Resilient and Scalable Architecture for Permissioned Blockchain Fabrics

Suyash Gupta
Exploratory Systems Lab
Department of Computer Science
University of California, Davis
Supervised by Mohammad Sadoghi

ABSTRACT

Since the introduction of Bitcoin—the first widespread application driven by blockchains—the interest in the design of blockchain-based applications has increased tremendously. At the core of these blockchain applications are consensus protocols that aim at securely replicating a client request among all replicas, even if some replicas are Byzantine faulty. Unfortunately, modern consensus protocols either yield low throughput or face design limitations.

In this work, we present the design of three consensus protocols that facilitate efficient consensus among the replicas. Our protocols help to scale consensus through the principles of phase-reduction, parallelization, and geo-scale clustering while ensuring no compromise in fault-tolerance. Further, we believe that the focus on consensus protocols is only one-side of the story. In specific, we present the design of a well-crafted permissioned blockchain fabric that can help even a slow consensus protocol outperform a faster protocol.

PVLDB Reference Format:
. . *PVLDB*, (): xxxx-yyyy, .
DOI:

1. INTRODUCTION

Since the introduction of *Bitcoin* the interest in the design of blockchain-based applications has increased tremendously [13, 17]. Blockchain-based solutions have garnered community interest as they guarantee *democracy* and *decentralization*. To exploit these guarantees, in recent years, several new blockchain databases and fabrics have been proposed [1, 2, 11, 18]. These blockchain databases achieve decentralization by employing age-old *replication* semantics and ensure democracy by running a *fault-tolerant consensus* protocol. At the core of any blockchain application is a BFT consensus protocol that ensures all replicas of this blockchain application reach *consensus* on the ordering of incoming client requests, this even if some of the replicas are byzantine [4, 9, 11, 16, 21].

Proceedings of the VLDB Endowment, Vol. , No.
ISSN 2150-8097.
DOI:

On looking closely, we find an unusual trend: even after a decade of its introduction and after several prominent research projects, crypto-currencies are still the major *known use-cases* of blockchains. This raises a key question: *Why have blockchain applications seen such a slow wider adoption?* The low throughput and high latency of BFT consensus are cited as key reasons for this. Prior works [14, 19] have shown that traditional distributed databases can achieve throughputs of the order $100K$ transactions per second, while the initial *permissionless blockchain applications*, Bitcoin [17] and Ethereum [20], reach throughputs of only a few transactions per second. In these crypto-currency applications, low throughputs are seen as acceptable, as the techniques used enable an alternative currency that is unregulated by any government or corporation. Notice that these crypto-currency blockchains support *open-membership*, as anyone can anonymously join these blockchains.

Recent attacks on permissionless blockchains have highlighted that their open-membership is often unnecessary and undesirable [7]. This led to industry-grade *permissioned* designs, where only a select group of users, some of which may be untrusted, can participate [2]. A majority of these permissioned designs employ the classical Pbft protocol and achieve throughputs of up-to $10K$ transactions per second [1, 2], which is still short of the performance expected of modern systems. This isn't surprising because Pbft achieves consensus in *three* phases, of which two necessitate quadratic communication complexity. Moreover, since the introduction of Pbft, several new BFT protocols have been proposed [6, 16, 21]. We believe the inherent limitations of these protocols as a logical consequence for existing blockchain databases to skip these protocols.

In this work, we present the design of *three* new BFT protocols that yield high-throughput while providing the same fault-tolerance guarantees as the Pbft protocol. First, we employ *speculation* and facilitate *out-of-order processing* of messages to achieve consensus in just two phases. Second, we scale up the Pbft protocol by allowing *multiple consensuses* to happen in *parallel*. Third, we scale out the Pbft protocol by *clustering* replicas, which permits consensus on a *global scale*. Further, we also observe that the low throughputs of existing permissioned fabrics are due to missed opportunities during their design and implementation. Hence, we present the design of a well-crafted blockchain fabric that can achieve an order-of-magnitude increase in throughput by exploiting parallelization and pipelining opportunities.

2. BYZANTINE FAULT-TOLERANCE

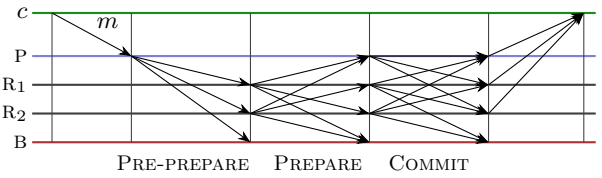

Figure 1: The three-phase PBFT protocol.

Pbft [4] is often described as the first BFT protocol to allow consensus to be incorporated by practical systems. Pbft follows the primary-backup model where one replica is designated as the *primary* while other replicas act as backups. Pbft guarantees a successful consensus among n replicas if at most f of them are byzantine, where $n \geq 3f + 1$.

When the primary replica receives a client request, it assigns it a sequence number and sends a Pre-prepare message to all the backups to execute this request in the sequence order (refer to Figure 1). Each backup replica on receiving the Pre-prepare message from the primary shows its agreement to this order by broadcasting a Prepare message. When a replica receives Prepare message from at least $2f$ distinct backup replicas, then it achieves a guarantee that a *majority* of the non-faulty replicas are aware of this request. Such a replica marks itself as *prepared* and broadcasts a Commit message. Next, when this replica receives Commit messages from $2f + 1$ distinct replicas, then it achieves a guarantee for the order of this request, as a majority of the replicas must have also prepared this request. Finally, this replica executes the request and sends a response to the client.

Multi-Path Execution. Zyzzyva [16] introduces a multi-path protocol to achieve efficient consensus. In the *fast-path*, the consensus happens in a single linear phase. When a backup replica receives a Pre-Prepare message from the primary, it executes the request and sends a response to the client. Hence, a replica does not even wait to confirm that the order is the same across all the replicas. However, if the primary is malicious, Zyzzyva switches to the *slow-path*. In specific, for the fast-path to be successful, the client waits for responses from all the replicas. If the client *timeouts* while waiting for responses, it switches to the slow-path. Hence, the fast-path of Zyzzyva cannot handle even one simple failure. Note that Zyzzyva depends on *good clients* to ensure correct order if the primary is malicious. Moreover, a recent work has uncovered a flaw in Zyzzyva's design [6].

Sbft [6] removes the flaw from Zyzzyva's design but requires two linear phases in its fast-path and an additional third phase in its slow-path. Moreover, in all the protocols we have discussed until now, if the primary is malicious, then it is replaced. This replacement requires detecting the faulty primary and exchanging correct states among the replicas. Hotstuff [21] suggests switching primary after each consensus. However, this comes at a significant cost; Hotstuff requires sequential consensus processing. In Hotstuff, each subsequent primary needs to wait for messages from a quorum of replicas before starting the next consensus. This negatively impacts the system throughput as messages can no longer be processed out-of-order.

3. PROOF-OF-EXECUTION

Our first step is to concoct a fast yet reliable consensus protocol. We call this protocol *Proof-of-Execution* (PoE) [8],

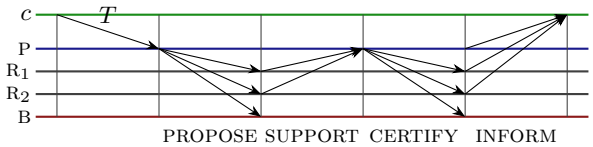

Figure 2: Normal-case algorithm of PoE: Client c sends its request containing transaction T to the primary P, which proposes this request to all replicas. Although replica B is Byzantine, it fails to affect PoE.

which employs *three* ingredients for ensuring efficient consensus. First, PoE prevents use of any multi-path design as switching from fast to slow path requires dependence on timeouts, which degrades system performance [5]. Second, PoE allows replicas to speculatively execute the requests but facilitates rollbacks in case of inconsistencies. Final, PoE allows out-of-order processing, which eliminates any bottlenecks associated with sequential consensus protocols.

In Figure 2, we sketch the normal-case working of PoE. As no one size fits all systems, we believe the design of a BFT protocol should be independent of the choice of *underlying* cryptographic signature scheme. Hence, PoE can adapt itself to both symmetric and asymmetric-cryptographic signature schemes [15]. For the sake of brevity, we will describe PoE built on top of Threshold Signatures `TS`.

The consensus steps of the PoE protocol are sketched in Figure 2. In PoE, the client c initiates execution by sending its request m to the primary P. To initiate replication and execution of m as the k-th transaction, the primary proposes m to all replicas by broadcasting a Propose message.

After a replica R receives a Propose message from P, it checks whether at least $2f$ other replicas also received the same proposal from P. To perform this check, each replica agrees to *support* the first k-th proposal it receives from the primary by sending a Support message that includes its unique *threshold share* to the primary. The primary P waits for $2f + 1$ threshold shares, and on receiving such shares, it combines them into a *threshold signature* and broadcasts as a Certify message. When a replica R receives the Certify message, it *view-commits* to m as the k-th transaction in view v. After R view-commits to m, R schedules T for speculative execution. Consequently, m will be executed by R after all preceding transactions are executed. After execution, R informs the client of the order of execution and of any execution result r. A client considers its transaction successfully executed after it receives identical response messages from $2f + 1$ distinct replicas.

4. PARALLEL CONSENSUS

Until now, all the BFT protocols that we studied followed a primary-backup model. This dependence on the primary severely affects the throughput and scalability of these protocols. The primary replica not only receives all client requests but is also responsible for ensuring consensus is reached on the order for these requests among all other replicas. If the primary *fails* to ensure consensus, then all remaining replicas need to *replace* this primary. This replacement process is necessary as, without it, non-faulty replicas may never converge. Unfortunately, primary replacement is not cheap, as it requires pausing consensus on all outstanding requests until the primary is replaced.

A promising solution to all these problems is to make a BFT consensus primary agnostic. Such a solution would

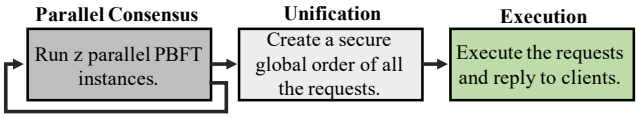

Figure 3: Three stages of the MultiBFT protocol.

require us to give all replicas the power to act as a primary. This brings us to the design of our *Multiple Byzantine Fault-Tolerance* (MultiBFT) paradigm [9, 10]. MultiBFT *parallelizes the consensus* by requiring each replica to run *multiple instances* of the Pbft protocol in parallel.

Using parallelization, MultiBFT ensures that the non-faulty replicas are *always accepting and ordering client requests*, this independent of any malicious behavior or attack. Figure 3 illustrates a succinct representation of our MultiBFT paradigm. For the sake of explanation, we assume MultiBFT works in *rounds*. Each *round* of MultiBFT includes *three stages*: *parallel consensus*, *unification*, and *execution*. The notion of a *round* helps in generating a common order and recovering from instance failures but it does not prevent individual primaries from working independently.

Prior to any round, MultiBFT requires each replica to prepare to run z instances of Pbft protocol in parallel. A round r begins when the primary of each instance proposes a client request. Firstly, in the *parallel consensus* stage, each instance runs Pbft on its client request. Secondly, in the *unification* stage, the replica waits for all its z instances to complete replication (reach consensus on their respective requests). If every instance successfully replicates a request, then a common order for execution of these requests is determined. If one or more instances are *unable to replicate* requests, then the primaries for those instances must be faulty and recovery is initiated. Finally, in the *execution* stage, each replica executes all the client requests in the common order. Notice that in Figure 3, we have a *loop*. This loop states that while unification and execution are ongoing for requests of round r, the instances are already replicating requests for round $r + 1$.

5. GEO-SCALE CONSENSUS

To enable geo-scale deployment of a permissioned blockchain system, we believe that the underlying consensus protocol must distinguish between *local* and *global* communication. To resolve this challenge, we present our *Geo-Scale Byzantine Fault-Tolerant* consensus protocol (GeoBFT) [11] that uses topological information to group all replicas in a single region into a single cluster. Likewise, GeoBFT assigns each client to a single cluster. This clustering helps in attaining high throughput and scalability in geo-scale deployments. GeoBFT operates in rounds, and in each round, every cluster will be able to propose a single client request for execution. Each round consists of the three steps sketched in Figure 4: *local replication*, *global sharing*, and *ordering and execution*, which we further detail next.

At the start of each round, each cluster chooses a single transaction of a local client. Next, each cluster *locally replicates* its chosen transaction in a Byzantine fault-tolerant manner using Pbft. At the end of successful local replication, Pbft guarantees that each non-faulty replica can prove successful local replication via a *commit certificate*.

Next, each cluster shares the locally-replicated transaction along with its commit certificate with all other clusters. To minimize inter-cluster communication, we use a novel *optimistic global sharing protocol*. Our optimistic global sharing protocol has a global phase in which clusters exchange locally-replicated transactions, followed by a local phase in which clusters distribute any received transactions locally among all local replicas. Finally, after receiving all transactions that are locally-replicated in other clusters, each replica in each cluster can deterministically *order* all these transactions and proceed with their *execution*. After execution, the replicas in each cluster inform only local clients of the outcome of the execution of their transactions (e.g., confirm execution or return any execution results).

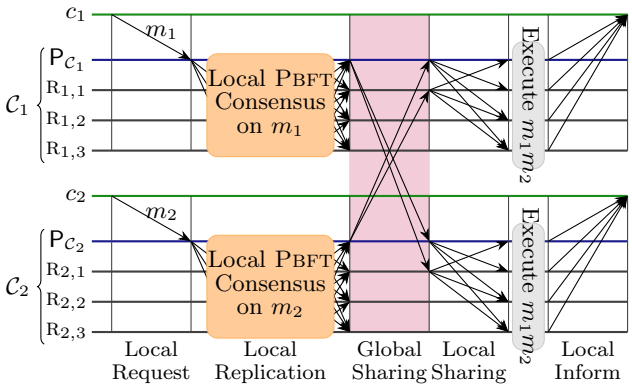

Figure 4: Representation of the GeoBFT protocol running on two distinct clusters.

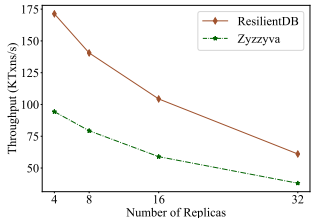

Figure 5: Two permissioned applications employing distinct BFT protocols (80K clients per experiment).

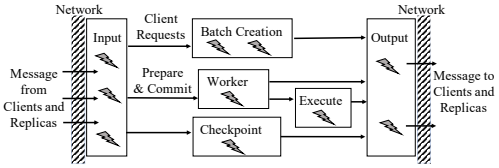

Figure 6: Pipeline at ResilientDB replicas.

6. RESILIENTDB

Although efficient consensus can help increase the throughput of a permissioned blockchain application, we believe this philosophy only reflects a one-sided story. In specific, the design and implementation of the underlying system also plays a major role in determining its scalability.

We use Figure 5 to illustrate such a possibility. In this figure, we measure the throughput of our optimally designed permissioned blockchain system ResilientDB [11, 12] against a *protocol centric* permissioned blockchain system that adopts practices suggested in BFTSmart [3]. We intentionally make ResilientDB adopt the slow Pbft protocol while the other system employs the fast Zyzzyva protocol. Despite this, ResilientDB achieves a throughput of 175K transactions per second, scales up to 32 replicas, and attains up to 79% more throughput.

ResilientDB lays down an efficient client-server architecture. At the *application layer*, we allow multiple clients to co-exist, each of which creates its own requests. For this purpose, clients can either employ an existing benchmark suite or design a *Smart Contract* suiting to the active application. Next, clients and replicas use the *transport layer* to exchange messages across the network. ResilientDB also provides a *storage layer* where all the metadata corresponding to a request and the blockchain is stored. At each replica, there is an *execution layer* where the underlying consensus protocol is run on the client request, and the request is executed.

ResilientDB includes multi-threaded deep pipelines that allow it to achieve high-throughput consensus among its replicas (refer to Figure 6). We permit increasing (or decreasing) the number of threads of each type. With each replica, we associate multiple *input* and *output* threads to communicate with the network. ResilientDB also associates multiple *batch-threads* with the primary replica to batch client requests. Using an optimal batching policy can help mask consensus costs. To process other messages and run the phases of an underlying consensus protocol, ResilientDB also includes several *worker-threads*. Once a request is replicated, each replica asks its *execute-thread* to execute the request and reply to the client. Depending on the underlying protocol, we also allow threads for checkpointing, garbage collection, and certificate exchange.

7. CONCLUSIONS

In this work, we present design of three efficient BFT protocols that aim to reduce the costs associated with BFT consensus. Our protocols present mechanisms to reliably reduce phases of Pbft protocol, parallelize Pbft protocol and scale Pbft to global setting by clustering replicas. Moreover, these protocols can be easily combined to yield a single high-throughput consensus protocol. Further, we illustrate that the design and implementation of the underlying permissioned blockchain fabric is equally important.

8. REFERENCES

[1] M. J. Amiri, D. Agrawal, and A. E. Abbadi. CAPER: A cross-application permissioned blockchain. *Proceedings of the VLDB Endowment*, 12(11):1385–1398, 2019.

[2] E. Androulaki et al. Hyperledger Fabric: A distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, pages 30:1–30:15. ACM, 2018.

[3] A. Bessani, J. Sousa, and E. E. P. Alchieri. State machine replication for the masses with bft-smart. In *DSN*, 2014.

[4] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, pages 173–186. USENIX Association, 1999.

[5] A. Clement, E. Wong, L. Alvisi, M. Dahlin, and M. Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI, pages 153–168. USENIX Association, 2009.

[6] G. Golan Gueta, I. Abraham, S. Grossman, D. Malkhi, B. Pinkas, M. Reiter, D. Seredinschi, O. Tamir, and A. Tomescu. Sbft: A scalable and decentralized trust infrastructure. In *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 568–580, 2019.

[7] S. Gupta, J. Hellings, S. Rahnama, and M. Sadoghi. An in-depth look of BFT consensus in blockchain: Challenges and opportunities. In *Proceedings of the 20th International Middleware Conference Tutorials*, pages 6–10. ACM, 2019.

[8] S. Gupta, J. Hellings, S. Rahnama, and M. Sadoghi. Proof-of-Execution: Reaching Consensus through Fault-Tolerant Speculation. *CoRR*, abs/1911.00838, 2019.

[9] S. Gupta, J. Hellings, and M. Sadoghi. Brief announcement: Revisiting consensus protocols through wait-free parallelization. In *33rd International Symposium on Distributed Computing (DISC 2019)*, volume 146, pages 44:1–44:3, 2019.

[10] S. Gupta, J. Hellings, and M. Sadoghi. Scaling blockchain databases through parallel resilient consensus paradigm. *CoRR*, abs/1911.00837, 2019.

[11] S. Gupta, S. Rahnama, J. Hellings, and M. Sadoghi. Resilientdb: Global scale resilient blockchain fabric. *PVLDB*, 13(6):868–883, 2020.

[12] S. Gupta, S. Rahnama, and M. Sadoghi. Permissioned blockchain through the looking glass: Architectural and implementation lessons learned. In *40th IEEE International Conference on Distributed Computing Systems*, 2020.

[13] S. Gupta and M. Sadoghi. *Blockchain Transaction Processing*, pages 1–11. Springer International Publishing, 2018.

[14] S. Gupta and M. Sadoghi. EasyCommit: A non-blocking two-phase commit protocol. In *Proceedings of the 21st International Conference on Extending Database Technology*, pages 157–168. Open Proceedings, 2018.

[15] J. Katz and Y. Lindell. *Introduction to Modern Cryptography*. Chapman and Hall/CRC, 2nd edition, 2014.

[16] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative byzantine fault tolerance. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, pages 45–58. ACM, 2007.

[17] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2009.

[18] F. Nawab and M. Sadoghi. Blockplane: A global-scale byzantizing middleware. In *35th International Conference on Data Engineering (ICDE)*, pages 124–135. IEEE, 2019.

[19] T. Qadah, S. Gupta, and M. Sadoghi. Q-store: Distributed, multi-partition transactions via queue-oriented execution and communication. In *Proceedings of the 23nd International Conference on Extending Database Technology*, pages 73–84. OpenProceedings.org, 2020.

[20] G. Wood. Ethereum: a secure decentralised generalised transaction ledger, 2016. EIP-150 revision.

[21] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham. HotStuff: BFT consensus with linearity and responsiveness. PODC, 2019.