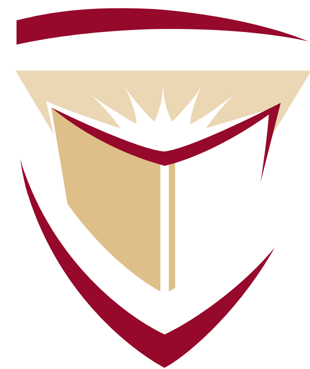
Basil: Breaking up BFT with ACID (transactions)

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

Harshilsinh Solanki(40298679)

Date:02/04/2025



UNIVERSITÉ Concordia UNIVERSITY

Agenda

- Background and Motivation
- Basil Overview
- System Overview
- Overview of Workflow
- Execution Phase
- Prepare Phase(Stage 1 & Stage 2)
- Writeback Phase
- Transaction Recovery: Fallback Scenario
- Evaluation
- Critical Analysis
- Conclusion

Background & Motivation

- Existing BFT System are rely on totally ordered logs and sharded architectures, which impact the on throughput , latency and transaction flexibility.
- Bottleneck situation Existing system are having (single point failure)(processing all request sequentially can become bottleneck)
- Although some BFT Use sharding for parallel transaction. But transaction that access in disjoint shared can execute concurrently , but in operation within shared are still ordered.
- Drawback of existing architecture:
    - They pay performance penalty of redundant coordination(for committing in ds, totally order in shared operation)
    - Fairness issues with leader based sharding
    - Limited Transaction Expressiveness(read/ write)

How Basil Overcome this Issues?

- Basil, a serializable BFT key-value store that implement the abstraction of a trusted shared log, Novel design address each of the drawback of traditional BFT systems:
  - It borrows databases ability to leverage concurrency control to support highly concurrent but serializable transactions, thereby adding parallelism to the log.
  - It sidesteps concerns about the fairness of leaderbased systems by giving clients the responsibility of driving the execution of their own transactions.
  - it eliminates redundant coordination by integrating distributed commit with replication. (Merging the commit & replication) so that, in the absence of faults and contention, transactions can return to clients in a single round trip.
  - It improves the programming API, offering support for general interactive transactions that do not require a-priori knowledge of reads and writes.

Basil - Overview

- Basil Introduced the Two complementary notions of correctness.
  - **Byzantine Isolation** – Ensures correct clients observe only valid database states produced by other correct clients.
  - **Byzantine Independence** – Prevents Byzantine actors from controlling transaction outcomes.
- **Independent Operability** – Enforces correctness on a per-client and per-transaction basis, avoiding pessimistic locks.
- **Optimistic Concurrency Control (OCC)** – Enhances parallelism while mitigating Byzantine interference.
- **Multiversioned Timestamp Ordering (MVTSO) Variant** – Reduces transaction abort rates while preventing Byzantine disruptions.
- **Novel Fallback Mechanism** – Allows clients to complete pending transactions without blocking non-conflicting operations.

System Properties of BASIL

- **Byzantine Serializability** is a way to ensure that, even if some users (called Byzantine clients) act in bad or unpredictable ways (e.g., lying, cheating, or sending incorrect data), the rest of the system still behaves in a reliable and consistent manner for **honest users**.
- But Still Byz-serializable system could still allow Byzantine actors to systematically abort all transactions. Hence, by defining 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**: Basil ensures that bad or malicious clients (Byzantine actors) cannot control or block the actions of honest clients.
- **Basil** ensures that honest clients can still perform their operations even if some clients are trying to cause problems, as long as not everyone is working against them. This is a key advantage over leader-based systems, where bad actors can easily team up to block honest users.

System Overview

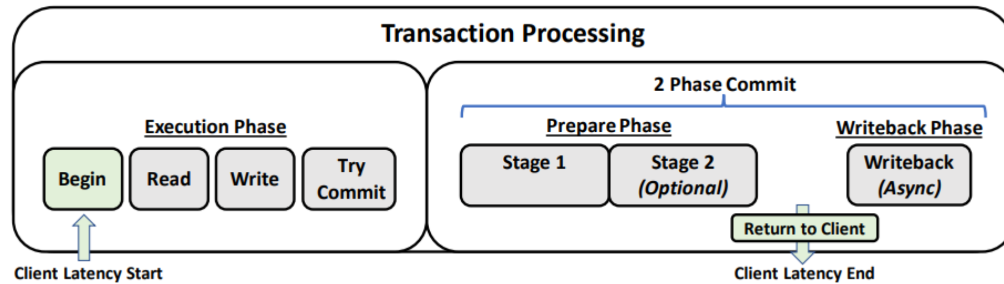
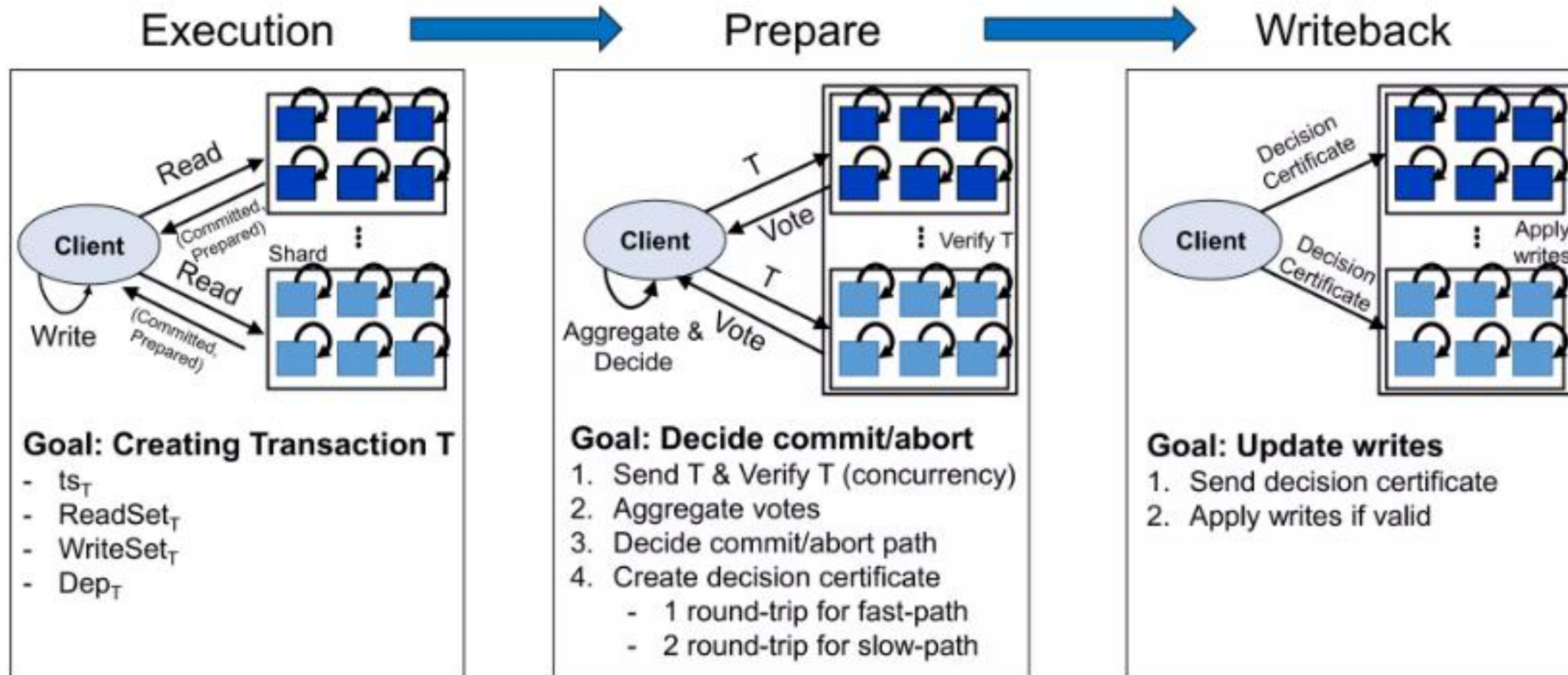


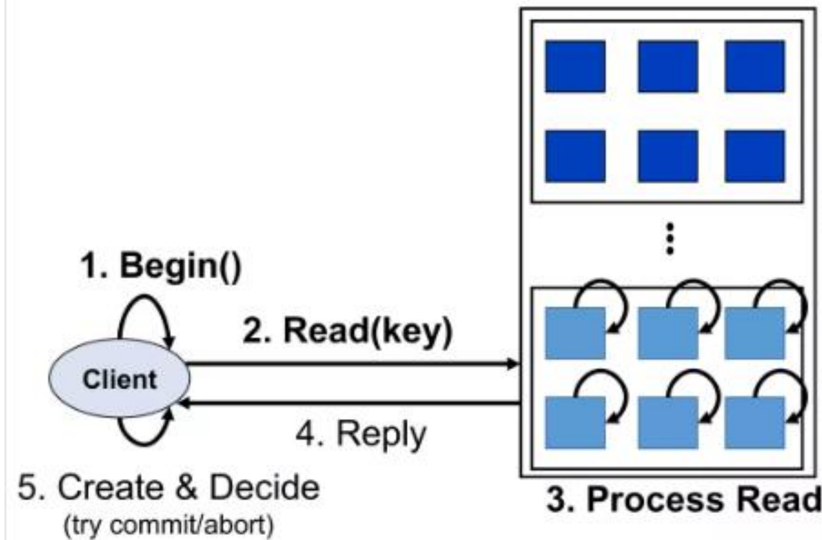
Figure 1. Basil Transaction Processing Overview

- BASIL transaction processing consists of **three phases**:
- **Execution Phase:** Clients execute transactional operations. Reads are sent to remote replicas, while writes are buffered locally. The system supports **interactive and cross-shard transactions**, allowing dynamic queries across multiple shards.
- **Prepare Phase:** Each shard **votes** on whether committing the transaction maintains **serializability**. Replicas within a shard can process votes **out of order** for performance optimization.
- **Writeback Phase:** The client aggregates shard votes, determines the final transaction outcome (commit or abort), notifies the application, and asynchronously updates replicas. This ensures transaction decisions remain valid even under **Byzantine or benign failures**.

Overview of Workflow

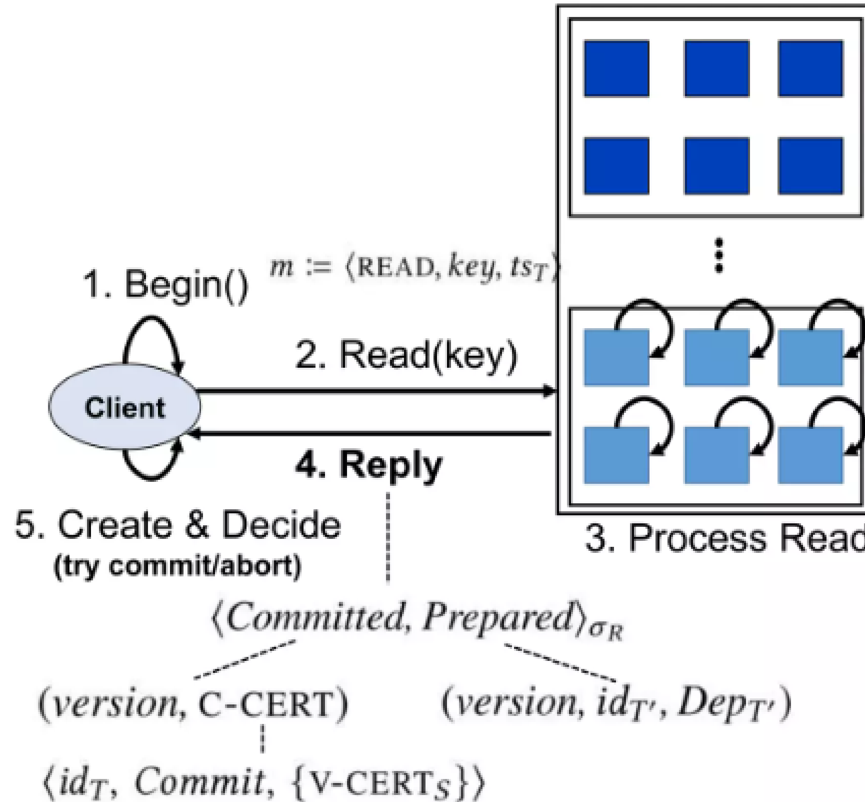


Execution Phase



- **Begin()**
  - Client choose timestamp ts_T := (Time, ClientID) for transaction T
- **Read(key)**
  - Client sends m:=<READ, key, ts_T> to at least 2f+1 replicas in an involving shard
- **Process Read**
  - Replicas validate timestamp; only accepts it if ts_T is smaller than $R_{Time} + \delta$ (= localClock + delta)
  - If valid, update key's RTS (read-timestamp) to ts_T

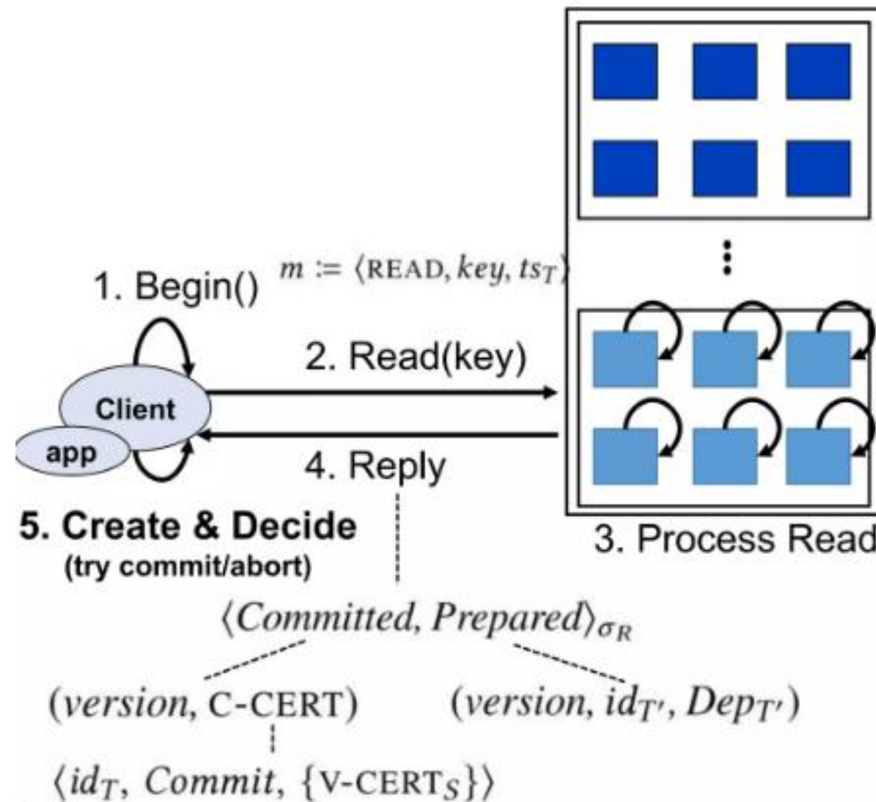
Execution Phase



- **Reply**
  - Replica returns <*Committed*, *Prepared*>
  - *Committed*
    - *version*: Latest committed version
    - C-CERT: commit certificate (later explained)
  - *Prepared (on-going tx)*
    - *version*: Latest prepared version
    - *T'*: A transaction that created prepared version
    - $id_{T'}$: TxId of *T'*
    - *DepT'* : Dependencies of *T'*

    → *T'* cannot commit unless all the transactions in $Dep_{T'}$ commit first

Execution Phase



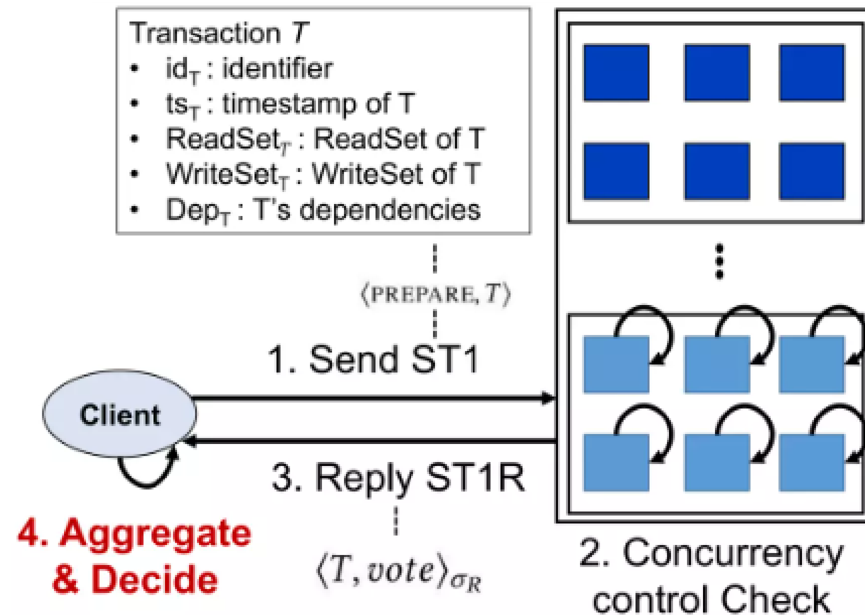
- **Create ()**
  - Client waits for at least f + 1 valid replies
  - **Chooses the highest valid timestamped version** (prevent stale read)
  - Add (key, **version**) to ReadSetT
  - Add (**version**, $id_{T'}$) to Dep_T
    - If the **version** was prepared but not committed
- **Decide()**
  - After execution of T is finished, **application** tells Client whether commit/abort
  - **Decide to Abort()**
    - Client asks replicas to remove its read timestamps from all keys in $ReadSet_T$
  - **Try to Commit()**
    - Client initiates Prepare phase
    - Create transaction T

Prepare Phase

- (Stage 1) Aggregating votes for the final decision
  - Client aggregates votes for the transaction T
  - Based on voting results, determines one of
    - 1).Commit-Fast 2).Abort-Fast 3). Commit-Slow 4)). Abort-slow

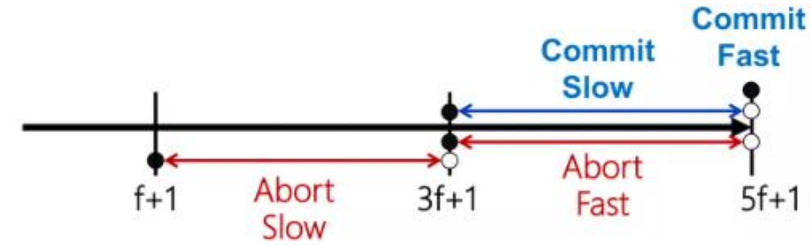
- (Stage 2) Making decision durable(only for Slow-path)
  - Ensure client’s final decision durable across failures.

Prepare Phase (Stage 1)



- **Send ST1**
  - Client sends <PREPARE, T> to involving shards
- **Concurrency Control Check (con't)**
  - Replicas locally checks validity of T for vote
- **Reply ST1R**
  - Replica replies a vote (commit/abort) for T
- **Aggregate & Decide**
  - Client aggregates votes and determines one of 1) commit-fast, 2) abort-fast, 3) commit-slow, 4) abort-slow
  - Proceed to Writeback phase for fast-path
  - Proceed to Stage 2 for slow-path

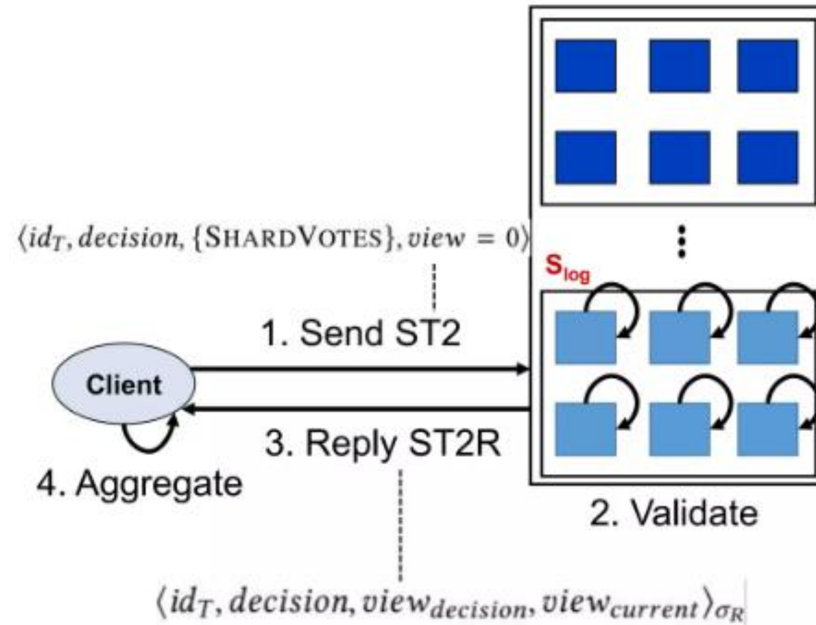
Client’s Final Decision



- Based on the aggregated vote results of ST1R

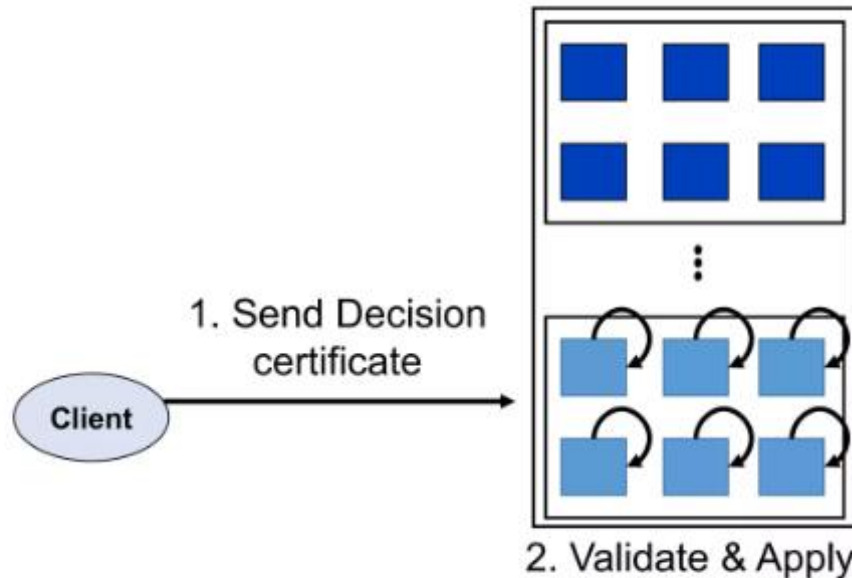
Decision	Condition	Result	Output
Commit-Fast	5f+1 Commit votes	Proceed to **Writeback**	V-CERT : <idT, S, Commit, {ST1R}> // fast shard, durable
Abort-Fast	3f+1 ≤ Abort votes	Proceed to **Writeback**	V-CERT : <idT, S, Abort, {ST1R}> // fast shard, durable
Commit-Slow	3f+1 ≤ Commit votes < 5f+1	Proceed to **Stage 2**	Vote tally : <idT, S, Commit, {ST1R}> // slow shard, not durable
Abort-Slow	f+1 ≤ Abort votes < 3f+1	Proceed to **Stage 2**	Vote tally : <idT, S, Abort, {ST1R}> // slow shard, not durable

Prepare Phase (Stage 2) Making Decision Durable



- **Send ST2**
  - Client chooses **one shard** (S_{log}) in Stage 1 deterministically using id_T
  - Client sends ST2 = <id_T, decision, {ShardVotes}, view=0> to replicas in S_{log}
  - {ShardVotes} ; votes of all shards
  - view ; 0 for client (others for recovery)
- **Validate**
  - Replicas in S_{log} validates ST2
- **Reply ST2R (ACK)**
  - $view_{decision}$, $view_{current}$; used for recovery
- **Aggregate**
  - Waits for $n - f$ matching ST2R messages
  - Create $\text{V-CERT}_{S_{log}} := \langle id_T, S, decision, \{\text{ST2R}\}\rangle$

Writeback Phase

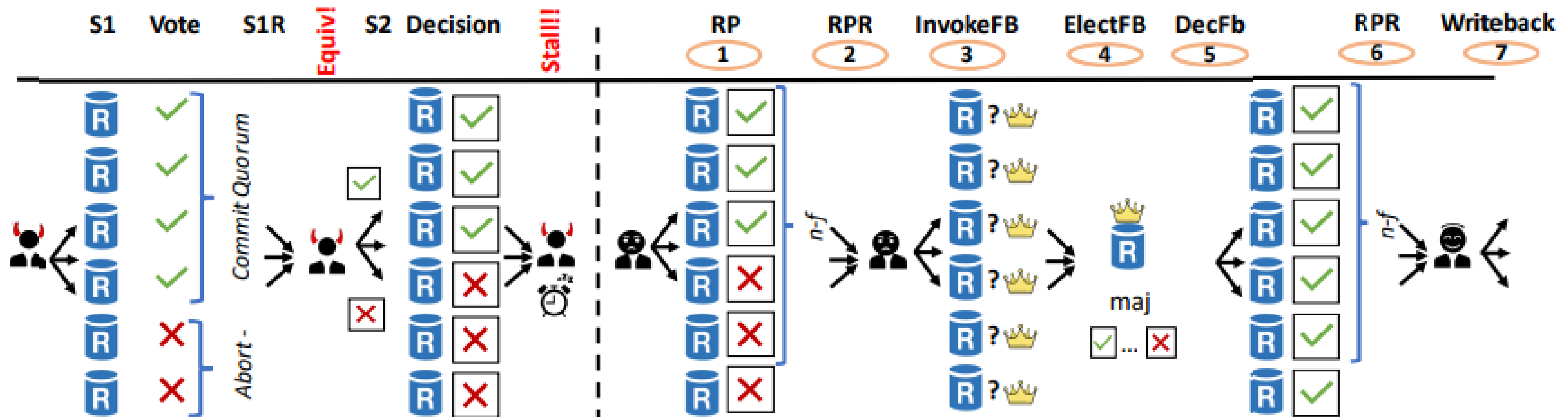


- **Send decision certificate**
  - Client asynchronously broadcasts decision certificate (C-CERT or A-CERT) to all shards that participated in the Prepare phase
    - C-CERT : $\langle id_T,\ Commit,\ \{\text{V-CERT}_S\}\rangle$
    - A-CERT : $\langle id_T,\ Abort,\ \{\text{V-CERT}_S\}\rangle$
- **Validate & Apply**
  - Replica validates certificate and applies writes to datastore

Transaction Recovery: Fallback Scenario

- In the BASIL protocol, the **fallback scenario** is triggered when Byzantine nodes cause a **stall** in the transaction commit process. This happens due to **equivocation, conflicting votes, or quorum failures**.
- To recover, BASIL initiates a **fallback mechanism** with the following steps:
    1. **Recovery Prepare (RP & RPR):** Nodes exchange their latest transaction states.
    2. **Invoke Fallback (InvokeFB):** If inconsistency is detected, a fallback process is initiated.
    3. **Elect Fallback Leader (ElectFB):** A correct leader is elected based on majority agreement.
    4. **Decision Fallback (DecFB):** The leader makes the final decision to commit or abort.
    5. **Writeback:** The decision is propagated to all nodes, ensuring system consistency.
- There will be a two cases in which Recovery can be made:
    - Common case: matching results; Commit Quorum (3f+1) or Abort Quorum(f+1)
    - Divergent case: unmatching results; Commit Quorum (3f+1) and Abort Quorum(f+1)

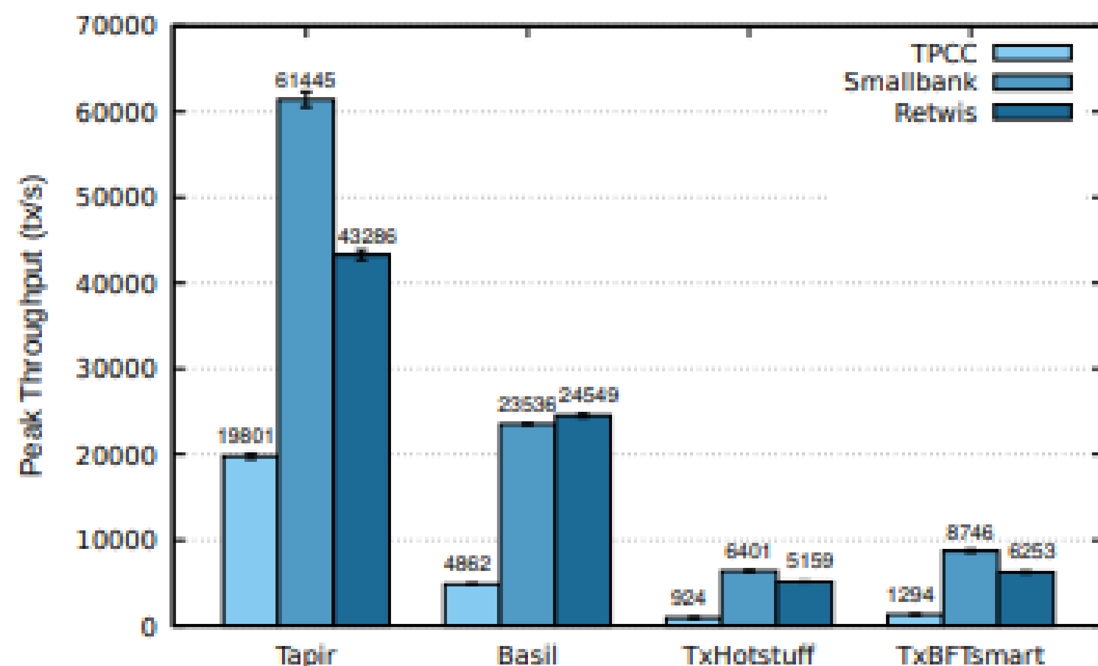
Transaction Recovery: Fallback Scenario



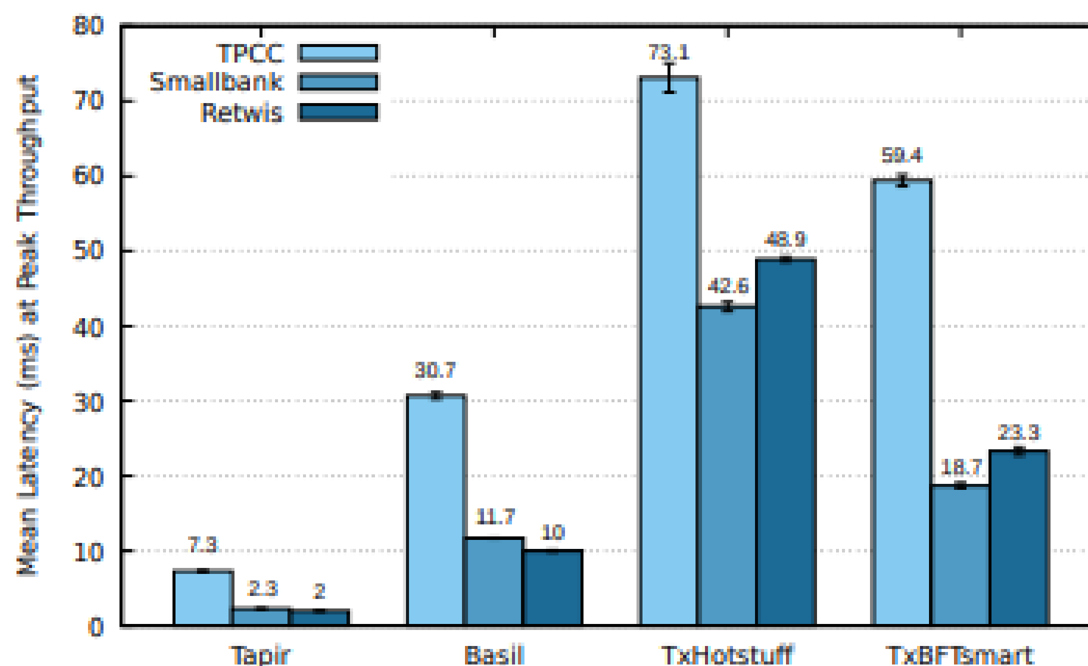
Evaluation

- Baselines
  - TAPIR(SOSP’15); non-byzantine distributed database
  - TxHotStuff (PODC’19), TxBFT-SMaRt (DSN’14) ; tailored for optimistic tx
- **Experimental Setup:**
  - **CloudLab , m510 machines (8-core 2.0 GHz CPU, 64 GB RAM, 10 GB NIC, 0.15ms ping latency), run experiments for 90 seconds (30s warmup/cool-down)**
  - **Client execute in a closed-loop, reissuing aborted transaction using a standard exponential backoff**
  - **F=1 (n= 2f+1 for TAPIR, 3f+1 for HotStuff and BFT-SMaRt)**
- Experiments:
  - Performance (bench: TPC-C, Smallbank, Retwis)
  - BFT overhead (Bench: YCSB-T)
  - Basil under (Client) Failure

Evaluation: Baseline Comparisons



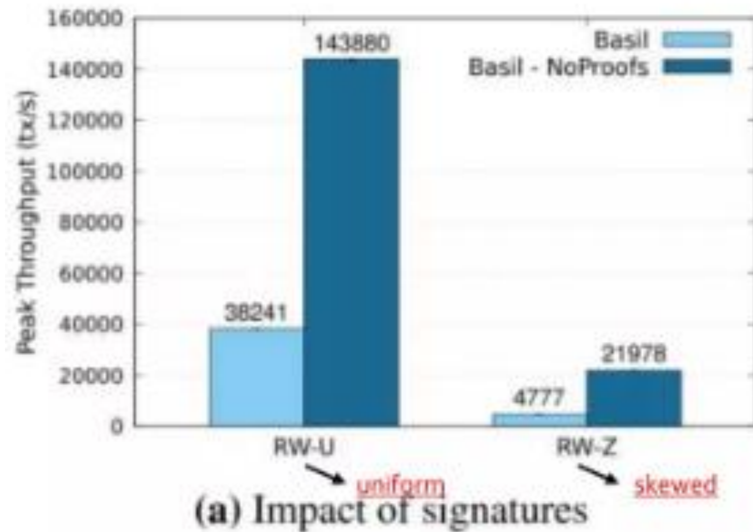
(a) Throughput in tx/s



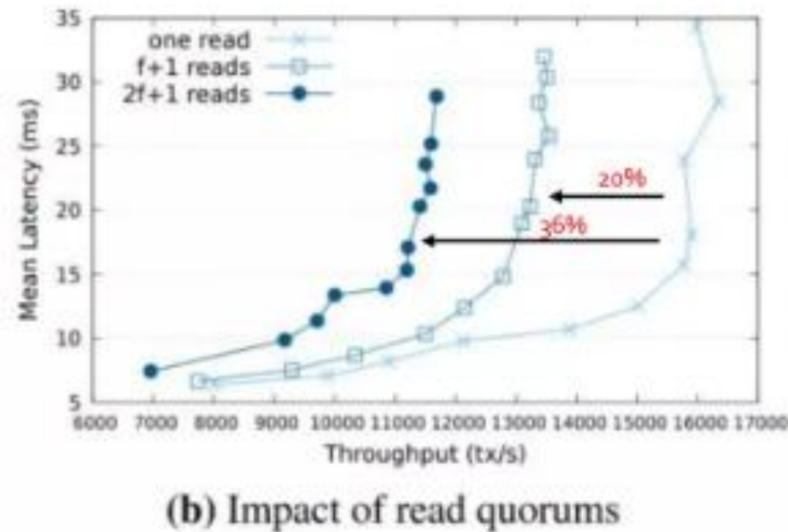
(b) Latency in ms

Figure 4. Application High-level Performance

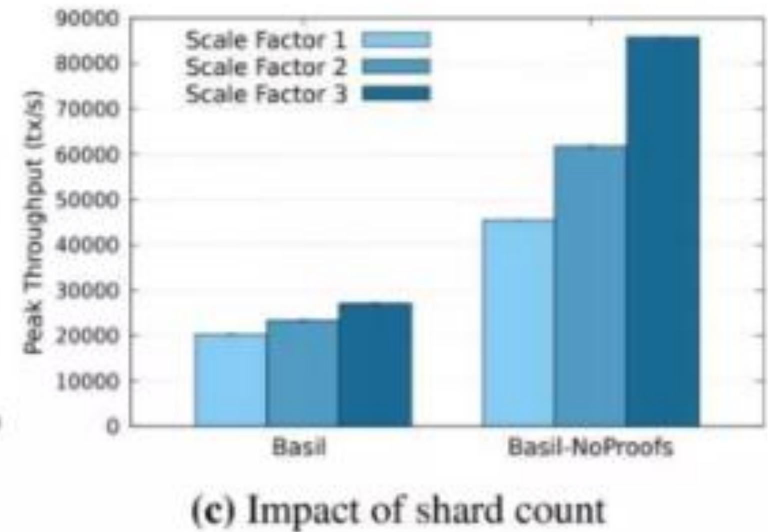
Evaluation: Overhead



(a) Impact of signatures

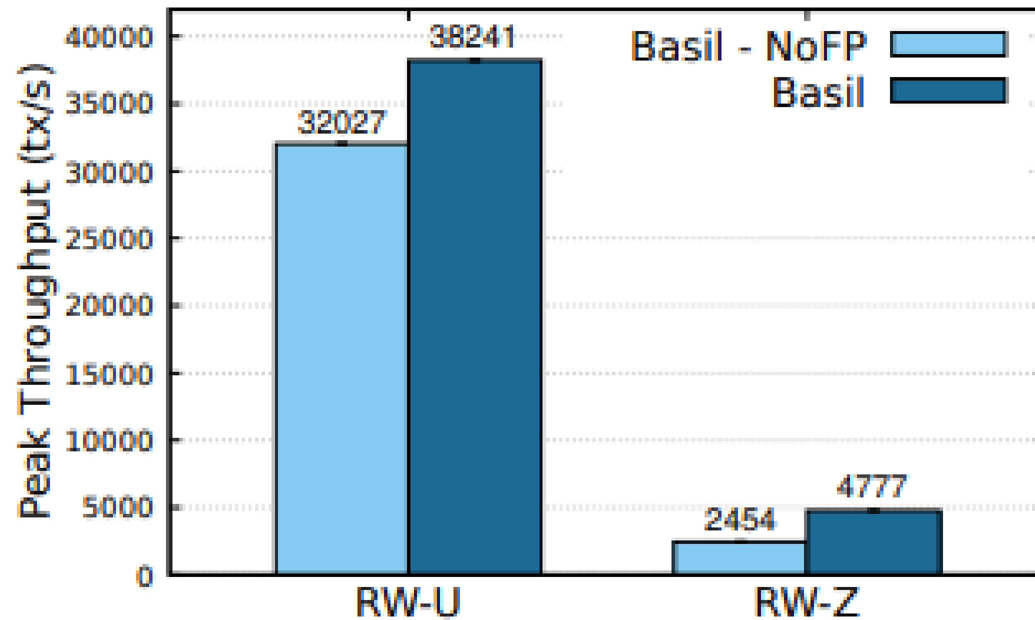


(b) Impact of read quorums

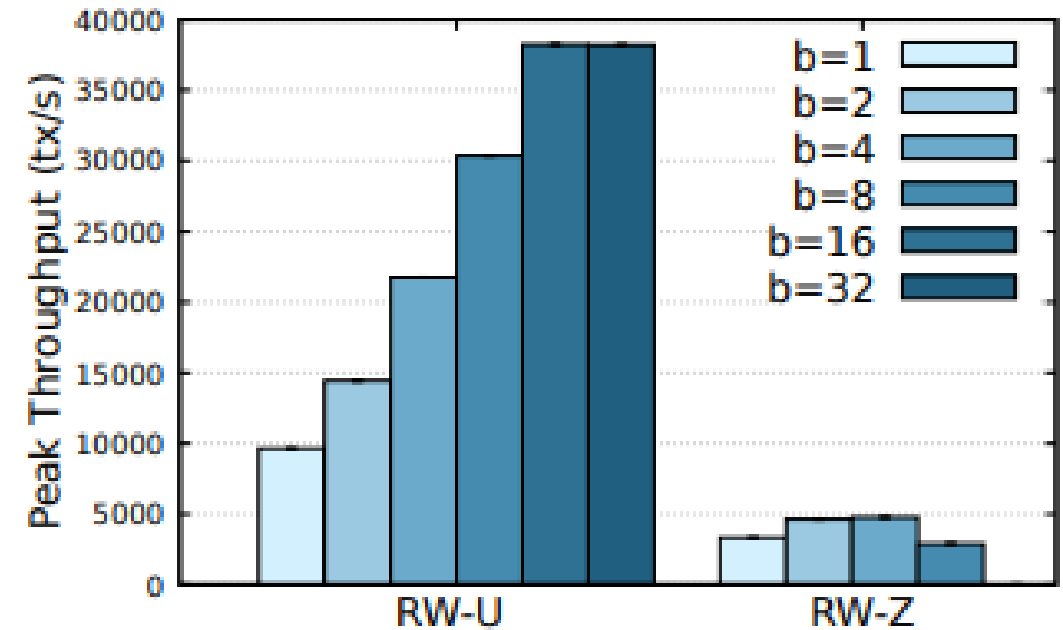


(c) Impact of shard count

Evaluation: Basil Optimization



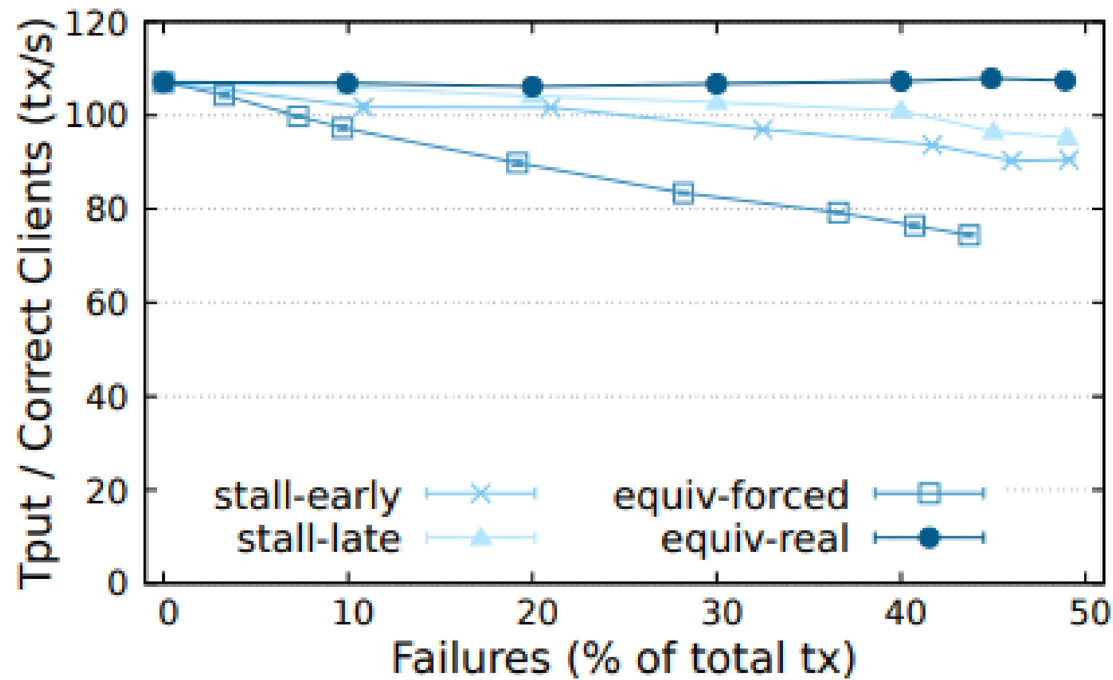
(a) Throughput with/without fast path



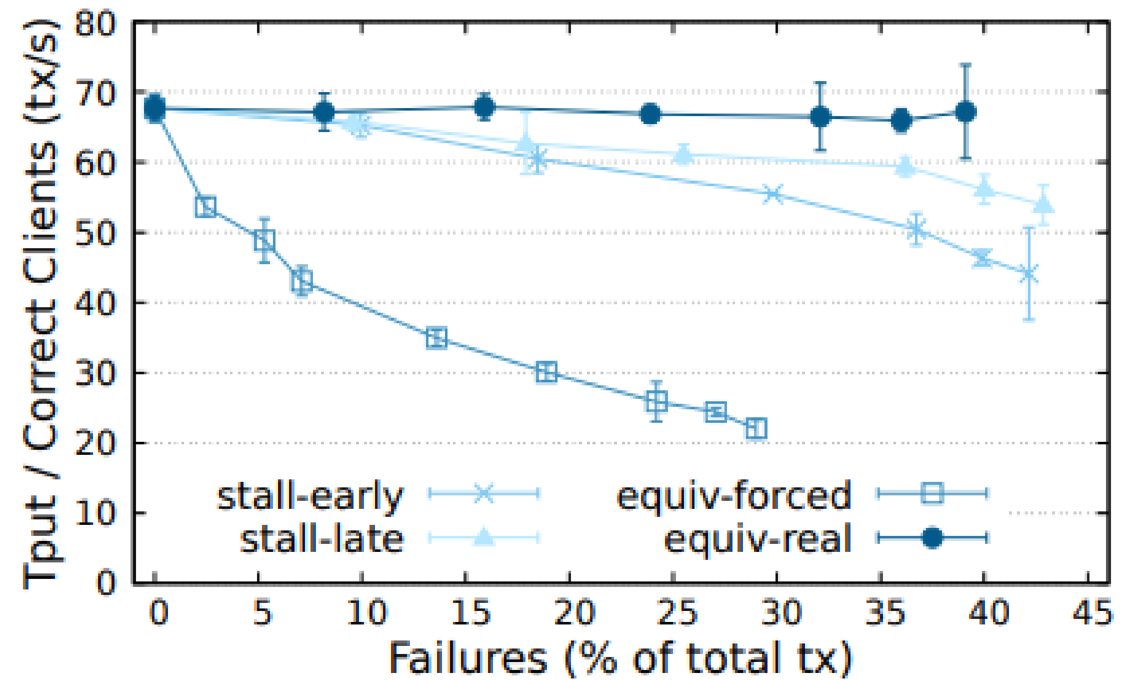
(b) Throughput vs. batch size

Figure 6. Basil Optimizations

Evaluation: Basil Under Failures



(a) Tput vs. Failures (RW-U)



(b) Tput vs. Failures (RW-Z)

Figure 7. Basil performance under Client Failures

Critical Analysis

- **Strong Points;**
  - Ensures serializability of transactions while maintaining transaction independence, even in environments with contention and Byzantine failures. This ensures correctness and consistency without compromising performance.
  - Combines 2PC with consensus for faster decision-making (96% of transactions commit/abort in one round-trip).
  - Extensive benchmarking with TPC-C, Smallbank, and Retwis, showing robust performance across different workloads.
- **Weak Point:**
  - While Basil improves throughput with its fast path, the overhead associated with signature verification during transactions could still be a bottleneck in high-traffic environments, particularly when handling large-scale transactions with frequent cryptographic operations.
  - The paper doesn't provide much insight into how long-running transactions (those that span multiple rounds or involve complex operations) are handled. These could present performance challenges, especially in real-world systems where transactions often need more than just basic read-write operations.F=1 (n= 2f+1 for TAPIR, 3f+1 for HotStuff and BFT-SMaRt)

Conclusion

- This paper presents Basil, the first leaderless BFT transactional key-value store supporting ACID transactions.
- Basil offers 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.
- During fault and contention-free executions Basil commits transactions in a single round-trip.

Thank you