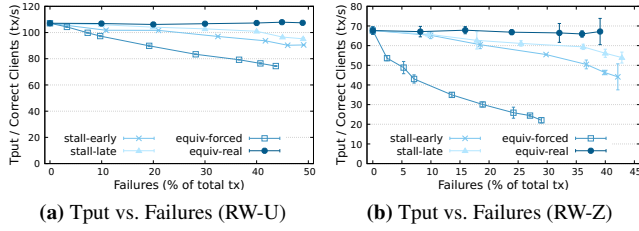
(a) Tput vs. Failures (RW-U) **(b)** Tput vs. Failures (RW-Z)

Figure 6. Basil performance under Client Failures

of malicious clients over correct ones. The cost of forced equivocation is higher as it requires three rounds of message processing (fallback invocation, election, and decision adoption). In reality, *equiv-real* sees no throughput drop, as the lack of contention makes equivocation impossible: Byzantine clients cannot build the necessary conflicting V-CERT's.

The RW-Z workload is instead contention-bottlenecked: higher latency implies more conflicts, and thus lower throughput. The impact of *stall-late* stalls remains small, as all affected clients still recover the transaction on the common case fast path (incurring only one extra roundtrip). The performance degradation is slightly higher in *stall-early*, as stalled-early transactions do not finish the transactions on which they depend before stalling. Instead, affected correct clients must themselves invoke the fallback for stalled dependent transactions, which inceases latency. In practice, dependency chains remain small: because of the Zipfian nature of the workload, correct clients quickly notice stalled transactions and aggressively finish them. We note that stalled transactions do not themselves increase contention: Basil allows the stalled writes of prepared but uncommitted transactions to become visible to other clients as dependencies. A stalled transaction thus causes dependency chains to grow, but does not increase the conflict window. The throughput drop that results from forcing equivocation failures is, in contrast, significant: equivocation requires three round-trip to resolve and may lead to transactions aborting and to cascading aborts in dependency chains. In practice, Byzantine clients in *equiv-real* are once again rarely successful in obtaining the conflicting ST1R messages necessary to equivocate, even in a contended workload (0.048% of the time for 40 % faulty transactions) as 99% of transactions commit or abort on the fast path.

7 Related Work

State machine replication (SMR) [88] maintains a total order of requests across replicas, both in the crash failure model [26, 47, 54–57, 59, 62, 64, 76, 77, 95]) and in the Byzantine setting [16, 22–24, 27, 28, 36, 42, 43, 50, 51, 58, 63, 67, 69, 84, 99, 101], where they have been used as a main building block of Blockchain systems [8, 13, 14, 17, 23, 41, 49, 92]. To maintain a total order abstraction, existing systems process all operations sequentially (for both agreement and execution), thus limiting scalability for commutative workloads. They

are, in addition, primarily leader-based which introduces additional scalability bottlenecks [73, 89, 104] as well as fairness concerns. Rotating leaders [23, 28, 101] reduce fairness concerns, and multiple-leader based systems [16, 60, 73, 89] increase throughput. Recent work [46, 48, 105] discusses how to improve fairness in BFT leader-based systems with supplementary ordering layers and censorship resilience. Basil sidesteps these concerns by adopting a leaderless approach and addresses the effects of Byzantine actors beyond ordering through the stronger notion of Byzantine Independence.

Fine-grained ordering Existing replicated systems in the crash-failure model leverage operation semantics to allow commutative operations to execute concurrently [52, 56, 61, 73–75, 81, 91, 98, 104]. This work is much rarer in the BFT context, with Bilbos [18] and Zzyzx [45] being the only BFT protocol that seek to leverage commutativity. However, unlike Basil, Bilbos is limited to a static transaction model and introduces blocking between all *potentially* concurrent transactions; while Zzyzx resorts to a SMR substrate protocol under contention. Other existing Quorum-based systems naturally allow for non-conflicting operations to execute concurrently, but do not provide transactions [10, 31, 65, 68].

Sharding Some Blockchains rely on sharding to parallelize independent transactions, but continue to rely on a total-order primitive within shards [13, 49, 102]. As others in the crash-failure model have highlighted [75, 103, 104], this approach incurs redundant coordination and fails to fully leverage the available parallelism within a workload.

DAGs Other permissionless Blockchains use directed acyclic graphs rather than chains [83, 85, 87], but require dependencies and conflicts to be known prior to execution.

Byzantine Databases Basil argues that BFT systems and Blockchains are in fact simply databases and draws on prior work in BFT databases. HRDB [96] offers interactive transactions for a replicated database, but relies on a trusted coordination layer. Byzantium [40] designs a middleware system that utilizes PBFT as atomic broadcast (AB) and provides Snapshot Isolation using a primary backup validation scheme. Augustus [79] leverages sharding for scalability in the mini-transaction model [12] and relies on AB to implement an optimistic locking based execution model. Callinicos [78] extends Augustus to support armored-transactions in a multi-round AB protocol that re-orders conflicts for robustness against contention. BFT-DUR [82] builds interactive transactions atop AB, but does not allow for sharding. Basil instead supports general transactions and sharding without a leader or the redundant coordination introduced by atomic broadcast.

Byzantine Clients Basil, being client-driven, must defend against Byzantine clients. It draws from prior work targeted at reducing the severity and frequency of client misbehavior [40, 65, 66, 78, 79, 82] and extends Liskov and Rodrigues' [65] definition of Byz-Linearizability to formalize the first safety and liveness properties for transactional BFT systems.

8 Conclusion

Basil shows BFT systems can offer the *abstraction* of a totally-ordered ledger while supporting highly concurrent transaction processing and ensuring *Byz-serializability*. Basil clients make progress *independently*, while *Byzantine Independence* limits the influence of Byzantine participants.

References

[1] Byzantine fault-tolerant (bft) state machine replication (smart) v1.2. https://github.com/bft-smart/library.

[2] CloudLab. https://www.cloudlab.us.

[3] Retwis benchmark. http://retwis.redis.io/.

[4] Apple supplier list. https://www.apple.com/supplier-responsibility/pdf/Apple-Supplier-List.pdf, 2019.

[5] State Farm and USAA Work Together to Test a Blockchain Solution. https://newsroom.statefarm.com/blockchain-solution-test-for-subrogation/, 2019.

[6] Calibra: A simple global payment system and financial infrastructure that empowers billions of people. https://libra.org/en-US/, 2020.

[7] IBM food trust. A new era for the world's food supply. https://www.ibm.com/blockchain/solutions/food-trust, 2020.

[8] JP Morgan Quorum. https://www.goquorum.com/, 2020.

[9] Transform cross-border payments with IBM blockchain world wire. https://www.ibm.com/blockchain/solutions/world-wire, 2020.

[10] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie. Fault-scalable Byzantine fault-tolerant services. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 59–74, 2005.

[11] A. Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. PhD thesis, 1999.

[12] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 159–174, 2007.

[13] M. Al-Bassam, A. Sonnino, S. Bano, D. Hrycyszyn, and G. Danezis. Chainspace: A sharded smart contracts platform. arXiv:1708.03778, 2017.

[14] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, et al. Hyperledger Fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, pages 1–15, 2018.

[15] J. Ansel and M. Olszewski. Bftree - scaling hotstuff to millions of validators. https://storage.googleapis.com/celo_whitepapers/BFTree%20-%20Scaling%20HotStuff%20to%20Millions%20of%20Validators.pdf, 2019.

[16] B. Arun, S. Peluso, and B. Ravindran. ezbft: Decentralizing Byzantine fault-tolerant state machine replication. In *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 565–577, 2019.

[17] M. Baudet, A. Ching, A. Chursin, G. Danezis, F. Garillot, Z. Li, D. Malkhi, O. Naor, D. Perelman, and A. Sonnino. State machine replication in the libra blockchain. *The Libra Association Technical Report*, 2019.

[18] R. Bazzi and M. Herlihy. Clairvoyant state machine replications. In *International Symposium on Stabilizing, Safety, and Security of Distributed Systems*, pages 254–268. Springer, 2018.

[19] P. A. Bernstein and N. Goodman. Multiversion concurrency control-theory and algorithms. *ACM Transactions on Database Systems (TODS)*, 8(4):465–483, 1983.

[20] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*, volume 370. Addison-wesley New York, 1987.

[21] P. A. Bernstein, D. W. Shipman, and W. S. Wong. Formal aspects of serializability in database concurrency control. *IEEE Transactions on Software Engineering*, 5(3):203–216, May 1979.

[22] A. Bessani, J. Sousa, and E. E. Alchieri. State machine replication for the masses with BFT-SMaRt. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 355–362, 2014.

[23] E. Buchman. *Tendermint: Byzantine fault tolerance in the age of blockchains*. PhD thesis, 2016.

[24] M. Castro, B. Liskov, et al. Practical Byzantine fault tolerance. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 173–186, 1999.

[25] T.-H. H. Chan, R. Pass, and E. Shi. Pala: A simple partially synchronous blockchain. *IACR Cryptol. ePrint Arch.*, 2018:981, 2018.

[26] T. Chandra, R. Griesmer, and J. Redstone. Paxos made live – an engineering perspective. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 398–407, 2007.

[27] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche. Upright cluster services. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 277–290, 2009.

[28] A. Clement, E. Wong, L. Alvisi, M. Dahlin, and M. Marchetti. Making Byzantine fault tolerant systems tolerate Byzantine faults. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, page 153–168, 2009.

[29] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC)*, pages 143–154, 2010.

[30] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '12, pages 251–264.

[31] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. HQ replication: A hybrid quorum protocol for Byzantine fault tolerance. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 177–190, 2006.

[32] N. Crooks, M. Burke, E. Cecchetti, S. Harel, R. Agarwal, and L. Alvisi. Obladi: Oblivious serializable transactions in the cloud. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 727–743, Carlsbad, CA, Oct. 2018. USENIX Association.

[33] Cypherium.io. Cypherium-whitepaper-2-0. https://www.cypherium.io/whitepaper/cypherium-whitepaper-2-0/.

[34] Deloitte. Using blockchain & internet-of-things in supply chain traceability. https://www2.deloitte.com/content/dam/Deloitte/lu/Documents/technology/lu-blockchain-internet-things-supply-chain-traceability.pdf, 2017.

[35] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux. OLTP-Bench: An extensible testbed for benchmarking relational databases. In *Proceedings of the VLDB Endowment (PVLDB)*, 2013.

[36] S. Duan, S. Peisert, and K. N. Levitt. hBFT: speculative Byzantine fault tolerance with minimum cost. *IEEE Transactions on Dependable and Secure Computing*, 12(1):58–70, 2014.

[37] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

[38] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[39] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[40] R. Garcia, R. Rodrigues, and N. Preguiça. Efficient middleware for Byzantine fault tolerant database replication. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, pages 107–122, 2011.

[41] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich. Algorand: Scaling Byzantine agreements for cryptocurrencies. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 51–68, 2017.

[42] R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić. The next 700 BFT protocols. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, pages 363–376, 2010.

[43] G. G. Gueta, I. Abraham, S. Grossman, D. Malkhi, B. Pinkas, M. K. Reiter, D.-A. Seredinschi, O. Tamir, and A. Tomescu. SBFT: A scalable decentralized trust infrastructure for blockchains. *arXiv:1804.01626*, 2018.

[44] A. Haeberlen, P. Aditya, R. Rodrigues, and P. Druschel. Accountable virtual machines. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, page 119–134, 2010.

[45] J. Hendricks, S. Sinnamohideen, G. R. Ganger, and M. K. Reiter. Zzyzx: Scalable fault tolerance through Byzantine locking. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 363–372, 2010.

[46] M. Herlihy and M. Moir. Enhancing accountability and trust in distributed ledgers. *arXiv:1606.07490*, 2016.

[47] F. P. Junqueira, B. C. Reed, and M. Serafini. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 245–256, 2011.

[48] M. Kelkar, F. Zhang, S. Goldfeder, and A. Juels. Order-fairness for Byzantine consensus. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 2020.

[49] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pages 583–598, 2018.

[50] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: speculative Byzantine fault tolerance. *ACM SIGOPS Operating Systems Review*, 41(6):45–58, 2007.

[51] R. Kotla and M. Dahlin. High throughput Byzantine fault tolerance. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 575–584, 2004.

[52] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete. MDCC: Multi-data center consistency. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, pages 113–126, 2013.

[53] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)*, 6(2):213–226, 1981.

[54] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[55] L. Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column)*, 32(4):51–58, 2001.

[56] L. Lamport. Fast Paxos. Technical Report Microsoft Research Technical Report MSR-TR-2005-112, 2005.

[57] L. Lamport. Generalized consensus and paxos. Technical Report Microsoft Research Technical Report MSR-TR-2005-33, 2005.

[58] L. Lamport. Byzantizing paxos by refinement. In *Proceedings of the International Symposium on Distributed Computing (DISC)*, pages 211–224, 2011.

[59] B. Lampson. The ABCD's of paxos. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, volume 1, page 13, 2001.

[60] B. Li, W. Xu, M. Z. Abid, T. Distler, and R. Kapitza. Sarek: Optimistic parallel ordering in Byzantine fault tolerance. In *Proceedings of the European Dependable Computing Conference (EDCC)*, pages 77–88, 2016.

[61] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 265–278, 2012.

[62] H. C. Li, A. Clement, A. S. Aiyer, and L. Alvisi. The paxos register. In *Proceedings of the IEEE International Symposium on Reliable Distributed Systems (SRDS)*, pages 114–126, 2007.

[63] B. Liskov. From viewstamped replication to Byzantine fault tolerance. In *Replication*, pages 121–149. Springer, 2010.

[64] B. Liskov and J. Cowling. Viewstamped replication revisited. Technical Report MIT-CSAIL-TR-2012-021, MIT, July 2012.

[65] B. Liskov and R. Rodrigues. Tolerating Byzantine faulty clients in a quorum system. In *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 34–34, 2006.

[66] A. F. Luiz, L. C. Lung, and M. Correia. Byzantine fault-tolerant transaction processing for replicated databases. In *Proceedings of the IEEE International Symposium on Network Computing and Applications (NCA)*, pages 83–90, 2011.

[67] D. Malkhi, K. Nayak, and L. Ren. Flexible Byzantine fault tolerance. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 1041–1053, 2019.

[68] D. Malkhi and M. Reiter. Byzantine quorum systems. In *Distributed computing*, volume 11, pages 203–213. Springer, 1998.

[69] J.-P. Martin and L. Alvisi. Fast byzantine consensus. *IEEE Transactions on Dependable and Secure Computing*, 3(3):202–215, 2006.

[70] R. C. Merkle. A digital signature based on a conventional encryption function. In *Proceedings of the International Conference on the Theory and Application of Cryptographic Techniques (EUROCRYPT)*, pages 369–378. Springer, 1987.

[71] meter.io. Supercharge ethereum for the financial internet. https://www.meter.io/.

[72] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song. The honey badger of BFT protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 31–42, 2016.

[73] I. Moraru, D. G. Andersen, and M. Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 358–372, 2013.

[74] S. Mu, Y. Cui, Y. Zhang, W. Lloyd, and J. Li. Extracting more concurrency from distributed transactions. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 479–494, 2014.

[75] S. Mu, L. Nelson, W. Lloyd, and J. Li. Consolidating concurrency control and consensus for commits under conflicts. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 517–532, 2016.

[76] B. M. Oki and B. H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 8–17, 1988.

[77] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 305–319, 2014.

[78] R. Padilha, E. Fynn, R. Soulé, and F. Pedone. Callinicos: Robust transactional storage for distributed data structures. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 223–235, 2016.

[79] R. Padilha and F. Pedone. Augustus: Scalable and robust storage for cloud applications. In *Proceedings of the ACM European Conference*

on Computer Systems (EuroSys), pages 99–112, 2013.

[80] C. H. Papadimitriou. The Serializability of Concurrent Database Updates. *Journal of the ACM (JACM)*, 26(4):631–653, 1979.

[81] S. J. Park and J. Ousterhout. Exploiting commutativity for practical fast replication. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 47–64, 2019.

[82] F. Pedone and N. Schiper. Byzantine fault-tolerant deferred update replication. *Journal of the Brazilian Computer Society*, 18(1):3–18, 2012.

[83] H. Pervez, M. Muneeb, M. U. Irfan, and I. U. Haq. A comparative analysis of dag-based blockchain architectures. In *Proceedings of the IEEE International Conference on Open Source Systems and Technologies (ICOSST)*, pages 27–34, 2018.

[84] M. Pires, S. Ravi, and R. Rodrigues. Generalized paxos made Byzantine (and less complex). *Algorithms*, 11(9):141, 2018.

[85] S. Popov and Q. Lu. IOTA: feeless and free. *IEEE Blockchain Technical Briefs*, 2019.

[86] D. P. Reed. Implementing atomic actions on decentralized data. *ACM Transactions on Computer Systems (TOCS)*, 1(1):3–23, 1983.

[87] T. Rocket, M. Yin, K. Sekniqi, R. van Renesse, and E. G. Sirer. Scalable and probabilistic leaderless BFT consensus through metastability. *arXiv:1906.08936*, 2019.

[88] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.

[89] C. Stathakopoulou, T. David, and M. Vukolić. Mir-BFT: High-throughput BFT for blockchains. *arXiv:1906.05552*, 2019.

[90] C. Su, N. Crooks, C. Ding, L. Alvisi, and C. Xie. Bringing modular concurrency control to the next level. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 283–297, 2017.

[91] P. Sutra and M. Shapiro. Fast genuine generalized consensus. In *Proceedings of the IEEE International Symposium on Reliable Distributed Systems (SRDS)*, pages 255–264, 2011.

[92] The Linux Foundation. An introduction to Hyperledger. https://www.hyperledger.org/wp-content/uploads/2018/07/HL_Whitepaper_IntroductiontoHyperledger.pdf, 2018.

[93] The Linux Foundation. Change healthcare using Hyperledger Fabric to improve claims lifecycle throughput and transparency, 2019.

[94] Transaction Processing Performance Council. The TPC-C home page. http://www.tpc.org/tpcc.

[95] R. Van Renesse, N. Schiper, and F. B. Schneider. Vive la différence: Paxos vs. viewstamped replication vs. zab. *IEEE Transactions on Dependable and Secure Computing*, 12(4):472–484, 2014.

[96] B. Vandiver, H. Balakrishnan, B. Liskov, and S. Madden. Tolerating Byzantine faults in transaction processing systems using commit barrier scheduling. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 59–72, 2007.

[97] C. Xie, C. Su, C. Littley, L. Alvisi, M. Kapritsos, and Y. Wang. High-performance ACID via modular concurrency control. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[98] X. Yan, L. Yang, H. Zhang, X. C. Lin, B. Wong, K. Salem, and T. Brecht. Carousel: Low-latency transaction processing for globally-distributed data. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 231–243, 2018.

[99] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for Byzantine fault tolerant services. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 253–267, 2003.

[100] M. Yin. libhotstuff. https://github.com/hot-stuff/libhotstuff.

[101] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham. HotStuff: BFT consensus with linearity and responsiveness. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 347–356, 2019.

[102] M. Zamani, M. Movahedi, and M. Raykova. Rapidchain: Scaling blockchain via full. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 931–948, 2018.

[103] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. Ports. When is operation ordering required in replicated transactional storage? *IEEE Data Engineering Bulletin*, 39(1):27–38, 2016.

[104] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. K. Ports. Building consistent transactions with inconsistent replication. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 263–278, 2015.

[105] Y. Zhang, S. Setty, Q. Chen, L. Zhou, and L. Alvisi. Byzantine ordered consensus without Byzantine oligarchy. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 633–649, 2020.

A Correctness Sketch

We sketch the main theorems and lemmas necessary to show safety and liveness for Basil. Full proofs can be found in Appendix B We proceed in two steps: we first prove safety without the fallback protocol. We then extend our correctness argument to handle fallback cases.

First, we prove that each replica generates a locally serializable schedule.

Lemma 1. *On each correct replica, the set of transactions for which the MVTSO-Check returns Commit forms an acyclic serialization graph.*

We then show that decisions for transactions are unique.

Lemma 2. *There cannot exist both an C-CERT and a A-CERT for a given transaction.*

Next, we define a notion of conflicting transactions: a transaction T_i conflicts with T_j if adding T_j to a history containing T_i would cause the execution to violate serializability.

Lemma 3. *If T_i has issued a C-CERT and T_j conflicts with T_i, then T_j cannot issue a C-CERT.*

Given these three lemmas, we prove that Basil satisfies Byz-Serializability.

Theorem 1. *Basil maintains Byzantine-serializability.*

Finally, we show that Basil preserves Byzantine independence under non-adversarial network assumptions.

Theorem 2. *Basil maintains Byzantine independence in the absence of network adversary.*

We now explicitly consider the fallback protocol. We first show that certified decisions remain durable.

Lemma 4. *Fallback leaders cannot propose decisions that contradict existing slow path C-CERT/A-CERT.*

Additionally, we show that any reconciled decision could have been proposed by a client.

Lemma 5. *Any decision proposed by a fallback leader was proposed by a client.*

Given these two Lemmas we conclude:

Theorem 3. *Invoking the fallback mechanism preserves Theorem 1 and Theorem 2.*

Next we show that, given partial synchrony [37] and the existence of a global stabilization time (GST), Basil's fallback mechanism guarantees progress for correct clients.

Theorem 4 A correct client can reconcile correct replicas' views in at most two round-trips and one time-out.

We use this result to prove that:

Lemma 6. *After GST, and in the presence of correct interested clients, a correct fallback leader is eventually elected.*

Given this lemma, we show that Basil allows correct clients to complete their dependencies during synchronous periods.

Theorem 5. *A correct client eventually succeeds in acquiring either a C-CERT or A-CERT for any transaction of interest.*

B Proofs

B.1 Byzantine-Serializability

Lemma 1. *On each correct replica, the set of transactions for which the MVTSO-Check returns Commit forms an acyclic serialization graph.*

Proof. We use Adya's formalism here [11]. An execution of Basil produces a direct serialization graph whose vertices are committed transactions and whose edges are one of three types:

- $T_i \xrightarrow{ww} T_j$ if T_i writes the version of object x that precedes T_j in the version order.
- $T_i \xrightarrow{wr} T_j$ if T_i writes the version of object x that T_j reads.
- $T_i \xrightarrow{rw} T_j$ if T_i reads the version of object x that precedes T_j's write.

We assume, as does Adya, that if an edge exists between T_i and T_j, then $T_i \neq T_j$.

First, we prove that if there exists an edge $T_i \xrightarrow{rw/wr/ww} T_j$, then $i < j$. We consider each case individually.

$\xrightarrow{ww}$ ***case.*** Assume that there is a $T_i \xrightarrow{ww} T_j$ edge. This means that T_i writes a version of object x that precedes T_j in the version order. MVTSO's version order for each object is equivalent to the timestamp order of the transactions that write to the object. Note that the order in which versions are added to the list may not match the timestamp order, but the versions are added to the appropriate indices in the list such that the timestamps of all preceding versions are smaller and the timestamps of all subsequent versions are larger than the new version's timestamp. This implies that $i < j$.

$\xrightarrow{wr}$ ***case.*** Assume that there is a $T_i \xrightarrow{wr} T_j$ edge. This means that T_j reads a version of object x written by T_i. MVTSO's algorithm returns, for a read in T_j, the latest version of an object whose version timestamp is lower than j. This implies that $i < j$.

$\xrightarrow{rw}$ ***case.*** This case is the most complex. We prove it by contradiction. Assume that there is a $T_i \xrightarrow{rw} T_j$ edge such that this edge is the first edge where $i \geq j$. $T_i \xrightarrow{rw} T_j$ means that T_i reads the version of object x that precedes T_j's write. Let that version be x_k. By the assumption that $T_i \neq T_j$, $i > j$. By the definition of the direct serialization graph, $T_k \xrightarrow{wr} T_i$ and $T_k \xrightarrow{ww} T_j$. The previous cases imply that $k < i$ and $k < j$.

Consider a replica that ran the MVTSO-Check for both T_i and T_j. There are two subcases: either the check for T_i was executed before the check for T_j or vice versa.

1. T_i **before** T_j. T_i must have committed because it exists in the DSG. This implies that T_i was in the *Prepared* set when the check was executed for T_j (Line 14 of Algorithm 1. When the check for T_j reached Line 10 of Algorithm 1 for T_j's write of x_j, the condition was satisfied by T_i's read of x_k because $k < j$ and $j < i$. Therefore, the check for

T_j returned Vote-Abort. However, this is a contradiction because T_j committed.

2. T_j **before** T_i. T_j must have committed because it exists in the DSG. This implies that T_j was in the *Prepared* set when the check was executed for T_i. When the check for T_i reached Line 7 of Algorithm 1 for T_i's read of x_k, the condition was satisfied by T_j's write of x_j because $k < j$ and $j < i$. Therefore, the check for T_i returned Vote-Abort. However, this is a contradiction because T_i is committed.

In all cases, the preliminary assumption leads to a contradiction. This implies that $i < j$.

Next, we use the fact that if there exists an edge $T_i \xrightarrow{rw/wr/ww} T_j$, then $i < j$ to prove that the set of transactions for which MVTSO-Check returns Commit is serializable.

Acyclicity. The set of transactions is serializable if the DSG has no cycles. Assume for a contradiction that there exists a cycle consisting of n transactions $T_{ts_1}, ..., T_{ts_n}$. By the previous fact, this implies that $ts_1 < ... < ts_n < ts_1$. However, transaction timestamps are totally ordered. This is a contradiction. Thus, the DSG has no cycles.

$\square$

Lemma 2. *There cannot exist both an C-CERT and a A-CERT for a given transaction.*

Proof. There are two ways that the client can form a C-CERT/A-CERT: through the single phase fast-path, and through the two-phase slow path. We consider both in turn. Recall, that on the fast path a C-CERT contains a list of ST1R as evidence for each shard, while an A-CERT contains a list of ST1R for a single shard (that voted to abort). On the slow path instead, both a C-CERT and an A-CERT contain only a list of ST2R only for the logging shard S_{log}. The intuition for this difference is that on the fast path the two-phase commit decision is yet to be validated, while on the slow path, the two-phase commit decision was already taken and logged in an effort to reduce redundant logging.

In the following, we show that for all possible combinations of certificates, no C-CERT and A-CERT can co-exist.

Commit Fast Path, Abort Fast Path Assume that a client generates both a C-CERT and an A-CERT for T and that both went fast path. A fast C-CERT requires $5f+1$ replicas (commit fast path quorum) to vote commit T on *every* shard, while a fast A-CERT requires either 1 vote (abort fast path quorum, case (1)) if a C-CERT is present to prove the conflict, or $3f + 1$ votes (abort fast path quorum, case(2)) on a *single* shard otherwise. We distinguish abort fast path quorum cases 1 and 2: (*i*): If a C-CERT exist for a conflicting transaction, then by definition of a C-CERT, at least $3f + 1$ (commit slow path quorum) replicas of the shard in question must have voted to commit a transaction T' that conflicts with T. Since correct replicas never change their vote, by Lemma 1, at least $2f + 1$ correct replicas must vote to abort T. Yet, this

scenario would require that on this shard, at least one correct replica equivocated ($2f + 1$ correct and $3f + 1$ must overlap in at least one correct replica). We have a contradiction. (*ii*): Correct replicas never change their vote, yet this scenario would require that on one shard, at least one correct replica equivocated ($5f + 1$ and $3f + 1$ must overlap in at least one correct replica). We have a contradiction.

Abort Fast Path, Commit Slow Path Assume that a client generates both a C-CERT that went slow path and an A-CERT that went fast path for T. A slow C-CERT requires $n - f$ matching ST2R replies from the logging shard. In order for a correct replica to send a commit ST2R message, it must have received a vote tally of at least $3f + 1$ commit votes created on *every* shard. Instead, a fast A-CERT requires either 1 vote (abort fast path quorum, case (1)) if a C-CERT is present to prove the conflict, or $3f + 1$ votes (abort fast path quorum, case(2)) on a *single* shard otherwise. We distinguish abort fast path quorum cases 1 and 2: (*i*): If a C-CERT exist for a conflicting transaction, then by definition of a C-CERT, at least $3f + 1$ (commit slow path quorum) replicas of the shard in question must have voted to commit a transaction T' that conflicts with T. Since correct replicas never change their vote, by Lemma 1, at least $2f + 1$ correct replicas must vote to abort T. Yet, this scenario would require that on this shard, at least one correct replica equivocated ($2f + 1$ correct and $3f + 1$ must overlap in at least one correct replica). We have a contradiction. (*ii*): Correct replicas never change their vote, yet this scenario would require that on one shard, at least one correct replica equivocated ($3f + 1$ and $3f + 1$ must overlap in at least one correct replica). We have a contradiction.

Commit Fast-Path, Abort Slow-Path. Assume that a client generates a C-CERT that went fast path, and a A-CERT that went slow path for a transaction T. A fast C-CERT requires $5f+1$ replicas (commit fast path quorum) to vote to commit T on *every shard*, while a slow A-CERT requires $n - f = 4f + 1$ ST2R messages with decision to abort. In order for a correct replica to send an abort ST2R message, it must have received a vote tally of at least $f + 1$ abort votes created on a single shard. Correct replicas never change their vote, yet this scenario would require that on one shard, at least one correct replica equivocated ($5f + 1$ and $f + 1$ must overlap in at least one correct replica). We have a contradiction.

Commit Slow-Path, Abort Slow-Path. Assume that a client generates a C-CERT and an A-CERT, both of which went slow path. Recall that each slow path certificate requires $n - f$ matching ST2R replies from the logging shard. As correct replicas never change their decision, assembling two sets of $n - f$ matching ST2R replies would require correct. replicas to equivocate. We have a contradiction.

In all cases, we show that there cannot be both a C-CERT and A-CERT for a given transaction T.

$\square$

From Lemma 2 it follows that no two correct replicas can ever process a different outcome (commit/abort) for a transaction T. Thus, given a fixed set of total transactions, all replicas are eventually consistent.

Lemma 3. *If T_i has issued a C-CERT and T_j conflicts with T_i, then T_j cannot issue a C-CERT.*

A transaction T_j conflicts with T_i if adding T_j to the serialization graph would create a cycle. If T_i is in $Commit \cup Prepared$ at a replica, the MVTSO concurrency control check will return abort (Lemma 1).

Proof. As T_i has committed, a client has generated a C-CERT

If a client generates a C-CERT for T, then *at every involved shard* at least $3f + 1$ replicas voted to commit the transaction (the slow path requires 3f+1 votes, the fast path 5f+1), and consequently at least $2f + 1$ correct replicas.

Assume by way of contradiction that a client C has created a C-CERT for T_j. There are two ways of achieving this: either C generates a C-CERT through the fast path or it does so through the slow path. Note, that since T_i's and T_j' conflict, their involved shards must intersect in one common shard S_{common}.

Fast Path If this C-CERT was created through the fast path, then there must exist at least $4f+1$ correct replicas on S_{common} that voted to commit T_j. But, at least $2f + 1$ correct replicas have already voted to commit T_i, and would thus vote to abort T_j. We have a contradiction. C cannot generate a C-CERT through the fast path.

Slow Path If this C-CERT was created through the slow path, then there must exist at least $2f + 1$ correct replicas on S_{common} (out of $3f + 1$) that voted to commit T_j (ST1R). But, at least $2f + 1$ correct replicas had already voted to commit T_i (and thus should vote to abort T_j). As there are only a total of $4f+1$ correct replica, this means that one correct replica voted to commit T_j despite it creating a cycle in the serialization graph with T_i (violating Lemma 1). We have a contradiction, C cannot generate a C-CERT through the fast path. $\square$

Theorem 1. *Basil maintains Byzantine-serializability.*

Proof. Consider the set of transactions for which a C-CERT could have been assigned. Consider a transaction T in this set. By Lemma 2, there cannot exist an A-CERT for this transaction. By Lemma 3, there cannot exist a conflicting transaction T' that generated a C-CERT. Consequently, there cannot exist a committed transaction T' in the history. The history thus generates an acyclic serialization graph. The system is thus Byz-Serializable. $\square$

B.2 Byzantine Independence
Theorem 2. *Basil maintains Byzantine independence in the absence of network adversary.*

We show, that once a client submits a transaction for validation, the transaction's result cannot be unilaterally decided by any group of (colluding) Byzantine participant, be it client or replica.

Proof. First, we observe that a Byzantine client may never independently choose the result of a transaction. It requires evidence (through SHARDVOTES and V-CERT) supplied by replicas.

Second, and as a direct consequence, once a client has prepared a transaction, it cannot unilaterally cause dependent transactions to abort.

Third, both abort and commit quorums contain at least one correct replica. Fast-path commits/aborts also necessarily contain at least one correct replica. Consequently, if a transaction commits (or aborts), at least one correct node voted to commit (or abort). A set of Byzantine replicas cannot, on their own, decide the outcome of a transaction.

Finally, a set of Byzantine replicas cannot force a client to read an 1) imaginary read 2) a stale read. Specifically, a client reads committed reads only if there are associated with a valid C-CERT. Moreover, clients read from the $f + 1$th latest read version returned. This ensures that the returned version is at least as recent as what could have been written by an correct node. $\square$

We note that this is not sufficient to guarantee Byzantine independence when an adversary controls the network. A network adversary could systematically reorder transactions at all replicas such that these transactions could abort for instance.

B.3 Fallback Safety
In the absence of an elected leader (Phase 1 and 2 of the Fallback protocol), clients are simply echoing ST1 or ST2 messages and follow the same rules as the traditional execution. Byzantine clients equivocating are further indistinguishable from concurrently active clients. As such, all theorems in continue to hold.

We further note that a fallback leader is only elected when a client detects inconsistent ST2R messages, i.e., when a transaction decision was logged on the slow path. Thus we only need to consider the effects of the fallback protocol on decisions that were generated through the slow path. Moreover, slow-path C-CERT and A-CERT consist exclusively of messages sent by the logging shards (fallback replicas operate only on the logging shard). We thus consider this shard only in the rest of our proofs. We show that 1) if a slow path A-CERT or C-CERT already exists, the fallback protocol never produces a conflicting decision certificate) and that b) if no decision certificate exists, any decision C-CERT/V-CERT created respects a decision made by some client (i.e. the decision respects the transaction's original two-phase commit shard votes).

For convenience, we re-state the decision reconciliation rule used by the fallback: the fallback replica proposes a decision $dec_{new} = maj(\{\text{ELECTFB}.decision\})$. We note that matching views are not required here.

We start by showing:

Lemma 4. *Fallback leaders cannot propose decisions that contradict existing slow path* C-CERT/A-CERT.

Proof. First, we note that, if a correct replica sends an ELECTFB message for view v, it has necessarily adopted a view $v' \geq v$ and thus will never accept a decision for a smaller view (1).

For any existing slow path C-CERT/A-CERT, the associated decision must have been adopted and logged by at least $3f + 1$ correct replicas (through ST2 and ST2R messages) in a matching decision view v. Without loss of generality, let the decision be Commit (the reasoning for Abort is identical) and the corresponding decision certificate be a C-CERT. Let R_{FB} be the **first** fallback leader to be elected for a view $v' > v$. At least $3f + 1$ correct replicas must be in v', since $4f + 1$ ELECTFB messages are required for a leader to be elected in view v' (2).

We note first that any existing C-CERT with view $v \leq v'$ must have been constructed from ST2R messages sent by correct replicas **before** moving to view v'. This follows directly from (1): correct replicas do not accept a decision for a smaller view. As $4f + 1$ matching replies are necessary to form the C-CERT, it follows that at least $3f + 1$ correct nodes have responded Commit in the ST2R message (3).

Correct replicas never change their vote. By (1) we have that $3f+1$ correct replicas must be in view v' and by (2) that $3f+1$ correct replicas voted Commit in view $v < v'$. It follows that at least $2f + 1$ correct replicas provided a Commit decision to the fallback leader R_{FB} in view v'.

Given the decision reconciliation rule, the fallback leader is guaranteed to receive $2f + 1$ Commit decisions, the only new decision that R_{FB} in view v' may propose is therefore Commit.

By induction, this holds for all consecutive views and fallback leaders: if there ever existed a $3f+1$ correct replicas that logged decision $dec_v = d$ in view v, then those same replicas will only ever log $dec_{v'} = d$ for views $v' > v$, since a fallback leader will always see a majority of decision d.

Consequently Basil fulfills Lemma 4. $\square$

Next, we show:

Lemma 5. *Any decision proposed by a fallback leader was proposed by a client.*

Proof. Let R_{FB} be the **first** elected fallback leader that proposes a decision (let its view be v).

By design, a correct replica only sends a message ELECTFB once it has logged a decision. Thus R_{FB} is guaranteed to receive a quorum of $4f + 1$ ELECTFB messages all containing decisions. As the fallback waits for $4f + 1$ messages, one decision must be in the majority. By the decision reconciliation rule, R_{FB} proposes the majority decision.

By assumption, R_{FB} is the first fallback leader to propose a decision. Thus, all decisions included in correct replicas' ELECTFB messages were a) made by a client, and b) are consistent with the provided shard votes (correct replicas will verify that ST2 messages have sufficient evidence for the decision).

Since any majority decision comprises at least $\geq 2f + 1$ ELECTFB messages, it follows that at least one was created by a correct client, and is hence valid. Consequently, any decision that R_{FB} can propose must have been issued by a client.

It follows that all correct replicas will receive valid decisions. In view $v + 1$, all decisions from correct replicas forwarded to the next fallback leader will also be valid. The same reasoning held above regarding the decision rule applies. By induction it follows that for all future views and respective fallback leaders, any (valid) proposed decision must have been proposed by a client.

Aside: Note, that if R_{FB} instead is Byzantine, it may collect two ELECTFB message quorums with different majorities, and equivocate by sending different (valid) DECFB messages to different replicas. In this case, different correct replicas may not adopt the same decision (thus precluding the generation of a decision-certificate), but any decision is nonetheless valid as it was originally proposed by some client. $\square$

We conclude our proof:

Theorem 3. *Invoking the fallback mechanism preserves Theorem 1 and Theorem 2.*

Proof. Lemma 5 states, that a fallback leader can only propose decisions that were proposed by clients. Lemma 4 additionally guarantees, that once slow path C-CERT/A-CERT exist, the fallback mechanism cannot change them. It follows that Lemma 2 still holds. Consequently, since any C-CERT/A-CERT generated through the fallback mechanism are indistinguishable from normal case operation, Theorems 1 and 2 remain valid. $\square$

B.4 Fallback Liveness

We first show that during sufficiently long synchronous periods, the election of a correct fallback leader succeeds. Concretely, we say that after some unknown *global synchronization time* (GST), an upper bound Δ hold for all message delays.

We note that replicas enforce exponential time-outs on each new view: a replica will not adopt a new view and start a new election until the previous view leader (whether client(s) or Fallback replica) has elapsed its time-out.

For convenience, we re-iterate the view change rules (§5, box 4b): (*i*) If a view v appears at least $3f + 1$ times among the *current views* received in *InvokeFB* (and v is larger than the replicas current view), then the replica adopts a new current view $v_{new} = v + 1$. (*ii*) Otherwise, it sets its current view to the largest view that appears at least $f + 1$ times among *current views* and is larger than the replicas current view. When counting how frequently a view is present in *current views*, R uses vote subsumption: the presence of view v counts as a vote also for all $v' \leq v$.

Theorem 4. *A correct client can reconcile correct replicas' views in at most two round-trips and one time-out.*

Proof. A client must provide $3f + 1$ matching view responses in the *InvokeFB* message (rule (*i*)) in order for replicas to adopt the next view and send a ELECTFB message. This implies that if a correct replica is currently in view v, there must exist at least $2f + 1$ correct replicas in a view no smaller than $v - 1$ (A). If a client cannot receive such a quorum, e.g. due to temporary view inconsistency, it must reconcile the views first. This is possible in a single additional step: By (A) any set of $4f + 1$ replica responses must contain at least $f + 1$ correct replicas' votes for a view $v' \geq max(correct.views) - 1$. Using rule (*ii*) a replica may skip ahead to v'. We point out, that while only two round-trips of message delays are required, a client may still have to wait out the view time-out between $v - 1$ and v (where v is the highest view of any correct replica).

If only a few replicas are in the highest few (i.e. less than $f + 1$ - in which case we could potentially catch up), and $4f + 1$ replicas are in the same view after using rule (*ii*), then a client does not even need another roundtrip (and a time-out) in order to guarantee the successful leader election (if all $4f + 1$ replicas send an ELECTFB message. If some are Byzantine, this might not happen of course).

Thus it follows, that a correct client requires at most two round-trips and one time-out to bring all correct replicas to the same view. $\square$

Next, we show that during sufficiently long periods of synchrony, a correct fallback is eventually elected.

Lemma 6. *After GST, and in the presence of correct interested clients, a correct fallback leader is eventually elected.*

Proof. In the presence of a correct interested client, Byzantine clients cannot stop the successful election of a new Fallback by continuously invoking a view-change on only a subset

of replicas. This follows straightforwardly from the fact that after GST, once the time-outs for views grow large enough to fall within Δ, a correct client will (*i*) bring all correct replicas to the same view, and (*ii*) there is sufficient time for the fallback replica to propose a decision before replicas move to the next view (and consequently reject any proposal from a lower view).

Moreover, election is non-skipping, as a correct client will broadcast a new-view invocation to all replicas.

Since fallback leader election is round-robin, it follows that a correct fallback replica will be elected after at most $f + 1$ view changes, with a sufficiently long tenure to reconcile a decision across all correct replicas. $\square$

Theorem 5. *A correct client eventually succeeds in acquiring either a* C-CERT *or* A-CERT *for any transaction of interest.*

Proof. First, we note, that a timely client can trivially complete all of its own transactions that have no dependencies. However, if a client is slow, or its transaction has dependencies, it may lose autonomy over its own transaction. For a given client c, we define the set $Interested_c$ to include its own transactions and all their dependencies, as well as any other arbitrary transactions whose completion a client is interested in.

We distinguish two cases for each $TX \in Interested_c$ that has timed out on its original client: (*i*) An interested client manages to receive a C-CERT/A-CERT by either issuing a RP message and receiving a Fast-Path Threshold of ST1R messages for all involved shards, or by issuing a new ST2 message and receiving $n - f$ ST2R messages from the logging shard. In this case, a client is able to complete the transaction independently as any client may broadcast C-CERT's/A-CERT's to the involved shards. Theorems 1 and 2 are maintained as this case follows the normal-case protocol operation.

(*ii*) An interested client cannot obtain decision certificates and starts a Fallback invocation. By Lemma 6, during a sufficiently long synchronous period and the presence of an interested client a correct Fallback replica will be elected after at most $f + 1$ election rounds. Such a Fallback replica will reconcile a consistent decision across all correct replicas in the log-shard, thus allowing the interested client to receive a $n - f$ matching ST2R messages. This allows it to construct the respective C-CERT/A-CERT, allowing it to complete the Writeback Phase.

$\square$