Indicus: Unchaining Byzantine Databases

Your N. Here
Your Institution

Second Name
Second Institution

Abstract

Your abstract text goes here. Just a few facts. Whet our appetites. Not more than 200 words, if possible, and preferably closer to 150.

1 Introduction

This paper introduces Indicus, the first leaderless transactional key-value store that is robust in the Byzantine Fault Model. Indicus aims to mitigate the tension between real-world, highly commutative transaction workloads striving for scalability, and the simplicity that totally ordered ledger abstractions such as Blockchains or State Machine Replication offer. Specifically, this paper asks the question: how can we enable *mutually distrustful parties* to consistently and reliably share data, while minimizing centralization.

FS: this maybe sounds too much like a permissioneless blockchain

The ability to share data online offers exciting opportunities, however, increased datasharing also raises the concern of how to *decentralize trust*. In banking, systems like SWIFT (cite) enable financial institutions to quickly and accurately enable cross-institutional transaction clearance, at the cost of placing their trust in the centralized SWIFT network. In manufacturing, online data sharing can improve accountability and auditing amongst the globally distributed supply chain, but there may not be an identifiable source of trust. Consider the supply chain for the latest iPhone: it spans three continents, and hundreds of different contractors [1] that may neither trust Apple, nor each other, yet must be willing to share and agree on information concerning the construction of the same product. Moreover, single authorities that distribute their dataceneters globally (i.e. Google, Amazon), may not even trust their *own* datacenters located in authoritative domains or legislations out of its control.

Recognizing this challenge by both the research and industrial communities, much effort has focused on enabling shared computation between mutually distrustful parties in the context of Byzantine Fault Tolerance (BFT) and Blockchains. Systems proposed in the literature of BFT provide the abstraction of a totally ordered request log; the log is agreed upon by the n participants in the system, of which at most f can misbehave. In the Blockchain world, Bitcoin and Ethereum have become popular distributed computing platforms providing the same log abstraction while aiming for decentralizing trust and open membership. At the intersection, systems such as Hyperledger Fabric or Ethereum Quorum aim to cater to Blockchain markets while leveraging traditional BFT approaches for scalability. Efforts to incorporate Blockchain/BFT technologies into market ready infrastructures are pervasive (cite all kinds of examples.)

FS: examples: Applications of permissioned Blockchains? Hyperledger Fabric: Health-care/Pharma, Honywell uses for aircraft, Telecom settling
Applications that use Ethereum Quorum? (based on Raft or Istanbul BFT): JP Morgan auto inventory, Statefarm, Starbucks, Copyright (Louis Vuitton)

This paper argues that there exists a fundamental mismatch between the implementation of a totally ordered log and the reality of much large-scale distrubted processing. Many large-scale distributed systems consists primarily of unordered and unrelated operations. For example, a product supply chain consists of many concurrent steps that do not require ordering. Imposing an ordering on non-conflicting operations is not only often unecessary, but costly: participants in the shared computation must vote to order operations and serialize request execution accordingly, thus harming both throughput and end-to-end latency.

While there exists work on mitigating this scalability bottleneck through sharding (cite Omniledger, Chainspace, Callinicos), the latent total order requirement introduces uncesseary coordination overheads, as coordination is performed twice, at the level of individual shards, and across

shards (cite ordering paper). This is especially problematic when workloads are geo-replicated (citation?), or when, as in BFT, the replication factor is high. FS: sharding conflates two objectives: horizontal hardware scaling and transactional parallelism (which requires a "smart" locality mapping in order to be efficient") Furthermore, achieving a total order often necessitates a single point of centralization, in order to propose a sequencing order. This is especially undesirable in trust concerned settings as such a sequencer exposes not only a scalability bottleneck but a fairness vulnerability. A sequencer may be biased, frontrun requests or censor others and is inherently antithetical to the need of decentralized, trustless solutions. Lastly, these systems support transactions under the assumption that their read and write operations are known a priori, which limits the set of applications that they can support. In summary, prior BFT solutions succumb to one or more of the following fallacies, some of which are correlated: They a) impose a restrictive total order, b) use leader in some form or another, c) assume fixed transaction sets or d) incur redundant coordination overheads when sharding.

Leveraging commutativity between transactions is not a new idea. As a research trend of mitigating the scalability bottleneck, EPaxos (cite), TAPIR (cite) and CURP (cite) only consider the ordering between potentially conflicting operations. However, these systems assume the stronger crash-failure model and are non-trivial to extend to the Byzantine model, so that they cannot directly solve the problem of data sharing among mutually distrustful, malicious or arbitrarily failing parties. Existing research, in essence, is either attempting to build concurrency control and sharding functionalities over BFT replication, or integrating these functionalities into a crash-failure replication protocol. In this paper, we will show how to build these desiring functionalities inside a BFT replication protocol. Specifically, our goal is to *provide the illusion of a centralized shared log, rather than the non-scalable reality of a totally ordered log*.

While copious efforts exist to design decentralized systems that exploit transaction semantics for the crash failure model, few, if any attempts have been made for the Byzantine Fault Model. This naturally raises the question, why so? One explanation is that in Byzantine Systems, some centralization is in fact highly desirable as it simplifies the problem by identifying a single point of accountability. A natural way to improve both scalability and fairness is to avoid this bottleneck, at the cost of paving the path for a wider set of undesirable phenomena. The challenge of a leaderless byzantine system is simple yet daunting: It empowers both malicious users and system components to collude and misbehave in potentially unaccountable ways. In this paper we will show to overcome this challenge by designing and implementing the leaderless BFT system Indicus that avoids near-all centralization while replicating interactive ACID transactions in a partial order.

The principle of *partial ordering* forms the core of Indicus' design. Unlike State Machine Replication (SMR) based systems that achieve agreement on computation (i.e. state transitions) by imposing a total order on execution across all replicas, Indicus executes transactions speculatively at clients, while validating and replicating *all* transactions in arbitrary order at any given replica. Abstractly, Indicus proposes each transaction for a seperate concurrent binary consensus instance and enforces an implicit partial order by aborting transactions when inconistencies arise. In order to maintain a serializable transaction history, Indicus assigns each transaction an optimistic timestamp and uses a variation of Multiversioned Timestamp Ordering (MVTSO) for concurrency control. Ordering conflicts between transactions whose interleavings would violate prescribed Isolation guarantees are broken based on the given timestamps and speculative execution results.

Overall, the Indicus design has the following implications:

- Indicus is robust to censorship and frontrunning as there exist no central authority that admits and orders transactions
- Indicus allows commutative/non-conflicting Transactions to both be executed and validated out of order, thus maximizing parallelism. FS: (This should increase throughput and reduce latency, because clients arent waiting for all previously sequenced tx to finish. Consequently the tail latency does not dominate throughput as much.)
- Indicus minimizes the state, communication and computation load on replicas, as Clients serve as both execution hub and broadcast channel for their own Transactions. This avoids quadratic communication communication complexity in the normal case. FS: (Theoretically always, but we keep some for practicality in the fallback)
- As any Quorum system, Indicus is inherently load-balanced as there exists no leader bottleneck and all replicas have equal responsibilities.
- In Indicus liveness is a client local property. Unlike SMR, where the entire system halts during view changes, Byzantine participants may stall system progress only for the objects their transactions touch. Hence, the system appears live to any non-conflicting Transaction.

To achieve these properties Indicus incurs two main trade-offs. First, shifting responsibility from replicas to clients comes at the cost of higher computational requirements for clients, which may not be tolerable for all applications. In practice we envision Indicus clients to be dedicated transaction

managers, rather than end users. Second, like all optimistic concurrency control schemes, Indicus is vulnerable to congestion. When contention is high, abort rate soars; Indicus is not designed for such applications. Exploring tradeoffs between client/replica responsibilities as well as pessimistic concurrency control mechanisms are potential avenues for future work.

Results: ...

To summarize, this paper makes three main contributions:

1. It offers definitions for transactional Isolation guarantees for the Byzantine Fault Model
2. It presents the design, implementation, and evaluation of the first leaderless ACID transactional system that tolerates Byzantine Failures
3. It presents two variations (Indicus3 and Indicus5) of a client centric agreement mechanism that trade-off performance for replication degree.
4. FS: our MVTSO?

1.1 Intro-Notes

1.2 Header

- This paper introduces Indicus
- interactive, scalable DB in the byz fault model
- first to be fully client driven

NC: this might be personal taste, but I don't think being client-driven is particular novel or important. Being leaderless on the other hand seems like something we should emphasise - therefore: fair and can scale to replica bandwidth/processing limits

1.3 Context

- people/entities would like to share data and/or split replication costs. Keep joint records of relevant data (i.e. for cross Audit): Imagine a transaction clearance between banks (Libra as example: consortium of parties to manage a currency), a supply chain, different medical providers sharing patient data (for easy migration) - any system where users would like to easily migrate between providers. - Parties no longer from the same authority, potentially untrusted. –> byzantine fault tolerance not for increased software failure resistance (bugs etc), but to tolerate some level of misbehavior - While BFT was not warranted in single entity replication (extra cost for some more FT), it should be the standard for heterogenous replication. This is the basis for Blockchain where BFT is the norm.

NC: I don't think we need to be defensive about BFT. It's the norm in blockchain and blockchain is "hot" -Alternatively, big corporations that use transactional systems that are geo-replicated might be worried about administrative domains: I.e. google spanner having replicas located in authoritarian regimes or out of its own control. does not "trust" its "own" replicas.

- - SMR powerful fault tolerant technique. Creates illusion of single Database, thus simplifying application design. Replication is masked. NC: this comes out of no where

- ACID transactions: Absolute data integrity. Simplifies concurrency control. Intuitive data access logic. Concurrent access to retail inventories, bank balance, stocks is unavoidable. I.e. serializable isolation makes reasoning simple for application developers.

Interactive TX: Most general for any kind of application. FS: Any system that provides Interactive TX can handle One shot for example.

- Goal: Make BFT accessible to the mainstream. Requires scalability and low latency. Cost needs to be tolerable - more reasonable if replication is done as consortium (each party pays for subset). NC: I don't think we particularly need to discuss the benefits of transactions or the benefits of interactive transactions yet. I think it's cleaner to say that blockchain, backed up by BFT, has the problem of being totally ordered. Moves the text more quickly.

1.4 Existing systems shortcomings

- systems were designed for different assumptions: single authority low replication (historical use cases: aviation, space, nuclear power) vs high replication degree in a consoritum of untrusted parties
- non-interactive transactions
- totally ordered
- leader based: fairness and scalability bottleneck
- Crash Failure model
- single sharded

1.5 Key insights

- ??
- mismatch between requirements of a transactional store and implementation of totally ordered log
- implicit partial order suffices for TX
-EVERYTHING is a partial order: Entire Life cycle is parallel if non-conflicting

- flip the problem of trying to add scalability to existing BFT systems. add BFT to efficient CF DBs

- put clients in charge: responsible for their own liveness; a natural way to scale a system

NC: is this the "standard" way to scale a system?

- do single slot binary consensus
-

Challenge Empowering byzantine Clients is complicated and dangerous

1.6 Positive Implications of Insight/Design

- robust to censorship and frontrunning (if network not adversarial) as there exist no central authority
- allows commutative/non-conflicting Transactions to both be executed and validated out of order, thus maximizing parallelism.
- minimizes the state, communication and computation load on replicas, as Clients serve as both execution hub and broadcast channel for their own Transactions.
NC: I don't think everyone would agree that this is a good thing, replicas are big beefy machines at the datacenter, whereas clients might be lightweight devices where you explicitly don't want to run much computation - As any Quorum system, Indicus is inherently load-balanced as there exists no leader bottleneck and all replicas have equal responsibility
- In Indicus liveness is a client local property. Unlike SMR, where the entire system halts during view changes, Byzantine participants may stall system progress only for the objects their transactions touch. Hence, the system appears live to any non-conflicting Transaction. View changes are parallel to one another and are only triggered when a participant is interested, which is inline with our "insight" that everything is a partial order.

1.7 Results discussion

- latency should not be that much higher compared to Tapir
- throughput better than an atomic broadcast version
- abort rate doesnt rise too fast; mvtso improves abort rater over OCC (if not, then its useless overhead and we can use OCC)

1.8 Limitations

- Like any speculative system, Indicus is vulnerable to high contention in which case it needs to resort to aborts
- This can especially be exploited by byzantine participants if the network is adversarial.
-

1.9 Contribution summary

- Present definitions for what it means to provide ACID (specifically Isolation) guarantees in a byzantine setting
- We present design, implementation and evaluation of Indicus, a fully client-driven BFT ACID transactional system
- Design of a leaderless, client-driven execution and replication protocol that does not enforce a total order
- Present 2 flavors of Indicus: Indicus3 and Indicus5 that explore the trade off between replication degree and performance (latency and computational overhead) - The design of a more aggressive optimistic concurrency control scheme that is robust to byzantine behaviors

2 Definition- notes

Goals: - Define what liveness guarantees there should be in a byz system
- Want to offer interactive ACID TX in a byz setting
- be robust to byz contention

Challenges - Clients can intertwine their fate with byz participants. When can/should liveness be guaranteed and when should it not. Under which network assumptions?
- Unclear what correct behavior should be. Thus, we need to define what ACID (specifically Isolation) means in a byzantine context. byzantine Clients can execute arbitrary transactions, might not care for guarantees. Begs the question: Should one enforce "safety" for them too?
- What does it mean to be robust to byz contention? To what extent is independence possible

2.1 Model and Definitions

2.1.1 System Model

We assume a partitioned Database where data objects are spread uniformly across shards (randomly or based on locality). For fault tolerance we assume that each Shard contains 5f+1 replicas where f is the number of (static) faulty or compromised replicas. A faulty replica may behave in Byzantine fashion, i.e. in addition to crashing or omitting messages it may send arbitrary messages and deviate from prescribed protocols in any way. We denote any participant (replica or client) that follows the protocol as *honest*, while faulty participants are dubbed *byzantine*. We assume there may exist a finite but unbounded number of byzantine Clients that may deviate from the protocol arbitrarily. We further assume a strong, but static adversary that can freely coordinate the faulty participants. We do, however, assume the existance of sufficiently hard cryptographic primitives that allow for private/public key signatures and collision-resistant hashes that cannot be compromised by byzantine participants. We denote a signed message m, signed by principal p as $\langle m \rangle_{\sigma p}$.

We make no assumption on network synchrony in order to maintain safety, but in some cases may provide liveness only when the network is synchronous and messages are delayed by no more than a fixed but potentially unknown window. This is consistent with known impossibility results [FLP]. However, unlike traditional State Machine Replication

protocols in which the liveness of all Clients is correlated with the fate of the system (or often more specifically a leader), our system guarantees liveness not on a system basis, but on a per client basis. Concretely, we only guarantee liveness to clients that follow the protocol. Conversely, an honest client only loses liveness (even in an asynchronous setting) when it intertwines its fate with byzantine clients.

Application services may restrict the influence of Byzantine Clients on the Database state by authentication and enforcing access control. Beyond these measures, byzantine Clients that follow the protocol are indistinguishable from honest clients and may read and write at their leisure. While potential damage to the Database state cannot be avoided, it can be re-traced by auditing the Transaction logs.

2.1.2 System properties

We offer Clients an interactive Transaction interface that implements ACID transactions. While all ACID guarantees may be violated by individual byzantine Replicas, the system maintains the illusion of an ACID compliant state to all honest Clients. Moreover, we guarantee to Clients that the Database is *byzantine Serializable* as defined below. Intuitively this Isolation level guarantees that all honest Clients experience the Database as if there serializable. In order to formally capture this we lay some ground work: [1]

Let $Op = \{r, w\} \times K \times V$ and $Dec = \{Commit, Abort\}$ be the sets of possible operations and decisions respectively, where K is the set of existing data items (keys) and V the range of possible values. A *request* $req \in (Op \cup Dec) \times C$ maps any such operation or decision to the issuing Client from set C. We denote with $Hon \subseteq C$ the subset of honest Clients.

History H. Informally, a H contains the operations (read/write) and decisions (commit/abort) of every transaction issued in the system. Formally, we define a *history* H as a finite sequence of requests.

We define a projection $H|_c$ as the subsequence of requests in H that were issued by Client c. A sequence of requests $s = req_i \dots req_{i+t}$ in $H|_c$ form a *Transaction* if req_i is the first request by Client c or $req_{i-1} \in Dec \times c$, and if $req_{i+t} \in Dec \times c$.

We further define:

Honest History H(P). Given protocol P, A *history* H is *honest* if it was generated by participants who all follow P. I.e. concretely, $H(P) \equiv H = H|_{Hon}$. and

Honest-View Equivalent. A *history* H is honest-view equivalent to a *history*H' if the Operations and Decisions of all honest Clients are the same and if the final writes are the same.

Byz-I Given a protocol P and an isolation level I: A history H is *byzantine-I* if there exists an honest history *H'* such that H is honest-view equivalent to H' and H' satisfies I.

Thus, informally a byzantine Isolation level states that the state that honest Clients experience must be explicable by an execution in which all participants were honest. Note, that we make no assumptions on the state a byzantine client *chooses* to experience; i.e. byzantine Clients may read arbitrarily. This definition captures the requirements for any byzantine tolerant protocol that strives to maintain Isolation level I.

We further define the ideal progress properties to limit the influence Byzantine Clients have on system throughput.

Byzantine Independence Given a protocol P and an honest Client c. The result of a request r issued by c cannot be determinsistically decided by byzantine participants.

If the network is controlled by an adversary, this property is unattainable for Indicus. In order to offer this property we must strengthen our assumption on the network. Concretely, while the network may be asynchronous, the adversary does not control the network, and hence, may not reliably impact results.
An exception to this are wide-ranged flooding attacks (ddos) which are beyond the scope of this work. We point out, that a strawman system offering interactive transactions and speculative execution while relying on Atomic Broadcast for Validation ordering suffers the same fallacy: A byzantine leader may always frontrun requests. In fact, even with strengthened network assumptions, such a a system could not offer Byzantine Independence.

We adapt and define gracious and uncivil executions based on cite(aardvark) to match our model. (i.e. network not sync either). **Gracious Execution** An execution is gracious iff (a) the execution is synchronous with some implementation-dependent short bound on message delay (b) all clients and servers behave correctly and (c) there is no contention on the objects relevant to the execution. **Uncivil** An execution is uncivil iff (a) there is no bound on message delay (asynchrony) and (b) up to f servers and an arbitrary number of clients are Byzantine

[1] Defs for commands etc in accordance with BFT DUr. cite

3 Indicus

In the following we outline the Architecture and Protocols of Indicus, the first highly scalable (hopefully) database that tolerates both byzantine clients and replicas.
FS: Should Include what Indicus is supposed to do: Allow Clients to issue interactive TX. No Censorship, high throughput, low latency. Should be resilient to Byzantine Participants: Clients should experience a safe and live system.

4 Arch-Notes

Goals: - Indicus should be a transactional database
- It should be replicated and tolerate Byzantine behavior for fault tolerance and trust assumptions
- It should offer the most general set of transactions, i.e. Interactive Transactions
- It should enforce ACID guarantees, specifically Serialiability as Isolation level
- It should not be affected by censorship
- It should enforce only a partial order on Transations
- Leaderless
- Shardable

Challenges - Byzantine behavior requires additional safety precautions; higher replication degree
- Providing interactive Transactions requires offering Clients an interface for execution.
- Enforcing only a partial order implies the necessity of conflict checks. Concretely, maintaining serializability requires the use of Concurrency control.
- If we want no total order and no censorship then we cannot have a leader. This means empowering Clients which is dangerous in a byzantine system: exposes more opportunities for misbehavior.
- No leader means replicas may go out of sync, need to maintain consistency view of the database somehow
- Multi-shard transactions require additional coordination

4.1 Architecture

In Indicus, Clients are more than external participants who propose Transactions to be executed. Instead, Clients are first class citizens that rejoice in the fair treatment of democracy and take part in everyday system activities. Of course, such priviledge comes at the cost of added responsibilities. Concretely, Indicus does not provide a continuously and mysteriously operating black box Transaction machine, but rather offers Clients the tools to operate the system itself. One the one hand, the simple paradigm of letting clients work for themselves is more scalable as replicas need to do less work and can service more clients. On the other hand, it naturally incentivises clients to be industrious, as they hold the keys to their own liveness. Clients that do not meet a certain standard for productivity may be excluded by the system; The joys of being permissioned.

Indicus follows a traditional optimistic concurrency control architecture as shown in Figure 6. Clients speculatively execute their Transactions, issuing remote reads when necessary and buffering writes locally. As Clients execute their own Transactions they need not declare their read/write keys or values preemtively, but rather may conduct interactive Transactions, the most general Transaction model. Since execution is speculative and Clients are unaware of potential concurrency, they must validate their Transactions for Isolation correctness in order to be able to Commit. Intuitively, if there are no conflicting concurrent Transactions and a Client observed a consistent snapshot of the distributed Database state then it may commit, and otherwise it must abort or retry. Lastly, Transactions may span multiple shards, and hence the Commit/Abort decisions of each shard must be aggregated in a Two Phase Commit manner in order to finalize a safety appropriate result. Most notably, in Indicus not just exeuction is Client driven but the entire Transaction life cycle, thus maximizing scalability and fairness while putting each Client in charge of its own liveness. In the following we describe our Fault Model and define corresponding Transactional guarantees. We then outline the protocols for Execution, Validation and Writeback respectively.

5 Protocol-Notes

Structure: 0. Preamble what Indicus is and what it is supposed to offer. 1. Model., Properties 2. Architecture: Exec, Val, WB, Replica state?

3. Exec details: Read/Write Set computation 4. Concurrency Control check. Put in relation to Exec details 5. Validation Protocol: a) Voting and Decision rule component b) Logging. Explain similarity to other BFT protocols (logging could use anything, in 5f+1 it resembles Q/U because etc.) 6. Writeback and Multi-shard TX 7. Failures between 5/6 8. Optimizations: 9. Indicus3: Extra abort phase, different quorums (Validation and View change), Extra proofs 10. Correctness proofs: Only for 5f+1? - ACID: serializable specifically - Prove recovery maintains same result and consistency - Prove liveness (i.e. client has all tools necessary: dependency resolution, general aborts/abstains. Fallback live leader election.)

5.1 TX execution

Goal:
- clients should be able to read valid and consistent data from the database
- clients should see the most recent data

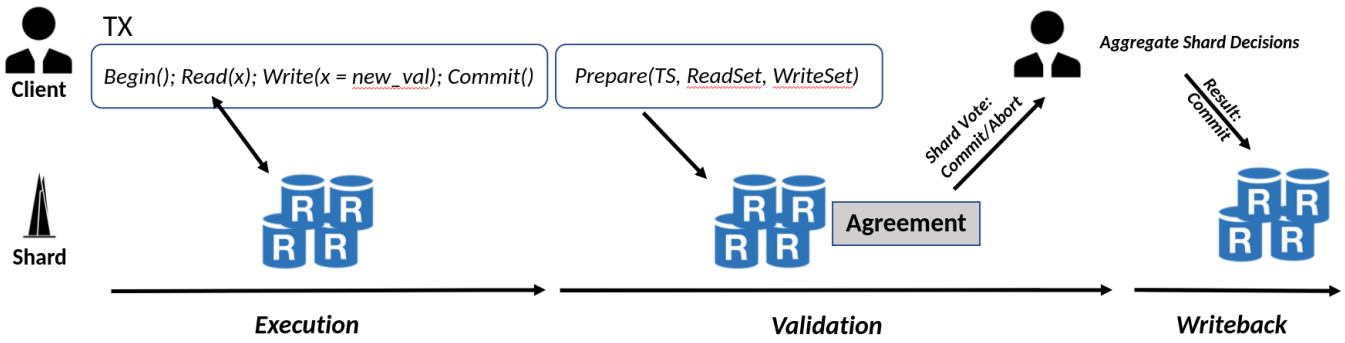

Figure 1: Transaction Lifecycle

Challenge
- Replicas can be out of sync and byzantine
- There might be concurrent transactions ongoing that are conflicting

Our Design:
- Clients speculatively execute their TX: Writes buffered locally, Reads go remote
- Reads can read committed and potentially committing values
- Necessary quorums sized so validity (and liveness) is maintained

Subtlelties: - unique TX ID – avoids equivocation and makes Tx uniquely indexable in efficient data structures - include dependencies in their TX (f+1 signed copies - Q: Why is this necessary? A: To make replicas believe it exists so that other clients are able to recover that knowledge.) -

5.2 Concurrency control

Goal:
(Exceptions, read leases, dependencies: Q: how to store full txid) - limit aborts

challenges:
- concurrent read/writes. Maintain isolation: serializability
- geo-distributed so execution and validation takes long –> more interleavings
- clocks are not perfectly synced
- minimize unecessary aborts
- byzantine clients can create artificial congestion

- Maintain Isolation while replicas out of order. In total order protocols (SMR), deviation from the "common" vote signals misbehavior, whereas for us that is not the case.

Our Design:
- TSO based concurrency control: Assign optimistic timestamps that define serialization order
- Enforce loose bound on the timestamps. This is fine for safety, but necessary to restrict progress damage. Alternative way is to have a TS gen phase, but this induces a RTT and certificate to attach so it is undesirable. Instead we ignore Transactions that are too far in the future, as such reads could make a lot of writes abort. Reads are not affected by writes with large timestamps, since we allow to read from RTS time.
- Reads and Writes are tested on Conflicts with previously committed or possibly committing Transactions. Evaluation happens based on TS and whether serialization order would be violated.
- MVTSO: 3 techniques to reduce the number of aborts:
- Read from TS, and not newest. This allows reads (especially long reads, which can happen in a WAN network) to avoid aborting due to later writes. This requires a multi version store.
- Read possibly committing writes. This is to avoid missing writes that should have been seen. Effectively minimizes the "lockout window" that the latency of validation-writeback phases incur.
- Reads issue RTS in order to acquire "locks" on concurrent writes that would cause reads to abort. This requires the Timestamp to be known in advance.

Subtlelties:
- concurrent read and dependency validation could make own dependencies abort. This can especially be strategically abused by byz clients. To avoid this, add dependency exceptions: Async read exceptions sent by honest clients to minimize this window.
- Read leases: Read "locks" could be arbitrarily acquired by byz clients without intention of completing a TX. Thus they are on a timeout. Larger writes are not affected anyways. Application option: decide who to grant read locks, since they are not a safety mechanism, but an additional progress "shield"
- Efficient dependency storage: Only claim dependencies on f+1 for several reasons: a) at least one honest believes this could commit, b) allows to only store TXIDS in a TX and not full dependency, because that dep can be recovered when nec-

essary - having info about deps is necessary for client liveness.

- how to bound dep depth? Reads return a field that says dep depth. Honest clients will not create deeper deps. Byzantine Clients may try, but cannot, because the f+1 signed dep messages include the full dep tree, and replicas will not accept any TX that have depth > some d. In practice we probably want d = 1, so the fallback starts directly for the immediate depdency.
- in general it was possible to "miss" comittable/prepared tXs, thats okay since reading prepared is just an optimization.

- Abort decisions need to come with proofs that client can validate: Avoids fake aborts. In 5f+1 the proof is implicit since f+1 abort votes are necessary.
- Optimization: Retries for Writes. Comes at a tradeoff for validation phase latency, so we elaborate only later.
-

Replica datastructures: - for efficient lookups we have following data structures: - Hashmap from TXIDs to protocol state (for easy management of ongoing TX). - Key value store for prepared TX, includes Writes, Reads, and RTS - Committed State: Key value store of the DB. Allows to Read. Allows to lookup for conflicts. - Commit Log: Set of TXIDS that are committed (maps TXID to TX) (allows to check if deps have alreaday committed. - Abort Log: See Commit Log (allows to check if dependencies have already aborted) - Dependency set: Map from TXID to waiting dependents (allows to unblock waiting dependencies when a TX finishes) - (optional: dependant to dependency mapping. Once mapping is empty, this TX is not waiting on anybody anymore)

5.3 Validation

Goal:
- maintain ACID guarantees: Isolation, Durability, Atomicity
- Be client driven, i.e. leaderless and out of order
- avoid frontrunning
- be scalable
- be fast

Challenges:
- Maintain Isolation while creating highest chance to commit
- Tolerate byzantine replicas voting dishonestly
- Tolerate Client failures: equivocation, crashes, stalling, replay
–> Answer: make final decisions idempotent
- enable consistent recovery
- minimize state at replicas while still guaranteeing correct agreement. Make sure local knowledge is enough to guarantee global correctness
- minimize necessary roundtrips and communication complexity

Basic design:
- Voting phase:
- Client sends to all, All to CC check, all reply
- Client makes decision based off Quorum and Decision Rule (have figure for it?)
- Metaphor: Instead of a leader serializing and deciding on a decision, the decisin was made jointly by all replicas. Hence, "voting" phase

- Decision rule:
- Since votes can be inconsistent (no total order enforced on replicas) we must aggregate them
- Must be designed in a way that maintains Isolation
- 3f+1 commit necessary. Vice versa, 3f+1 abstain necessary. Absense of enough commit votes requires pessimistic decision in abort.
- Wait up to time out for at least 2f+1. If timeout expires and less than 2f+1, keep waiting.

- Ordering decision is made without a leader (democratically): Fairness is up to the network

- Persistant Logging. 2pc analogy. (or is 3pc analogy better?)
- Want to make sure, that whatever client decision was made (equivalent to choice of Vote quorum), is going to be persistent and idempotent. I.e. if commit/abort is returned to the application any re-issue of the protocol (which can be necessary under faults) is consistent with the original decision.
- In 2pc decision is logged to persistant storage. That obviously doesnt work in this setting (byz client, even under crash you would like to make progress). So a Client "logs" the decision by replicating it consistently to the replicas.
- Client sends decision, gets back echo
- If enough consistent echos then the result may return. This guarantees that decision can be recovered.
- 4f+1 matching replies required: We will show recovery later.

- Any "normal" agreement protocol could be used once a "decision" has been cast, at this point it is just for replication. I.e. one could plug in a leader based consensus mechanism here, but its overkill and unecessary. There are no conflicts anymore, so an order isnt necessary either. Just unordered broadcast/receive is enough: This is exactly Q/U basically. "Contention" is a byzantine client equivocating. Q/U makes the observation that with multiple clients it would not be live, thus we introduce the fallback view change mechanism.

Subtlelties:
- 5f+1 commit fast path: 3f+1 abstain/abort fast path. (Cannot have these with retries). If Abort includes proof, then 1 abort fast path as well.
-

5.4 Writeback

Goal:
- Finalize commit/aborts
- Maintain consistency
- Enable garbage collection

Challenges:
- client failures (crash, equivocation)
-

Basic Design:
- Client uses the Logging certificates to proceed. Aggregates these certificates for every shard.
- Any Client can fulfill this role, arbitrary amount in parallel or even Replicas because this operation is guaranteed to be idempotent (due to the validation and recovery logic)
- Upon Receiving Commit Abort, Replicas update their data structures. Remove from Prepared strucutres, Add to Commit/Abort Logs. - Dont need a dedicated append only ledger (?). It would be different for each replica anyways. Can just store these logs as Hashmaps (i.e. reuse our lookup structures)
-

Subtlelties:
- Optimization: Single shard logging
- The logging phase is redundant if there are multiple voting shards involved. Instead, the votes could be aggregated BEFORE logging to form the decison and then only be logged on a dedicated shard. This saves communication bandwidth and makes recovery simpler, because there is a dedicated single shard where to look.

5.5 Optimizations

Goals:
- reduce write aborts
- reduce redundancy in validation
- reduce read lock impact (byz clients)

Challenges:
- defend against byz client abuse
- respect all shard results
-

- Pretty much mentioned in other sections already. Seems like its more suitable to have it with the contet right away.?

5.6 Failures

Goals:
- be robust to byzantine client

challenges:
- byzantine clients can stall or equivocate during Validation/Writeback
- replicas can diverge due to byzantine clients. Need to reconcile safely
- Granting honest clients liveness –> ability to "realiably" finish all Transactions.

Design:
- Since the system overall has no shared notion of progress, we maintain liveness only on a per client bases. Specifically, view changing is not only unecessary if nobody cares about a TX but also useless, as no "liveness" is maintained.
- View change seperately for every single TX, because every single TX is consensus on a binary register
- Since liveness is directly coupled to Clients, view changes should naturally be coupled to clients interest. However, if multiple Clients are able to run the protocol at the same time, then there might never be a conclusion. Electing a single client to be in charge would be very difficult and also not bound the view changes necessary for progress, because there is an unbounded fraction of byz clients
- Solution: Delegate view change to a dedicated fixed size group.. The replicas! Gives the guarantee, that when the network is synchronous, after at most f view changes it will succeed. Sync needs to be assumed and thats ok according to FLP.
- A little more subtle: After at most f view changes IF an honest client is involved, otherwise no guarantee

LIVE ELECTION
- Protocol (in theory): Clients issue View change requests for a TX they are interested in. Replicas ONLY move to the next view if a client proves to them, that a Quorum existed in the past view. This guarantees, that replicas cannot diverge arbitrily far in their views. More concretely, there is always >=f+1 honest replicas that dont diverge in views by more than 1 and there are < f+1 honest replicas that diverge from other replicas by more than 1.

- Protocol (in practice): To avoid byzantine Clients view changing arbitrarily long and driving up timeouts before honest Clients are interested we introduce some all to all forwarding. Add additional (off critical viewchange path) exchange to the Fallback replica to issue and disseminate certificates. (Technically equivalent to a Writeback if single sharded. If multi-sharded 2 considerations: a) if not optimized single shard logging, then still need to aggregate decisions, b) if single shard logging, then replica needs to be aware of all other shards in order to forward (can still Writeback locally).

SAFE RECOVERY:
- Protocol: Choose 2f+1 if existing, otherwise f+1, otherwise redo p2 based on p1. (p1 guaranteed to exist since they voted. Replicas only vote if they had the p1. Client gave it to them if they didnt have it before.) (If replica sees 4f+1 matching right away, it can return that to the client and also distribute

as certs)
- Clients can only use Quorums from matching views to return to the application (simple counterexample: 3 commits 3 aborts and then swap)
- FB can use decisions from multiple views for the decision rule (i.e. decision rule is view agnostic). When could this arise? Because newer views subsume older views and FBs might have been byzantine, or been replaced too quickly.
- Proof by Induction: If something returned then ... guaranteed to only ever issue that decision
-

Necessary Extras:
- A Client first asks all replicas for existing p2 state or potential certificates. An honest client doesnt need the view change in that case - it was in order to get such certificates.
- Client acquires current view from all replicas by doing so. Uses this to start a view change. If f+1 matching exist, client uses those to catch up replicas that lag behind proactively. (just speeds things up: dont need to send all 4f+1)
- A client simultaneously re-issues a p1 message, in case some replicas have never seen the TX
- A client is repsonsible for the Writeback (any client can do this): Any "interested" client receives the p2 decisions made from a fallback and attempts to return.

Subtletlties:
- (Maybe useless:) Only a client that provides a dep proof (f+1) or abstain proof is allowed to issue fallback. Doesnt make it impossible to start FB, but at least some hurdle (a client could always just receive such a abstain message with the proof from a byz colluder replica
- Fallback election is not started if that TX itself has a dependency - wait for all deps to be finished themselves. This is to distinguish whether a Client has been slow, or its blocking itself due to another dependency.
- Tx map to different initial fallbacks: I.e. (TXID + view) mod n

5.7 Garbage Collection

Challenge: - memory footprint grows large due to multi version - certificates impose large signature overheads

Options: - Need to bound and remove versions below watermarks - In 5f+1 can read based on f+1 matching committed. Dont need certs and can still be safe for recovery and robust to replicas "claiming random dependency aborts"

5.8 Discussion:

- Explain/Define Client relative liveness. We dont provide liveness if something is async, "liveness" ill defined for this sort of system. What it means: Clients that follow P experience liveness, but there is no system notion because there is no shared total ledger. Progress is client relative
- Several seperate binary consensus instances

optional Extras to mention: -Can use Witnesses instead of replicas to lighten replication burden - only need to contain key/version and not values

5.9 Indicus3

View changes: - bounding rule works differently: slightly weaker guarantee - Decision rule: In View change hierarchy!! I.e. higher view beats anything. Implication: If conflicting decision to previous view was made, then previous view vote could not have been final. 1. 1 p3 commit: use it to return 2. 1 p3 abort: re-issue p2 abort 3. 1 p2 commit: re-issue p2 commit 4. 1 p2 abort: re-issue p2 abort OR re-do p1, either works 5. Nothing: re-issue p2 using p1 decisions.

Want the same practical all to all between replicas + fallback async certificates

6 Protocol

Indicus comes in two different flavors, Indicus3 and Indicus5 respectively, relying on varying replication degrees. Indicus3 requires 3f+1 replicas per shard to guarantee consistency, the minimum bound necessary for BFT SMR. Indicus5 uses a higher replication degree of 5f+1 replicas, but in return brings down both gracious execution latency and complexity during uncivil executions. We argue, that unlike past system settings where a single authority strives to maintain the fewest amount of replicas necessary for cost considerations, consortium systems with naturally higher replication degree are willing to pay the additional price for performance. Moreover, rather than trying to reach the threshold of replicas to tolerate some number of faults, these settings may start with a fixed number of replicas and consider the ratio of faults to be tolerable. When argueing from this perspective, the perceived difference between tolerating 1/3 and 1/5 of replica failures may be negligible. For the simplicity of exposition we outline Indicus5 in detail below. In section X we describe the differences in Indicus3.

6.1 TEsting

NEED FIg with replica state. Need to use it to explain what "Prepared"/Committable means

6.2 Execution

Clients in Indicus both execute and submit their own Transactions. As previously defined a Transaction TX is a sequence of read and write requests that is ultimately terminated by a Commit or Abort decision. A TX object, as shown in Figure 2 records the execution state necessary for Validation.

TX

- ClientID
- ClientSeqNo
- ReadSet = {(key, version)}
- WriteSet = {(key, value)}
- dependencies = $\{\langle(key, version)_{f+1\sigma}\rangle\}$
- TXID = H(TX)
- Timestamp = (Time, ClientID) optional:, TXID) this is just deterministic tie breaker when a client misbehaves.

Figure 2: Transaction

Client execution conducts as follows:

1. **Write(key, value)**. A Client executes a request Write(key, value) by locally buffering (key, value) and returning. Concretely: $WriteSet = WriteSet \cup (key, value)$

2. **Read(key, TS, RQS)** Given hyperparameter Read Quorum Size (RQS), a Client performs a read on given key at timestamp TS by reading from RQS different replicas.

   A replica returns a pair $\langle(Committed, Prepared)\rangle_{\sigma_r}$, where
   $Committed := (value, version)_{cert}$ such that $(value, version) \in R.CommitLog$, $version = max(q) : (key, val, q) \in R.CommitLog \wedge v < TS\}$ and *cert* is a certificate proving (value, version) was allowed to commit (see Section Writeback) and
   $Prepared := (value, version)$ such that $(value, version) \in R.PreparedSet$ and $version = max(q) : (key, val, q) \in R.PreparedSet \wedge Committed < v < TS\}$. I.e. Committed is the largest committed write and Prepared is the largest uncommitted write that may be committed and would make Committed outdated.

   1. Reading from 1 Replica: could abort because reading arbitrarily stale. Alternatively: Read from f+1 always to reduce proof/memory costs: Could not get matching and hence fail read. Still possible to abort by reading stale data 2. Reading f+1 matching Prepared that are larger than the commit –> choose that and add dependency Guaranteed to see 1 thats not intentionally stale. But still arbitrarily stale if unlucky. 3. If read from 3f+1/2f+1 then its effectively a read lock. If 1 prepared read was enough then this would be sufficient to see the newest "potential" value. However that is not possible for liveness reasons. Note, could still have missed a prepared write and have to abort because we required f+1 matching. Why f+1? so it comes from 1 honest: we dont need proofs, and we know it is in the system for recovery. Still no guarantee that newest commit was seen, but some protection from other writes.

   (Note: the TS used here is not the final one, its just the tuple of time and clientId. (The triple later is just necessary to differentiate two TX by a client that were assigned the same Time Add Read set decision rule, put in correlation to RQS: FS: could only accept reads if f+1 matching always, then no proofs necessary, but then reads can fail Add exceptions etc.)

3. **Commit** A Client finalizes its execution, computes the final Timestamp and submits for validation.

4. **Abort** A client terminates execution, broadcasts a read-release for all potentially acquired RTS and returns. FS: A byz client may not release the RTS. To circumvent this, RTS are just leases. Moreover, at some point writes will overtake them

- Read proofs required for honest client correctness. Alternatively one can read from f+1 only, but that can result in failed/older reads - Do not need to include read proofs in the prepare. According to Isolation definition byzantine Clients can read whatever they want. Moreover, Reads only have very limited external effect. The value does not matter for the CC check. The version has bounded effect: If it goes towards 0, then it is just a check between timestamps as normally. If it goes towards the TS, then it will never abort. - Fallback not just useful for dependency cleanup, but also "unclaimable" dependencies that need to finish.

6.3 Validation

6.4 Concurrency Control

- Optimization: retries - heights - Dependency resolution tree - Equivocation not possible if TXidentifier a function with dep as argument - Cannot claim dep if not f+1 times (If you want to, you would require proofs again, which we try to avoid because dep trees can grow exponentially). More reads also more likely to commit –> f+1 guarantees 1 honest thinks it is legit. - Exception for depedency and early async read response to inform of exceptions - Read leases instead of unlimited locks - in practice only grant to timely clients (not a safety measure, but an increased progress guarantee) -

6.5 Consistent logging.

Principles and challenges
protocol overview: pic

Algorithm 1 MVTSO-Check(TX, TS)

if $TS > localClock + \delta$ **then**
  pass
for $\forall key, version \in TX.read\text{-}set$ **do**
  if $\exists TX2 \in CommitLog : key \in TX2.write\text{-}set \wedge version < TX2.TS < TS$ **then**
    return ABORT, *TX2, TX2.CommitCert*
  if $\exists TX2 \in Prepared : key \in TX2.write\text{-}set \wedge version < TX2.TS < TS$ **then**
    return ABSTAIN, *TX2*
for $\forall key \in TX.write\text{-}set$ **do**
  if $\exists TX2 \in Prepared/CommitLog \wedge TX \notin TX2.dep$: $TX2.read\text{-}set[key].version < TS < TX2.TS$ **then**
    return RETRY,
  if $\exists RTS \in key.RTS : RTS > TS \wedge TX \notin RTS.dep$ **then**
    return RETRY,
Prepared.add(TX)
while $\exists d \in dep : d \notin CommitLog \cup AbortLog$) **do**
  Wait
for $\forall d \in dep$ **do**
  if $d \in CommitLog$ **then**
    if $d.TS > TS$ **then**
      return ABORT, d.CommitCert
  else
    return ABORT, d.AbortCert
return COMMIT

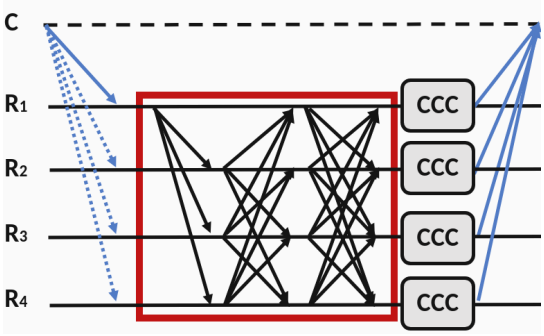

Figure 3: Atomic Broadcast

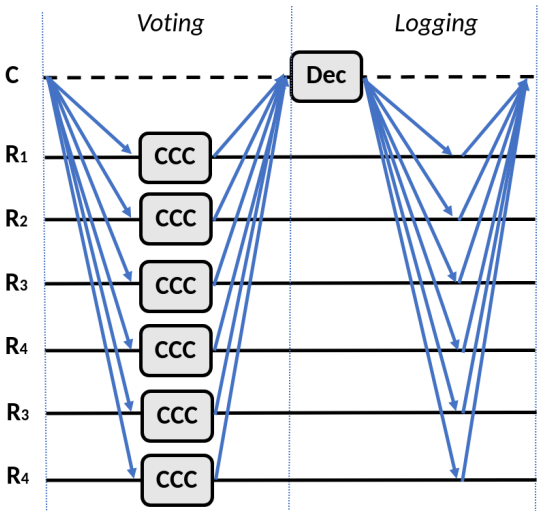

Figure 4: Logging

6.6 Writeback and Multi-shard 2pc

6.7 Failures

- Fallback: election (only starts if not waiting on another dep to avoid early eviction), views, resolution, subtelties with mvtso (block because of dep), necessity even without dependencies. Interested clients, write-back multishard. garbage collection - Fallback requires an extra round in order to learn about current views to start viewchange, but thats ok: Its co-function with learning about full TX, and checking for existing certificates. Timeout invocation is concurrent with p1 message.

6.8 Optimizations

- Retries - single shard logging
  - (Read Locks ; remove can be optimization for writes)
- OCC instead of mvtso structures if disallowing prepared writes to be visible. OCC if not worried about reads aborting

6.9 Garbage Collection

- Prepares get removed upon commit/abort. Eventually replica might need to become "interested" client.

6.10 Indicus3

3f+1 if not defending against byz colluders as much
  - no fast path - Commits in 2 rounds, Aborts in 3 (Alternatively symmetric version) - fallback quorums and bounds - proofs necessary for recovery - recovery rules

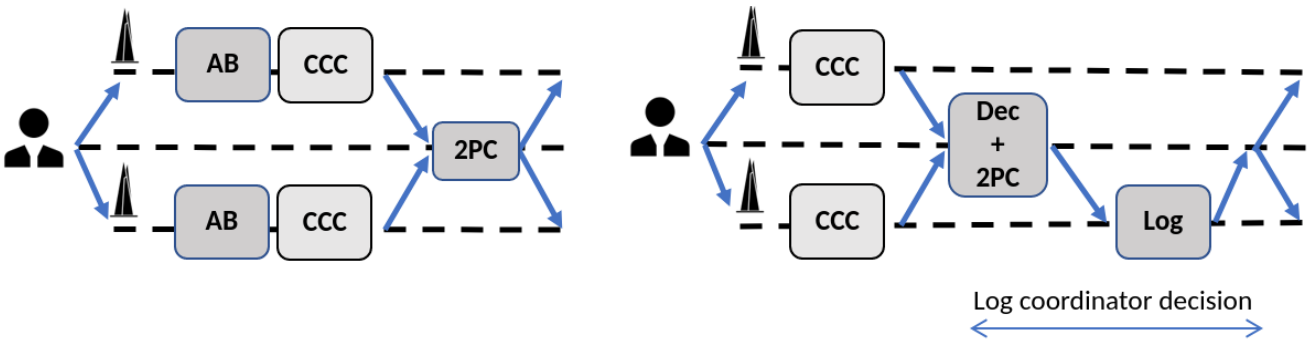

Figure 5: Single Shard Optimization

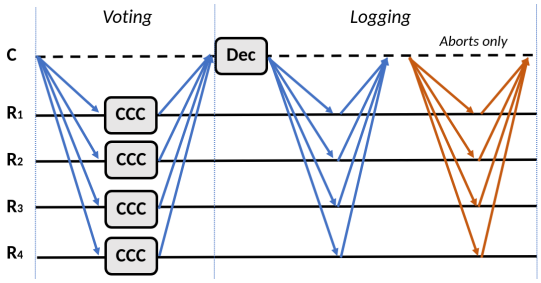

Figure 6: Logging

7 Implementation and Evaluation

7.1 Experiments

-Comparison vs Tapir -Evaluation of the signature and proofs overheads. -(Evaluation of performance under view changes: should show that its not affected too much) -(Comparison against our own system but with Validation doing Atomic Broadcast) -(Comparison against Hyperledger?)

8 Limitations

As is inherent to any Optimistic Execution and Concurrency Control, Indicus is vulnerable to highly congested workloads. When contention on select objects is high, concurrent execution of Transactions must yield the abort of some Transactions during Validation in order to maintain the Database Isolation guarantees. Note however, that when clients are in charge of execution, a pessimistic concurrency control solution such as two-phase-locking would incur an equal amount of deadlocks which would require resolution. The observation to make is that any system that conducts execution at the client application side speculates on concurrency. This however we stipulate, is unavoidable when trying to scale a system to the number of users rather than replica processing power. The traditonal ways to avoid the abort rate conundrum is to either restrict the transaction model, which in turn weakens the general applicability of the protocol, or to delegate execution to replicas and utilize State Machine Replication to serialize Transactions. SMR protocols with a single leader do not inquire any congestion based aborts as a single sequencer naturally eliminates concurrency. Indicus does not make these concessions in order to offer interactive Transactions and remain scalable. A workload that exhibits low commutativity and high contention should therefore refrain from adopting our system.

Similarly, as is the case in any transaction protocol, Indicus is vulnerable to ddos attacks by byzantine participants. A byzantine clients only opportunity at subverting progress for honest users is to artificially increase congestion. When such a client has unrestricted access control it may do so strategically iff it has control over the network. If it does not, it cannot reliably gain knowledge about concurrent transactions before they pass the validation step and must resort to flooding based attacks. Defense against such attacks is out of scope in our work, but is disincentivised as participants can be held accountable for their actions in a closed membership setting.

FS: technical limitations: We require more replicas to avoid certificate signature overheads. Since in 3f+1 one decision needs to be enough to recover. byzantine replicas/clients have more power to vote arbitrarily, because they cannot be held accountable for it. In total order protocols (SMR), deviation from the "common" vote signals misbehavior, whereas for us that is not the case. –> this should go as a challenge somewhere.

FS: Clients are more heavyweight. Not suitable for settings where clients just have minimal processing capacity. Also, Clients need to be registered in system with a sig - necessary to enforce access control in any system however

9 Related Work/Comparison

A lot of recent effort has gone into designing high throughput and low latency databases that leverage synergies between

transaction and replication layer to squeeze out any last performance. The recently proposed TAPIR transaction protocol leverages redundancy between transaction ordering and replication ordering to reduce total roundtrips, thus reducing latency in wide area networks. TAPIR shares several similarities with our system, most notably the absence of a leader and the resulting unordered validation structure. TAPIR too, leverages optimistic concurrency control to allow for concurrency among commutative transactions. When congestion is high however, throughput and tail latency worsen as abort rate grows. Janus avoids transaction aborts by dynamically reordering conflicting transactions. This however, is made possible by assuming one-shot transactions, i.e. fixed read/write keys, and thus reduces the application generality. Another DB, Carousel similarly assumes a restricted Transaction model in order to parallelize execution and validation. While all of these databases offer low latency replication, they are fundamentally limited to tolerating crash-failures. This strong failure assumption makes them less secure and not suitable for storing mission-critical, financial or highly sensitive data. With the surge in Blockchain interest Byzantine Fault Tolerant (BFT) protocols are experiencing a second Spring. While originally developed to tolerate arbitrary bugs, these protocols find increasing importance in settings where participants are untrusted or malicious. Permissioned Blockchains, organized by a consortium of registered participants can use traditional BFT State Machine Replication (SMR) protocols in order to achieve agreement. Starting with PBFT, there have been numerous adaptations such as FaB or Generalized byzantine Paxos, and most notably Zyzzyva which leverages speculative execution and a semi-client driven protocol to reduce latency. SBFT modifies PBFT to scale to large replication degrees by utilizing collectors, threshold signatures and a fast path akin to Zyzzyva. Aardvark states the importance of robustness against byzantine failures and takes measures to increase the rate of leader rotation. Tendermint pushes this idea to the extreme by rotating leaders continuously, at the cost of assuming a synchronous network. HotStuff too, explores the use of rotating leaders by exploiting the symmetry in PBFTs phases. Nevertheless, all protocols derived from the PBFT family suffer from the leader bottleneck as well as enforcing a total order even on commutative operations. BFT-Mir claims the leader proposing speed as the practical bottleneck in most implementations and improves upon this by allowing all replicas to act as proposers. Biblos achieves leaderless SMR by leveraging a non-skipping timestamp protocol. It furthermore allows for commutative Transactions to be executed in parallel at the cost of requiring read/write key sets to be known in advance. In order to preserve liveness however Bilbos falls back to a PBFT resolution. Q/U too, offers leaderless agreement via Qurorums for a limited read/write interface, but fails to terminate under contention. H/Q improves upon Q/U by adding PBFT fallback path under contention. Liskov et Al further explore byzantine Quorum protocols for a Read/Write interface, giving special attention to bounding the effects of byzantine clients. While the literature on BFT state machine replication is extensive, the efforts to offer a transactional interface for BFT is scarce. SMR itself can be utilized as a straw-man system to implement pre-defined Transactions (one-shot or stored procedures) by enforcing a common total order in which replicas will execute the transactions. HRDB offers a dedicated BFT Database, but assumes a trusted shepheard layer, thus not being truly BF resilient Byantium offers Snapshot Isolation for Transactions by repurposing PBFT as atomic broadcast. It executes requests only at a primary and uses replicas to validate results in the total order defined by the SMR protocol. Augustus implements mini-transactions, a limited TX model that declares all operations before execution. It offers scalability via partitioning and achieves consistency within partitions by utilizing atomic broadcast. Whiile Augustus assumes an optimistic execution model that allows for aborts under concurrency, its follow up work Callinicos implements a locking scheme in order to avoid Transaction aborts. To do so efficiently it resorts to a limited transaction model that requires knowledge of read/write sets and otherwise locks an entire partition in order to guarantee mutual exclusion, thus voiding any concurrency. BFT Deferred update replication adopts an interactive OCC transaction model comparable to ours, allowing execution to be speculative at clients. However, it uses PBFT atomic broadcast to enforce SMR on validation, thus resorting to a leader, and totally ordering all requests. It moreover does not extend to multiple shards. Chainspace and Omniledger implement blockchain sharding by layering 2PC and atomic broadcast (within shards) for UTXO transactions. Rapidchain offers efficient sharding for a permissionless system (what else? didnt read much, not so relevant). Hyperledger.. Mention other Permissioned Blockchain systems: Ethereum Quorum? [**?**]

10 Conclusion

Acknowledgments

The USENIX latex style is old and very tired, which is why there's no \acks command for you to use when acknowledging. Sorry.

Availability

USENIX program committees give extra points to submissions that are backed by artifacts that are publicly available. If you made your code or data available, it's worth mentioning this fact in a dedicated section.

References

[1] Apple. Apple supplier list, 2019.

[2] TPCW consortium. Tpc benchmark-w specification v. 1.8. http://www.tpc.org/tpcw/spec/tpcw_v1.8.pdf, 2002.