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

There is an increasing tension between the desire to share data online, and the security concerns it entails. This paper asks the question: how can we enable *mutually distrustful parties* to consistently and reliably share data, while minimizing centralization?

The ability to share data online offers exciting opportunities. In banking, systems like SWIFT enable financial institutions to quickly and accurately receive information such as money transfer instructions; and in manufacturing, online data sharing can improve accountability and auditing amongst the globally distributed supply chain.

Increased data sharing, however, raises questions of how to *decentralize trust*. Banking institutions must currently place their trust in the centralized SWIFT's network to issue payment orders. Sometimes there is even no identifiable source of trust. Consider the supply chain for the latest iPhone: it spans three continents, and hundreds of different contractors []; neither Apple nor these contractors trust each other, yet all must be willing to agree and share information about the construction of the same product.

Recognizing this challenge by both the research and industry 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 log; the log is agreed upon by the n participants in the system, of which at most f can misbehave. Each participant executes operations that may touch one to multiple objects in the log. In the blockchain world, Bitcoin and Ethureum have become popular distributed computing platforms providing the same log abstraction and aiming for decentralizing trust. Furthermore, Microsoft Azure has launched projects[] that leverage these blockchain platforms and extend the digital transformation beyond the companies' four walls in a supply chain.

This paper argues that there exists a fundamental mismatch between the implementation of a totally ordered log and the reality of much large-scale distributed processing. Many large-scale distributed systems consist 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 unnecessary, but costly: participants in the shared computation must vote to order operations, store the full state of the system, and replay the full log for auditing.

While there exists work on mitigating this scalability bottleneck through sharding [], the latent total order requirement introduces unnecessary coordination overhead, as coordination is performed twice, at the level of individual shards, and across shards. Callinicos [] and Omniledger [], for instance, runs a full BFT protocol for every operation. This is especially problematic when workloads are geo-replicated [], or when, as in BFT, the replication factor is high. Further, 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.

As another research trend of mitigating the scalability bottleneck, EPaxos[], TAPIR[] and CURP[] only consider the ordering between potentially conflicting operations, instead of commutative operations. However, these systems assume the crash-failure model and are non-trivial to be extended to the Byzantine model, so that they cannot directly solve the problem of data sharing among mutually distrustful 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 functionality 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.*

2 Introduction

Looking beyond treating transactions as black box requests and leveraging existing commutativity is not a new idea. However, while copious efforts exist to design and build more decentralized systems in order to exploit transaction knowledge for the Crash Failure model, few, if any attempts have been made for the Byzantine Fault Model. This naturally begs the question why so? One explanation is that for BFT systems centralization is in fact highly desirable as it simplifies the problem by pin-pointing a single point of accountability. Since historically, BFT systems have taken a rather niche role for applications that require "additional" safety, the principal design concern has always been maintaining consistency with efficiency and scalability being secondary concerns. Replication was traditionally done by a single authority and thus it is reasonable to assume a primary backup scheme, since neither fairness, nor total ordering were major concerns. As Byzantine Fault Tolerance moves into the mainstream with the popularity ascent of Blockchains these considerations are being revistited. For example, when operating a database as a consortium of mutually distrustful parties it is no longer desirable to grant priviledges to a leader and impose centralization for the sake of simplicity. Rather than trying to make a traditional BFT system more scalable we ask the question whether we can take the scalability lessons from Crash Failure settings and improve upon their robustness. While this is an attractive avenue, it is far from being straightforward. A natural way to scale a system is to move state and responsabilites to the clients. However, in a byzantine setting, giving up the comfort of a bounded and accountable set of replicas opens up the system to a wider set undesirable phenomena. Yet, this is exactly what we will do: Concretely, in this paper we will show how to design a BFT system named Indicus that is almost entirely client driven, and all the while both safe and live.

Overview:
Unlike State Machine Replication (SMR) based systems that achieve agreement on computation (i.e. state transitions or request results) by imposing a total order for execution, Indicus evaluates results out of order. Reducing the problem of agreement to a sequencing problem simplifies SMR design, yet squanders available commutativity between requests. Agreement on a total order strengthens the requirment of the system, when potentially unecessary. Concretely, agreement on a totally ordered ledger implies agreement for the entire history prefix for each new request. While desirable in some cases (a fundamental principle of Blockchains), this is unnecessary if any given request is commutative to any other. In fact, in this extreme case, all requests could reach agreement in parralel and out of order. In practice, a partial order, that only orders non-commutative requests suffices. To leverage this, Indicus performs agreement for each transaction seperately, and imposing only an implicit order when conflicts arrive. Abstractly, Indicus proposes each transaction for a single-shot binary consensus, i.e. transactions do not compete for shared slots. In order to maintain a serializable Transaction history, Indicus follows a standard Optimistic Concurrency Control technique called Timestamp Ordering (TSO): Indicus pre-defines a suggestion for a total order by assigning a Timestamp to each Transaction. Ordering conflicts between Transactions whose interleavings would violate prescribed Isolation guarantees are broken based on the given Timestamp and speculative execution results.

Overall, the Indicus design has the following positive outcomes:

- Indicus is robust to censorship and frontrunning as there exist no central authority (in SMR traditionally a leader) that decides what Transactions enter the system and decides on the ordering of Transactions
- Indicus allows commutative/non-conflicting Transactions to both be executed and validated out of order, thus maximizing parallelism. (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. (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 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

Since we do not sequence Transaction execution we require a concurrency control mechanism in order to maintain Isolation between Transactions. For this purpose, we design and implement a byzantine replicated MVTSO scheme, an aggressive version of Optimistic Concurrency control that empowers Read Transactions in the hope of reducing abort rates. Specifically, we allow to read old version, we allow to read

uncommitted writes and we acquire read leases.

Additionally, we propose definitions for what it means to enforce an Isolation level in a transactional system with byzantine participants. These are general purpose formalizations and can serve as guideline for future byzantine Database systems.

In section X, we discuss Limitations of this approach.

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.

3.1 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 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.

3.2 Model and Definitions

3.2.1 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 retraced by auditing the Transaction logs.

3.2.2 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

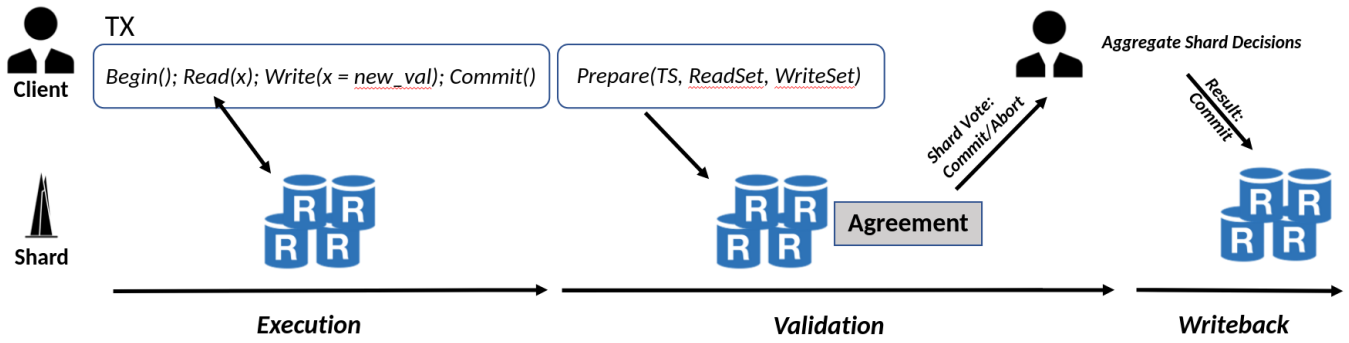

Figure 1: Transaction Lifecycle

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

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

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

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

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.

3.3 TEsting

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

3.4 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.

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 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 2f+1 then its effectively a read lock. 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 Add exceptions etc.)
3. **Commit** A Client finalizes its execution, computes the final Timestamp and submits
4. **Abort**

- 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.

3.5 Validation

3.6 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

Algorithm 1 MVTSO-Check(TX, TS)

1: **if** $TS > localClock + \delta$ **then**
2: pass
3: **end if**
4: **for** $\forall key, version \in TX.read\text{-}set$ **do**
5: **if** $version < Store[key].max\text{-}version < TS$ **then**
6: **return** ABORT, $Store[key].certificate$
7: **end if**
8: **if** $TS > MIN(prepared\text{-}writes[key] > version$ **then**
9: **return** ABSTAIN
10: **end if**
11: **end for**
12: **for** $\forall key \in TX.write\text{-}set$ **do**
13: **if** $\exists TX2 \in Prepared/CommitLog \wedge TX \notin TX2.dep$: $TX2.read\text{-}set[key].version < TS < TX2.TS$ **then**
14: **return** RETRY,
15: **end if**
16: **if** $\exists RTS \in key.RTS : RTS > TS \wedge TX \notin RTS.dep$ **then**
17: **return** RETRY,
18: **end if**
19: **end for**
20: Prepared.add(TX)
21: **while** $\exists d \in dep : d \notin CommitLog \cup AbortLog$) **do**
22: Wait
23: **end while**
24: **for** $\forall d \in dep$ **do**
25: **if** $d \in CommitLog$ **then**
26: **if** $d.TS > TS$ **then**
27: Abort
28: **end if**
29: **else**
30: Abort
31: **end if**
32: **end for**
33: **return** COMMIT

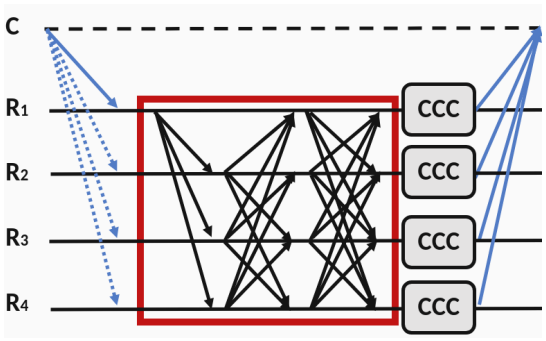

Figure 3: Atomic Broadcast

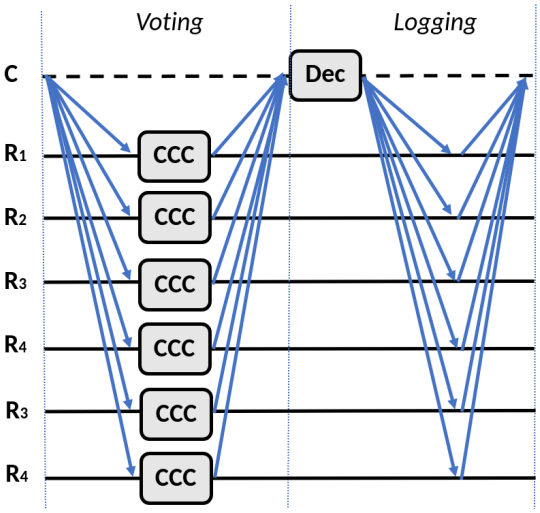

Figure 4: Logging

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) -

3.7 Consistent logging.

Principles and challenges

protocol overview: pic

3.8 Writeback and Multi-shard 2pc

3.9 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.

3.10 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

3.11 Garbage Collection

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

3.12 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

4 Implementation and Evaluation

5 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.

6 Related Work/Comparison

mention Hotstuff? 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 re-ordering 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. 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

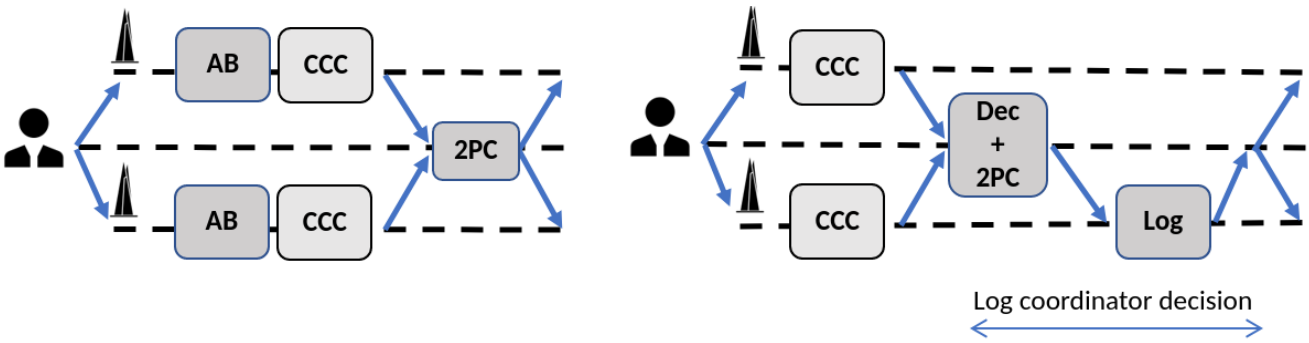

Figure 5: Single Shard Optimization

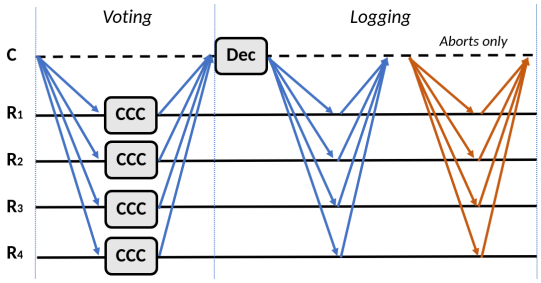

Figure 6: Logging

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 re-purposing 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.

[1]

7 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] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. *ACM SIGOPS Operating Systems Review*, 41(6):45–58, 2007.