

Permissioned Blockchain Through the Looking Glass: Architectural and Implementation Lessons Learned

Suyash Gupta* Sajjad Rahnama* Mohammad Sadoghi
MokaBlox LLC.
Exploratory Systems Lab
University of California, Davis

Abstract—Since the inception of Bitcoin, the distributed systems community has shown interest in the design of efficient blockchain systems. However, initial blockchain applications (like Bitcoin) attain very low throughput, which has promoted the design of *permissioned* blockchain systems. These permissioned blockchain systems employ classical Byzantine-Fault Tolerant (BFT) protocols to reach consensus. However, existing permissioned blockchain systems still attain low throughputs (of the order 10K txns/s). As a result, existing works blame this low throughput on the associated BFT protocol and expend resources in developing optimized protocols. We believe such blames only depict a one-sided story. In specific, we raise a simple question: *can a well-crafted system based on a classical BFT protocol outperform a modern protocol?* We show that designing such a well-crafted system is possible and illustrate that even if such a system employs a three-phase protocol, it can outperform another system utilizing a single-phase protocol. This endeavor requires us to dissect a permissioned blockchain system and highlight different factors that affect its performance. Based on our insights, we present the design of our enterprise-grade, high-throughput yielding permissioned blockchain system, ResilientDB, that employs multi-threaded deep pipelines, to balance tasks at a replica, and provides guidelines for future designs.

I. INTRODUCTION

Since the inception of *blockchain* [1], [2], the distributed systems community has renewed its interest in the age-old design of Byzantine-Fault Tolerant (BFT) systems. At the core of any blockchain applications is a BFT algorithm that ensures all the replicas of this blockchain application reach a *consensus*, that is, agree on the order for a given client request, even if some of the replicas are byzantine [3], [4], [5], [6], [7].

Surprisingly, even after a decade of its introduction and publication of several prominent research works, the major use-case of blockchain technology remains as a crypto-currency. This leads us to a key observation: *Why have blockchain (or BFT) applications seen such a slow adoption?*

The low throughput and high latency are the key reasons why BFT algorithms are often ignored. Prior works [8], [9], [10], [11] have shown that the traditional distributed systems can achieve throughputs of the order 100K transactions per second while the initial blockchain applications, such as Bitcoin [12] and Ethereum [13], have throughputs of at most ten transactions per second. Such low throughputs do not

affect the users of these applications, as the aim of these applications is to promote an alternative currency, which is unregulated by any large corporation, that is, anyone can join, and the identities of the participants are kept hidden (*open membership*). Evidently, this open-membership property has also led to several attacks on these applications [2], [14], [15].

This led to industry-grade *permissioned* blockchain systems, where only a select group of users, some of which may be untrusted, can participate [7]. However, the throughputs of current permissioned blockchain applications are still of the order 10K transactions per second [7], [16], [17]. Several prior works blame the low throughput and scalability of a permissioned blockchain system on to its underlying BFT consensus algorithm [2], [4], [18], [17]. Although these claims are not false, we believe they only represent a one-sided story.

We claim that the low throughput of a blockchain system is due to missed opportunities during its design and implementation. Hence, we want to raise a question: *can a well-crafted system-centric architecture based on a classical BFT protocol outperform a protocol-centric architecture?* Essentially, we wish to show that even a slow-perceived classical BFT protocol, such as PBFT [3], if implemented on skillfully-optimized blockchain fabric, can outperform a fast niche-case and optimized for fault-free consensus, BFT protocol, such as Zyzyva [4]. We use Figure 1 to illustrate such a possibility. In this figure, we measure the throughput of an optimally designed permissioned blockchain system (ResilientDB) and intentionally make it employ the slow PBFT protocol. Next, we compare the throughput of ResilientDB against a *protocol-centric* permissioned blockchain system that adopts practices suggested in BFTSmart [19] and employs the fast Zyzyva protocol. We observe that the *system-centric* design of ResilientDB, even after employing the three-phase PBFT protocol (two of the three phases require quadratic communication among the replicas) outperforms the system having a single-phase linear protocol Zyzyva. Further, ResilientDB achieves a throughput of 175K transactions per second, scales up to 32 replicas, and attains up to 79% more throughput.

This paper is aimed at illustrating that the design and architecture of the underlying system are as important as optimizing BFT consensus. Decades of academic research and industry experience has helped the community in designing efficient

*Both authors have equally contributed to this work.

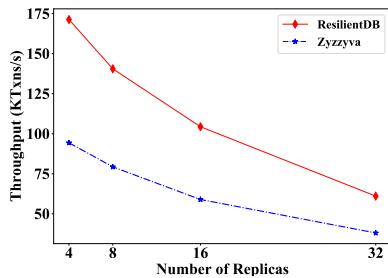


Fig. 1: Two permissioned applications employing distinct BFT consensus protocols (80K clients used for each experiment).

distributed applications [20], [21], [22], [23]. We use these principles to illustrate the design of a high-throughput yielding permissioned blockchain fabric, ResilientDB. In specific, we *dissect* existing permissioned blockchain systems, identify different performance bottlenecks, and illustrate mechanisms to eliminate these bottlenecks from the design. For example, we show that even for a blockchain system, ordering of transactions can be easily relaxed without affecting the security. Further, most of the tasks associated with transaction ordering can be extensively parallelized and pipelined. A highlight of our other observations:

- Optimal batching of transactions can help a system gain up to $66\times$ throughput.
- Clever use of cryptographic signature schemes can increase throughput by $103\times$.
- Employing in-memory storage with blockchains can yield up to $18\times$ throughput gains.
- Decoupling execution from the ordering of client transactions can increase throughput by 10%.
- Out-of-order processing of client transactions can help gain 60% more throughput.
- Protocols optimized for fault-free cases can result in a loss of $39\times$ throughput under failures.

These observations allow us to perceive ResilientDB as a reliable test-bed to implement and evaluate enterprise-grade blockchain applications.¹ We now enlist our contributions:

- We dissect a permissioned blockchain system and enlist different factors that affect its performance.
- We carefully measure the impact of these factors and present ways to mitigate the effects of these factors.
- We design a permissioned blockchain system, ResilientDB that yields high throughput, incurs low latency, and scales even a slow protocol like PBFT. ResilientDB includes an extensively parallelized and deeply pipelined architecture that efficiently balances the load at a replica.
- We raise *eleven* questions and rigorously evaluate our ResilientDB platform in light of these questions.

Note on this work: This paper is not aimed at designing efficient BFT consensus protocols, for which there already exists an extensive literature [3], [4], [24], [25], [26]. Further,

¹ ResilientDB is available and open-sourced at <https://resilientdb.com>.

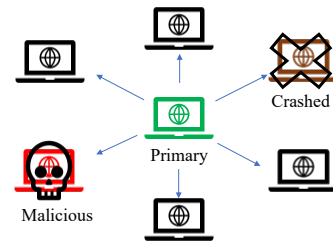


Fig. 2: This diagram illustrates a set of replicas of which some may be malicious or have crashed. One replica is designated as the primary, which leads the consensus on the received client request among the backup replicas.

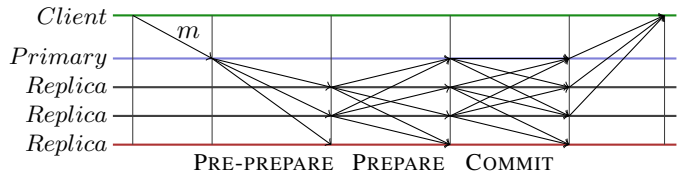


Fig. 3: The three-phase PBFT protocol.

this work does not aim at benchmarking open-membership and permissioned blockchain systems, as done by Blockbench [2]. Moreover, we do not advocate the use of any specific BFT protocol or permissioned blockchain system, but instead perform an in-depth analysis of a single permissioned blockchain system, to uncover insights that can help both researchers and practitioners to build next-generation blockchain fabrics.

II. BACKGROUND AND RELATED WORK

Before laying down the foundation for efficient design, we first analyze existing literature and practices in the domain of permissioned blockchain.

A. BFT Consensus

At the core of any blockchain application is a BFT consensus protocol, which states that given a client request and a set of replicas, some of which could be byzantine, the non-faulty replicas would agree on the order for this client request. We use Figure 2 to schematically represent consensus.

PBFT [3] 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* and other replicas act as the backup. PBFT only 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 3). 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.

More Replicas. It is evident from the PBFT protocol that any system relying on PBFT’s design for consensus would be expensive. Hence, several other efficient BFT designs (each with its limitation) have emerged lately [27]. For instance, Q/U [28] attempts to reduce BFT consensus to a single-phase through the use of $5f + 1$ replicas, but cannot handle concurrent requests. HQ [29] builds on top of Q/U and permits concurrency only if the transactions are non-conflicting.

Speculative Execution. Zyzzyva [4] introduces speculative execution to the BFT protocols to yield a linear BFT protocol. In Zyzzyva’s design, as soon as a backup replica receives a request 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. Zyzzyva requires just one phase, so its design helps to gauge the maximum throughput that can be attained by a BFT protocol. However, if the primary is malicious, Zyzzyva needs the help of its good clients to ensure a correct order. If the clients are malicious, then Zyzzyva is unsafe until a good client participates. Further, Zyzzyva’s fast case requires a client to receive a response from all the $3f + 1$ replicas before it marks a request complete. Prior works [30] have shown that just one failure is enough to lead Zyzzyva to very low throughput. A recent protocol, *Proof-of-Execution* (PoE) tries to remove the limitations of Zyzzyva and outperforms PBFT [31]. However, PoE also requires one phase of quadratic communication among its replicas.

Multiple Primaries: Several protocols [5], [18], [32] suggest dedicating multiple replicas as primaries to gain higher throughput. Multiple primaries can boost the throughput when the system is not limited by resources and network. Further, multiple primary protocols require coordination for the execution of their requests in the correct order.

B. Chain Management

A blockchain is an immutable ledger that consists of a set of *blocks*. Each block contains necessary information regarding the executed transaction and the *previous* block in its chain. The data about the previous block helps any blockchain achieve immutability. The i -th block in the chain can be represented as: $B_i := \{k, d, v, H(B_{i-1})\}$

This block B_i contains the sequence number (k) of the client request, the digest (d) of the request, the identifier of the primary v who initiated the consensus, and the hash of the previous block, $H(B_{i-1})$. In each blockchain application, every replica independently maintains its copy of the blockchain. Prior to the start of consensus, the blockchain of each replica has no element. Hence, it is initialized with a *genesis* block [1]. The genesis block is marked as the first block in the chain

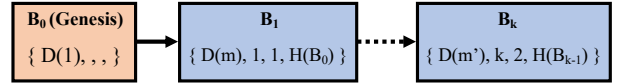


Fig. 4: A formal representation of the blockchain.

and contains dummy data. For instance, a genesis block can contain the hash of the identifier of the first primary, $H(\mathbb{P})$.

C. Alternative Blockchain Architectures

To improve the throughput attained by a permissioned blockchain application, researchers have also explored alternate architectures and designs to manage the blockchain.

DAG. Since the common data-structure in any blockchain application is the ledger, several systems incorporated a directed-acyclic graph to record the client transactions [16], [33], [34]. As a blockchain application expects a single order for all the transactions across all the replicas, so a DAG-based design allows replicas working on non-conflicting transactions to simultaneously record multiple transactions. However, a DAG-based design would require the merge of branches of a DAG once there are conflicting transactions, which in turn necessitates communication between all the replicas.

Sharding. Another approach to extract higher throughput from a blockchain system is to employ sharding [16], [17], [35]. Sharding splits the records accessed by the clients into several distinct partitions, where each partition is maintained by a set of replicas. Although sharding helps an application to attain high throughput when client transactions require access to only one partition, multi-partition transactions are expensive as they can require up to two additional phases to ensure safety.

Geo-Scale Clustering. Several protocols have suggested clustering replicas in the vicinity [36], [37]. For example, GeoBFT [37] facilitates running the PBFT protocol in each cluster in parallel. Although this design yields high-throughput, it reduces fault-tolerance as each cluster needs to have $3f + 1$ replicas.

Other Permissioned Systems. Several other permissioned blockchain systems such as Hyperledger Fabric [7], Multi-Chain [38], and Tendermint [39] have proposed high-level system architecture to achieve high throughput. For instance, Hyperledger Fabric presents a distinct paradigm of executing transactions first and then ensuring they have a valid order, while Tendermint advocates reliance on a synchronous setting to attain higher throughput. Despite these exciting principles, these works miss the low-level system details, which is the main focus of this work. Further, several existing BFT protocols have employed BFTSmart [19] as a standard implementation for PBFT. This is noteworthy as BFTSmart associates a non-pipelined architecture with PBFT and avoids other design optimizations for the sake of design simplicity.

III. DISSECTING PERMISSIONED BLOCKCHAIN

Most of the strategies we discussed in the previous section focussed at: (i) optimizing the underlying BFT consensus algorithm, and/or (ii) restructuring the way a blockchain is

maintained. We believe there is much more to render in the design of a permissioned blockchain system beyond these strategies. Hence, we identify several other key *factors* that reduce the throughput and increase the latency of a permissioned blockchain system or database.

Single-threaded Monolithic Design. There are ample opportunities available in the design of a permissioned blockchain application to extract parallelism. Several existing permissioned systems provide minimal to no discussion on how they can benefit from the underlying hardware or cores [16], [17], [25]. Due to the sustained reduction in hardware cost (as a consequence of Moore’s Law [40]), it is easy for each replica to have at least *eight* cores. Hence, by parallelizing the tasks across different threads and pipelining several transactions, a blockchain application can highly benefit from the available computational power.

Successive Phases of Consensus. Several works advocate the benefits of performing consensus on one request at a time [16], [41], while others promote aggregating client requests into large batches [7], [12]. We believe there is a communication and computation trade-off that needs to be analyzed before reaching such a decision. Hence, an optimal batching limit needs to be discovered.

Decoupling Ordering and Execution. On receiving a client request, each replica of a permissioned blockchain application has to order and execute that request. Although these tasks share a dependency, it is a useful design practice to separate them at the physical or logical level. At the physical level, distinct replicas can be used for execution. However, such an approach would incur additional communication costs. At the logical level, distinct threads can be asked to process requests in parallel, but additional hardware cores would be needed to facilitate such parallelism. In specific, a single entity performing both ordering and execution loses an opportunity to gain from inherent parallelism.

Strict Ordering. Permissioned blockchain applications rely on BFT protocols, which necessitate ordering of client requests in accordance with linearizability [3], [42]. Although linearizability helps in guaranteeing a safe state across all the replicas, it is an expensive property to achieve. Hence, we need an approach that can provide linearizability but is inexpensive. We observe that permissioned blockchain applications can benefit from delaying the ordering of client requests until execution. This delay ensures that although several client requests are processed in parallel, the result of their execution is in order.

Off-Memory Chain Management. Blockchain applications work on a large set of records or data. Hence, they require access to databases to store these records. There is a clear trade-off when applications store data in-memory or on an off-the-shelf database. Off-memory storage requires several CPU cycles to fetch data [43]. Hence, employing in-memory storage can ensure faster access, which in turn can lead to high system throughput.

Expensive Cryptographic Practices. Blockchain applications expect the exchange of several messages among the participating replicas and the clients, of which some may be

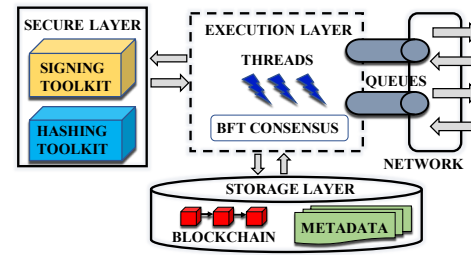


Fig. 5: ResilientDB Architecture.

byzantine. Hence, each blockchain application requires strong cryptographic constructs that allow a client or a replica to validate any message. These cryptographic constructs find a variety of uses in a blockchain application: (i) To sign a message. (ii) To verify an incoming message. (iii) To generate the digest of a client request. (iv) To hash a record or data.

To sign and verify a message, a blockchain application can employ either symmetric-key cryptography or asymmetric-key cryptography [44]. Although symmetric-key signatures, such as Message Authentication Code (MAC), are faster to generate than asymmetric-key signatures, such as Digital Signature (DS), DSs offer the key property of non-repudiation, which is not guaranteed by MACs [44]. Hence, several works suggest using DSs [7], [16], [17], [25]. However, a cleverly designed permissioned blockchain system can skip using DSs for a majority of its communication, which in turn will help increase its throughput. For generating digests or hash, a blockchain application needs to employ standard Hash functions, such as SHA256 or SHA3, which are secure.

IV. RESILIENTDB PERMISSIONED BLOCKCHAIN FABRIC

We now present our ResilientDB fabric, which incorporates our insights and fulfills the promise of an efficient permissioned blockchain system. In Figure 5, we illustrate the overall architecture of ResilientDB, which 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, they 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 ordered and executed. During ordering, the *secure layer* provides cryptographic support.

Since our aim is to present the design of a high-throughput permissioned blockchain system, for the rest of the discussion we use the simple yet robust PBFT protocol (explained in Section II) for reaching consensus among the replicas. Note that succeeding insights also apply to other BFT protocols.

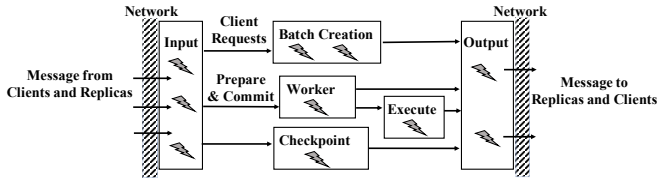


Fig. 6: Schematic representation of the multi-threaded deep-pipelines at each ResilientDB replica. The number of threads of each type can be varied depending on the requirements of the underlying consensus protocol.

A. Multi-Threaded Deep Pipeline

For implementing PBFT, we require ResilientDB to follow the primary-backup model. On receiving a client request, the primary replica must initiate PBFT consensus among all the backup replicas and ensure all the replicas execute this client request in the same order. Note that depending on the choice of BFT protocol, ResilientDB can be molded to adopt a different model (e.g. leaderless architecture).

In Figure 6, we illustrate the threaded-pipelined architecture of ResilientDB replicas. We permit increasing (or decreasing) the number of threads of each type. In fact one of the key goals of this paper is to study the effect of varying these threads on a permissioned blockchain. With each replica, we associate multiple *input* and *output* threads. In specific, we balance the tasks assigned to the input-threads, by requiring one input-thread to solely receive client requests, while two other input-threads to collect messages sent by other replicas. ResilientDB also balances the task of transmitting messages between the two output-threads by assigning equal clients and replicas to each output-thread. To facilitate this division, we need to associate a distinct *queue* with each output-thread.

B. Transaction Batching

ResilientDB allows both clients and replicas to batch their transactions. Using an optimal batching policy can help mask communication and consensus costs. A client can send a burst of transactions as a single request to the primary. Examples of applications where a client may batch multiple transactions are stock-trading, monetary-exchanges, and service level-agreements. The primary replica can also aggregate client requests together to significantly reduce the number of times a consensus protocol needs to be run among the replicas.

C. Modeling a Primary Replica

To facilitate efficient batching of requests, ResilientDB associates multiple *batch-threads* with the primary replica. When the primary replica receives a batch of requests from the client, it treats it as a single request. The input-thread at the primary assigns a monotonically increasing sequence number to each incoming client request and enqueues it into the common queue for the batch-threads. To prevent contention among the batch-threads, we design the common queue as *lock-free*. But *why have a common queue?* This allows us to

ensure that any enqueued request is consumed as soon as any batch-thread is available.

Each batch-thread also performs the task of verifying the signature of the client request. If the verification is successful, then it creates a batch and names it as the PRE-PREPARE message. PBFT also requires the primary to generate the digest of the client request and send this digest as part of the PRE-PREPARE message. This digest helps in identifying the client request in future communication. Hence, each batch-thread also hashes a batch and marks this hash as a digest. Finally, the batch-thread signs and enqueues the corresponding PRE-PREPARE message into the queue for an output-thread.

Apart from the client requests, the primary replica receives PREPARE and COMMIT messages from backup replicas. As the system is partially asynchronous, so the primary may receive both the PREPARE and COMMIT messages from a backup replica X before the PREPARE message from a backup Y . *How is this possible?* The replica X could have received sufficient number of PREPARE messages (that is $2f$) before the primary receives PREPARE from replica Y (total number of replicas are $n = 3f + 1$). In such a case, X would proceed to the next phase and broadcast its COMMIT message. Hence, to prevent any resource contention, we designate only one *worker-thread* the task of processing all these messages.

When the input-thread receives a PREPARE message, it enqueues that message in the *work-queue*. The worker-thread dequeues a message and verifies the signature on this message. If the verification is successful, then it records this message and continues collecting PREPARE messages corresponding to a PRE-PREPARE message until its count reaches $2f$. Once it reaches this count, then it creates a COMMIT message, signs and broadcasts this message. The worker-thread follows similar steps for a COMMIT message, except that it needs a total of $2f + 1$ messages, and once it reaches this count, it informs the *execute-thread* to execute the client requests.

D. Modeling a Backup Replica

As a backup replica does not create batches of client requests, ResilientDB assigns it fewer threads. When the input-thread at a backup replica receives a PRE-PREPARE message from the primary, then it enqueues it in the work-queue. The worker-thread at a backup dequeues a PRE-PREPARE message and checks if the message has a valid signature of the primary. If this is the case, then the worker-thread creates a PREPARE message, signs this message, and enqueues it in the queue for output-thread. Note that this PREPARE message includes the digest from the PRE-PREPARE message and the sequence number suggested by the primary. The output-thread broadcasts this PREPARE message on the network. Similar to the primary, each backup replica also collects $2f$ PREPARE messages, creates and broadcasts a COMMIT message, collects $2f + 1$ COMMIT messages, and informs the execute-thread.

E. Out-of-Order Message Processing

The key to the fast ordering of client requests is to allow ordering of multiple client requests to happen in parallel.

ResilientDB supports parallel ordering of client requests, while ensuring a single common order across all the replicas.

Example 1: Say a client C sends the primary replica P first request m_1 and then request m_2 . The input-thread at the primary P would assign a sequence number k to request m_1 and $k + 1$ to request m_2 . However, as the batch-threads can work at varying speeds, so it is possible that the consensus for requests m_1 and m_2 may either overlap, or some replica R may receive $2f + 1$ COMMIT messages for m_2 before m_1 .

In principle, Example 1 seems like a challenge for a blockchain application, as a replica may receive requests at sequence number $k + 1, k + 2, \dots$ before it commits request at number k . However, the property of out-of-order message processing is inherent in the design of most BFT protocols and is often overlooked.

Existing BFT protocols expect all the non-faulty replicas to act deterministic, that is, on identical inputs present identical outputs [3], [4], [18]. Further, they only accept a request after they have a guarantee that a majority of other replicas have also accepted the same request. For example, in the PBFT protocol, say a backup replica R receives a PRE-PREPARE message for client request m_1 with sequence number k . This replica R will not send a COMMIT message in support of the request m_1 until it receives $2f$ identical PREPARE messages from distinct replicas in support of m_1 . Further, the replica R will only execute request m_1 when it receives $2f + 1$ COMMIT messages from distinct replicas.

In the case of out-of-order message processing, if a replica gets $2f + 1$ COMMIT messages for a request with sequence number $k + 1$ before the request with number k , it will *not execute* $(k + 1)$ -th request before k -th request. Hence, the execution of all the succeeding requests has to be kept on hold. This ensures that the order of execution is identical across all the non-faulty replicas.

Of course, the primary \mathcal{P} could act malicious and could send all but the k -th request. To tackle such a scenario, BFT protocols already provide a primary-replacement (or *view-change*) algorithm [3], [4]. The aim of the view-change algorithm is to deterministically replace the malicious primary \mathcal{P} with a new primary \mathcal{P}' . It is the duty of this new primary \mathcal{P}' to ensure all the replicas reach the common state otherwise it will also be replaced. As ResilientDB uses existing BFT protocols, we skip presenting the details of existing view-change algorithm.

F. Efficient Ordered Execution

Although we parallelize consensus, we ensure execution happens in order. For instance, the requests m_1 and m_2 from Example 1 are executed in sequence order, that is, m_1 is executed before m_2 , irrespective of the order their consensus completed. At each replica, we dedicate a separate *execution-thread* to execute the requests. But, the key question remains: *how can we reduce the execution-thread's overhead of ordering.*

It is evident that the execution-thread has to wait for a notification from the worker-thread. In specific, we require the worker-thread to create an EXECUTE message and place this

message in the *appropriate* queue for the execution-thread. This EXECUTE message contains the identifier for the starting and ending transactions of a batch, which need to be executed. Note that we associate a large set of queues with the execution-thread. To determine the number of required queues for the execution-thread, we use the parameter QC .

$$QC = 2 \times Num_Clients \times Num_Req$$

Here, $Num_Clients$ represent the total number of clients in the system, while Num_Req represents the maximum number of requests a client can send without waiting for any response. We assume both of these parameters to be finite. Although QC can be very large, the queues are logical. So, the space complexity remains almost the same as for a single queue. But why is this practice advantageous?

Using this design the execute-thread can deterministically select the queue to dequeue. If k was the sequence number for last executed request, the execute-thread calculates $r = (k + 1) \bmod QC$ and waits for an EXECUTE message to be enqueued in its r -th queue. This design is more efficient than having a single queue, as a single queue would have forced several dequeues and enqueues until finding the next request in order to execute. Alternatively, we could have employed *hash-maps* but collision resistant hash functions are expensive to compute and verify [44].

Once the execution is complete, the execution-thread creates a RESPONSE message and enqueues it in the queue for output-threads, to send to the client. Note that ensuring execution happens in order provides a guarantee that a single common order is established across all the non-faulty replicas.

Block Generation. It is at this stage where we require the execution thread to create a block representing this batch of requests. As the execute-thread has access to the previous block in the chain, so it can easily hash this previous block and store this hash in the new block. Note that this step provides another opportunity for parallelism where the execute-thread can delegate the task of creating a new block to another thread.

G. Checkpointing

We also require replicas to periodically generate and exchange *checkpoints*. These checkpoints serve *two* purposes: (1) Help a failed replica to update itself to the current state. (2) Facilitate cleaning of old requests, messages and blocks. However, as checkpointing requires exchange of large messages, so we ensure it does not impact the throughput of the system. ResilientDB deploys a separate *checkpoint-thread* at each replica to collect and process incoming CHECKPOINT messages. These checkpoint messages simply include all the blocks generated since the last checkpoint. In specific, a CHECKPOINT message is sent only after a replica has executed Δ requests. Once execute-thread completes executing a batch, it checks if the sequence number of the batch is a *multiple of* Δ . If such is the case, it sends a CHECKPOINT message to all the replicas. When a replica receives $2f + 1$ identical CHECKPOINT messages from distinct replicas, then it marks

the checkpoint and clears all the data before the previous checkpoint [3], [4].

H. Buffer Pool Management

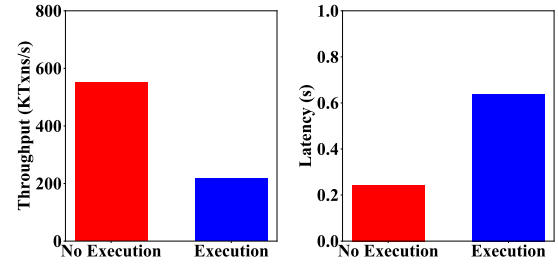
Until now, our description revolved around how a replica uses messages and transactions. In ResilientDB, we designed a *base class* that represents all the messages. To create a new message type, one has to simply inherit this base class and add required properties. Although on delivery to the network, each message is simply a buffer of characters, this typed representation helps us to easily manipulate the required properties. Similarly, we have designed a *base class* to represent all client transactions. An object of this transaction class includes: transaction identifier, client identifier, and transaction data, among many other properties.

When a message arrives in the system, a replica needs to allocate (`malloc`) space for that messages. Similarly, when a replica receives a client request, it needs to allocate corresponding transaction objects. When the lifetime of a message ends (or a new checkpoint is established), then the memory occupied by that message (or transactions object) needs to be released (`free`). To avoid such frequent allocations and deallocations, we adopt the standard practice of maintaining a set of *buffer pools*. At the system initialization stage, we create a large number of empty objects representing the messages and transactions. So instead of doing a `malloc`, these objects are extracted from their respective pools and are placed back in the pool during the `free` operation.

V. EXPERIMENTAL ANALYSIS

We now analyze how various parameters affect the throughput and latency of a Permissioned Blockchain (henceforth abbreviated as PBC) system. For the purpose of this study we use our ResilientDB fabric. Although ResilientDB can employ any BFT consensus protocol, we use the simple PBFT protocol to ensure the system design remains as our key focus. To ensure a holistic evaluation, we attempt to answer the following questions:

- (Q1) Can a well-crafted system based on a classical BFT protocol outperform a modern protocol?
- (Q2) How much gains in throughput (and latency) can a PBC achieve from pipelining and threading?
- (Q3) Can pipelining help a PBC become more scalable?
- (Q4) What impact does batching of requests has on a PBC?
- (Q5) Do multi-operation requests impact the throughput and latency of a PBC?
- (Q6) How increasing the message size impacts a PBC?
- (Q7) What effect do different types of cryptographic signature schemes have on the throughput of a PBC?
- (Q8) How does a PBC fare with in-memory storage versus a storage provided by a standard database?
- (Q9) Can an increased number of clients impact the latency of a PBC, while its throughput remains unaffected?
- (Q10) Can a PBC sustain high throughput on a setup having fewer number of cores?
- (Q11) How impactful are replica failures for a PBC?



(a) System throughput. (b) Latency.

Fig. 7: Upper bound measurements: (i) Primary responds back to the client without Execution, and (ii) Primary executes the request and replies to the client.

A. Evaluation Setup

We employ Google Cloud infrastructure at Iowa region to deploy our ResilientDB. For replicas, we use `c2` machines with an 8-core Intel Xeon Cascade Lake CPU running at 3.8GHz and having 16GB memory, while for clients we use `c2` 4-core machines. We run each experiment for 180 seconds, and collect results over *three* runs to average out any noise.

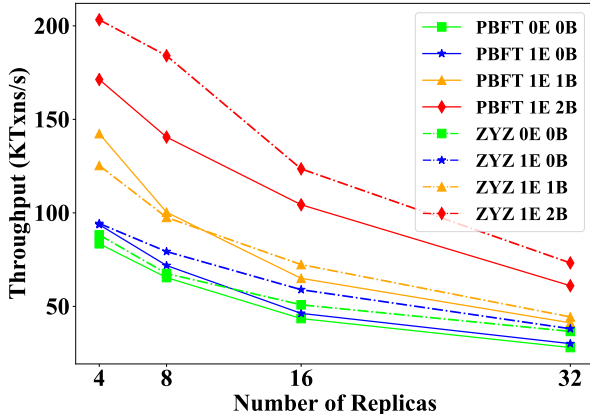
We use YCSB [2], [45] for generating workload for client requests. For creating a request, each client indexes a YCSB table with an active set of 600K records. In our evaluation, we require client requests to contain only write accesses, as a majority of blockchain requests are updates to the existing data. During the initialization phase, we ensure each replica has an identical copy of the table. Each client YCSB request is generated from a uniform Zipfian distribution.

Unless *explicitly* stated otherwise, we use the following setup: We invoke up to 80K clients on 4 machines and run consensus among 16 replicas. We employ batching to create batches of 100 requests. For communication among replicas and clients we employ digital signatures based on ED25519, and for communication among replicas we use a combination of CMAC and AES [44]. At each replica, we permit one worker-thread, one execute-thread and two batch-threads

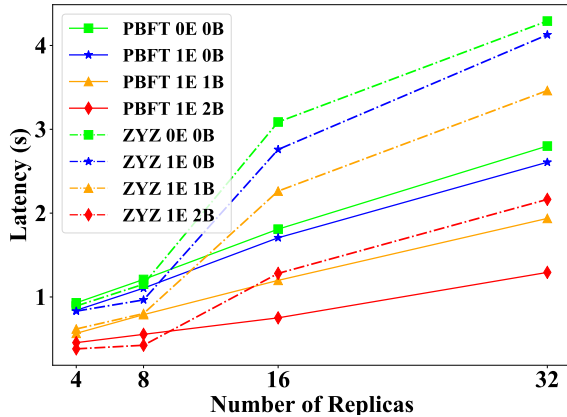
B. Effect of Threading and Pipelining

In this section, we analyze and answer questions Q1 to Q3. For this study, we vary the system parameters in two dimensions: (i) We increase the number of replicas participating in the consensus from 4 to 32. (ii) We expand the pipeline and gradually balance the load among parallel threads.

We first try to gauge the upper bound performance of our system. In Figures 7a and 7b, we measure the maximum throughput and latency a system can achieve, when there is no communication among the replicas or any consensus protocol. We use the term *No Execution* to refer to the case where all the clients send their requests to the primary replica and primary simply responds back to the client. We count every query responded back in the system throughput. We use the term *Execution* to refer to the case where the primary replica executes each query before responding back to the client. In both of these experiments, we allowed two threads to work



(a) System throughput.



(b) Latency.

Fig. 8: System throughput and latency, on varying the number of replicas participating in the consensus. Here, E denotes number of execution-threads, while B denotes batch-threads.

independently at the primary replica, that is, no ordering is maintained. Clearly, the system can attain high throughputs (up to 500K txns/s) and has low latency (up to 0.25s).

Next, we take two consensus protocols: PBFT and Zyzzyva, and we ensure that at least $3f + 1$ replicas are participating in the consensus. We gradually move our system towards the architecture of Figure 6. In Figures 8a and 8b, we show the effects of this gradual increase. We denote the number of execution-threads with symbol E, and batch-threads with symbol B. For all these experiments, we used only *one* worker-thread. The key intuition behind these plots is to continue expanding the stages of pipeline and the number of threads, until system can no longer increase its throughput. In this manner, it would be easy to observe design choices that could make even PBFT outperform Zyzzyva, that is, benefits of a *well-crafted implementation*.

On close observation of Figure 8a, we can trivially highlight the benefits of a good implementation. Further, these plots help to confirm our intuition that a multi-threaded pipelined architecture for a PBC outperforms a single-threaded design.

This is the key reason why our design of ResilientDB employs *one* execution-thread and *two* batch-threads apart from a single worker-thread.

Next, we explain our methodology for gradual changes. We first modified ResilientDB to ensure there are no additional threads for execution and batching, that is, all tasks are done by one worker-thread (0E 0B). On scaling this system we realized that this worker-thread was getting fully utilized. Hence, we partially divide the load by having an execute-thread (1E 0B). However, we again observed that the worker-thread at the primary was getting completely utilized. So we had an opportunity to introduce a separate thread to create batches (1E 1B). Although worker-thread was no longer saturating, the batch-thread was overloaded with the task of creating batches. Hence, we further divided the task of batching among multiple batch-threads (1E 2B) and ensured none of the batch-threads were fully utilized. Figures 9a and 9b show the utilization level for different threads at a replica. In this figure, we mark 100% as the maximum utilization for any thread. Using the bar for *cumulative utilization*, we show a summation of the utilization for all the threads, for any experiment. Note that for PBFT 1E 2B, the worker-thread at the backup replicas have started to saturate. But, as the architecture at the non-primary is following our design, so we split no further.

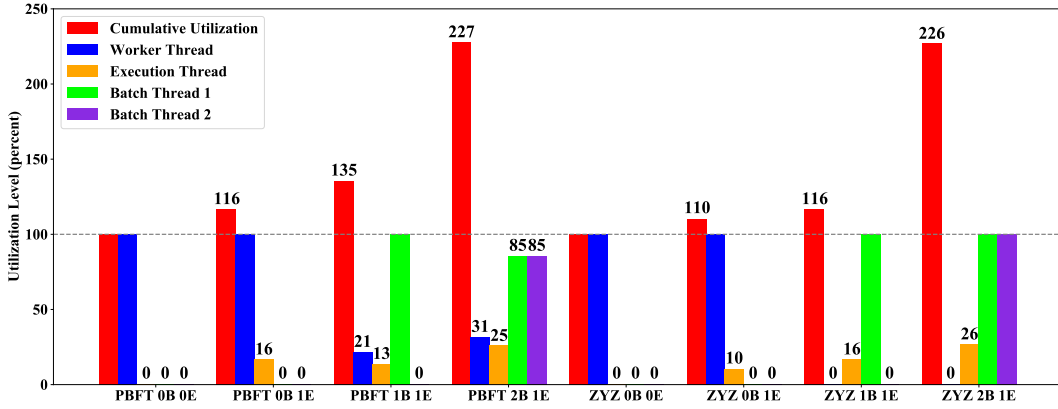
It can be observed that if PBFT is given benefit of ResilientDB’s standard pipeline (1E 2B), then it can attain higher throughput than all but one Zyzzyva implementations. The only Zyzzyva implementation (1E 2B) that outperforms PBFT is the one that employs ResilientDB’s standard threaded-pipeline. Further, even the simpler implementation for PBFT (1E 1B) attains higher throughput than Zyzzyva’s 0E 0B and 1E 0B implementations.

We have always stated that the design of ResilientDB is independent of the underlying consensus protocol. This can be observed from the fact that when Zyzzyva is given ResilientDB’s standard pipeline, then it can achieve throughput of 200K txns/s. Note that in majority of the settings PBFT incurs less latency than Zyzzyva. This is an effect of Zyzzyva’s algorithm, which requires the client to wait for replies from all the n replicas, where for PBFT the client only needs $f + 1$ responses. To *summarize*: (i) PBFT’s throughput (latency) increases (reduces) by $1.39\times$ (58.4%) on moving from 0E 0B setup to 1E 2B. (ii) Zyzzyva’s throughput (latency) increases (reduces) by $1.72\times$ (63.19%) on moving from 0E 0B setup to 1E 2B. (iii) Throughput gains up to $1.07\times$ are possible on running PBFT on an efficient setup, in comparison to basic setups for Zyzzyva.

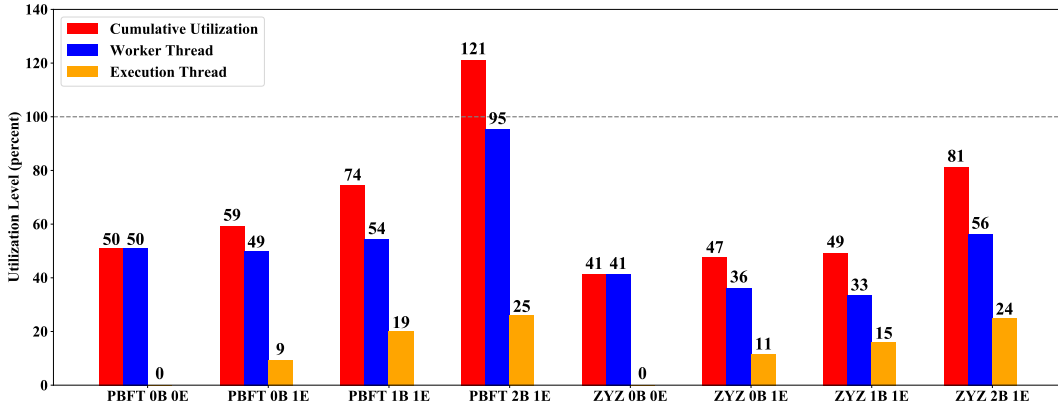
C. Effect of Transaction Batching

We now try to answer question Q4 by studying how batching the client transactions impacts the throughput and latency of a PBC. For this study, we increase the size of a batch from 1 to 5000.

Using Figures 10a and 10b, we observe that as the number of transactions in a batch increases, the throughput increases until a limit (at 1000) and then starts decreasing (at 3000).



(a) Primary Replica.



(b) Backup Replica.

Fig. 9: Utilization level of different threads at a replica. The mean is at 100%, which implies the thread is completely utilized.

At smaller batches, more consensus are taking place, and hence communication impacts the system throughput. Hence, larger batches help reduce the consensus. However, when the transactions in a batch are increased further, then the size of the resulting message and the time taken to create a batch by a batch-thread, reduces the system throughput. Hence, any PBC needs to find an optimal number of client transactions that it can batch. To *summarize*: batching can increase throughput by up to 66 \times and reduce latency by up to 98.4%.

D. Effect of Multi-Operation Transactions

We now answer question Q5, that is, understand how multi-operation transactions affect the throughput of a system? In Figures 11a and 11b, we increase the number of operations per transaction from 1 to 50. Further, we increase the number of batch-threads from 2 to 5, while having one worker-thread and one execute-thread. Although multi-operation transactions are common, prior works do not provide any discussion on such transactions. Notice that these experiments are orthogonal counterparts of the experiments in the previous section.

It is evident from these figures that on increasing the number of operations per transaction, the system throughput decreases. This decrease is a consequence of batch-threads getting saturated as they perform task of batching and allocating resources

for transaction. Hence, we ran several experiments with distinct counts for batch-threads. An increase in the number of batch-threads helps the system to increase its throughput, but the gap reduces significantly after the transaction becomes too large (at 50 operations). Similarly, more batch-threads help to decrease the latency incurred by the system.

Alternatively, we also measure the total number of operations completed in each experiment. Notice that if we base the throughput on the number of operations executed per second, then the trend has completely reversed. Indeed, this makes sense as in fewer rounds of consensus, more operations have been executed. To *summarize*: multi-operation transactions can cause a decrease of 93% in throughput and an increase of 13.29 \times in latency, on the two batch-threads setup. An increase in batch-threads from two to five increases throughputs up to 66% and reduces latencies up to 39%.

E. Effect of Message Size

We now try to answer question Q6 by increasing the size of the PRE-PREPARE message in each consensus. The key intuition behind this experiment is to gauge how well a PBC system performs when the requests sent by a client are large. Although each batch includes only 100 client transactions, individually, these requests can be large. Hence, these exper-

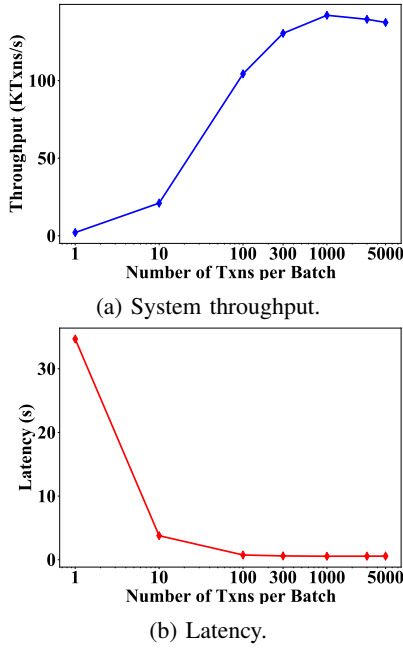


Fig. 10: System throughput and latency on varying the number of transactions per batch. In this experiment, 16 replicas participate in consensus.

iments are aimed at exploiting a different system parameter than the plots of Figure 10.

In Figures 12a and 12b, we study the variation in throughput and latency by increasing the size of a PRE-PREPARE message. We do this by adding a payload to each message, which includes a set of integers (8byte each). The cardinality of this set is kept equal to the desired message size.

It is evident from these plots that as the message size increases, there is a decrease in the system throughput and an increase in the latency incurred by the client. This is a result of network bandwidth becoming a limitation, due to which it takes extra time to push more data onto the network. Hence, in this experiment, the system reaches a network bound before any thread can hit its computational bound. This leads to all the threads being idle. To *summarize*: On moving from 8KB to 64KB messages, there is a 52% decrease in throughput and 1.09 \times increase in latency.

F. Effect of Cryptographic Signatures

In this section, we answer question Q7 by studying the impact of different cryptographic signature schemes. The key intuition behind these experiments is to determine which signing scheme helps a PBC achieve the highest throughput while preventing byzantine attacks. For this purpose, we run four different experiments to measure the system throughput and latency when: (i) no signature scheme is used, (ii) everyone uses digital signatures based on ED25519, (iii) everyone uses digital signatures based on RSA, and (iv) all replicas use CMAC+AES for signing, while clients sign their message using ED25519.

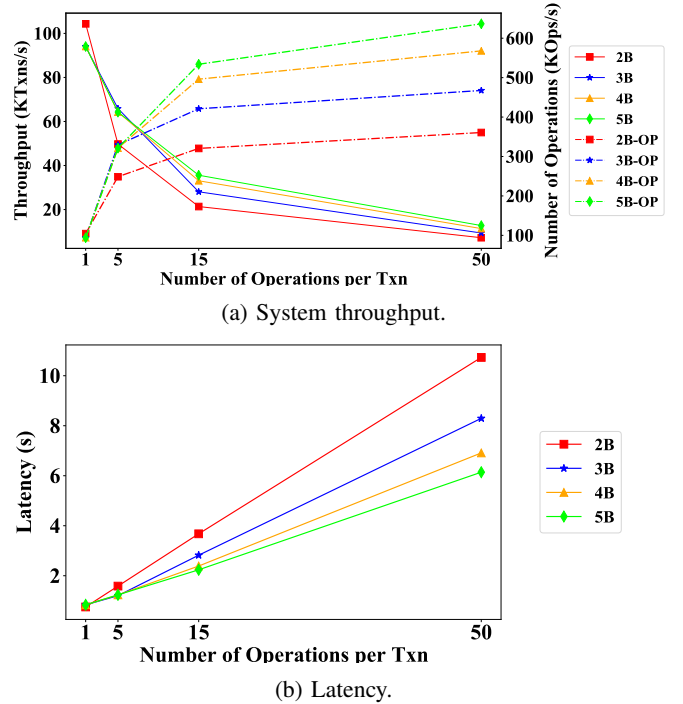


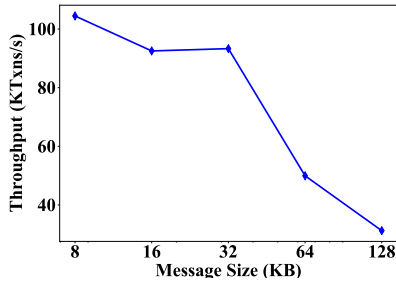
Fig. 11: System throughput and latency on varying the number of operations per transaction. Here, B denotes the number of batch-threads used in the experiment.

Figures 13a and 13b help us to illustrate the throughput attained and latency incurred by ResilientDB for different configurations. It is evident that ResilientDB attains maximum throughput when no signatures are employed. However, such a system does not fulfill the minimal requirements of a permissioned blockchain system. Further, using just digital signatures for signing messages is not exactly the best practice. An optimal configuration can require clients to sign their messages using digital signatures, while replicas can communicate using MACs. To *summarize*: (i) use of cryptography reduces throughput by at least 49% and increases latency by 33%. (ii) choosing RSA over CMAC, ED25519 combination would increase latency by 125 \times .

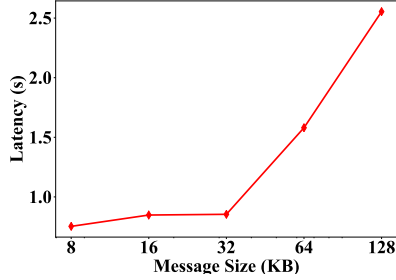
G. Effect of Memory Storage

We now try to answer question Q8 by studying the trade-off of having in-memory storage versus off-memory storage in a PBC. For testing off-memory storage, we integrate SQLite [46] with our ResilientDB architecture. We use SQLite to store and access the transactional records. As SQLite is external to our ResilientDB fabric, so we developed API calls to read and write its tables. Note that until now, for all the experiments, we assumed in-memory storage, that is, records are written and accessed in an in-memory key-value data-structure.

In Figures 14a and 14b, we illustrate the impact on system throughput and latency in the two cases. For the in-memory storage, we require the execute-thread to read/write the key-value data-structure. For SQLite, execute-thread initiates an

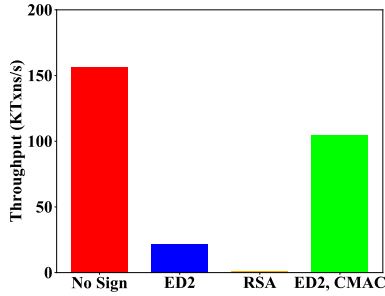


(a) System throughput.

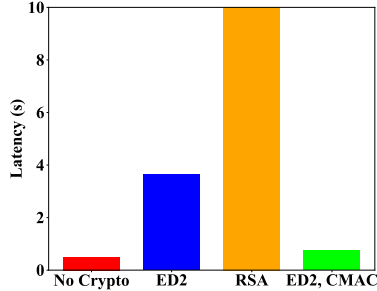


(b) Latency.

Fig. 12: System throughput and latency on varying the message size. Here, 16 replicas participate in consensus.



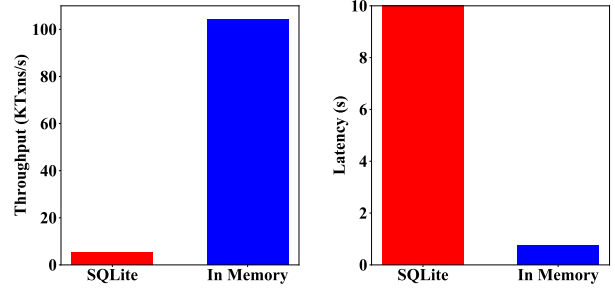
(a) System throughput.



(b) Latency.

Fig. 13: System throughput and latency with different signature schemes. Here, 16 replicas participate in consensus.

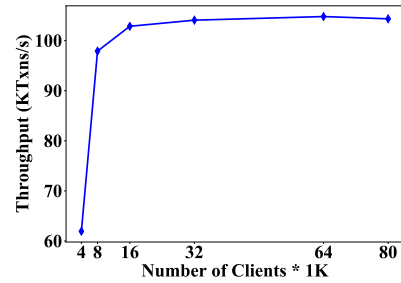
API call and waits for the results. It is evident from these plots that access to off-memory storage (SQLite) is quite expensive. Further, as execute-thread is busy-waiting for a reply, it performs no useful task. To *summarize:*, choosing SQLite over in-memory storage reduces throughput by 94% and increase latency by 24 \times .



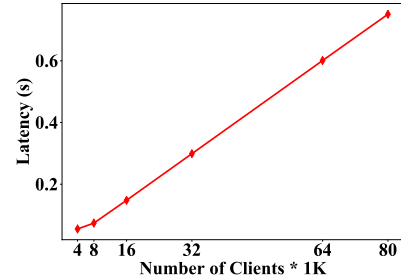
(a) System throughput.

(b) Latency.

Fig. 14: System throughput and latency for in-memory storage vs. off-memory storage. Here, 16 replicas used for consensus.



(a) System throughput.



(b) Latency.

Fig. 15: System throughput and latency on varying the number of clients. Here, 16 replicas participate in consensus.

H. Effect of Clients

We now study the impact of clients on a PBC system, and as a result, work towards answering question Q9. We observe the changes in throughput and latency on increasing the number of clients sending requests to a PBC from 4K to 80K.

Through Figure 15a, we conclude that on increasing the number of clients, the throughput for the system increases to some extent (up to 32K), and then it becomes constant. This is a result of all the threads processing at their maximum capacities, that is, the system is unable to handle any more client requests. As the number of clients increases, an increased set of requests have to wait in the queue before they can be processed. This wait can even cause a slight dip in throughput (on moving from 64K to 80K clients). This delay in processing causes a linear increase in the latency incurred by the clients (as shown in Figure 15b). To *summarize:* we

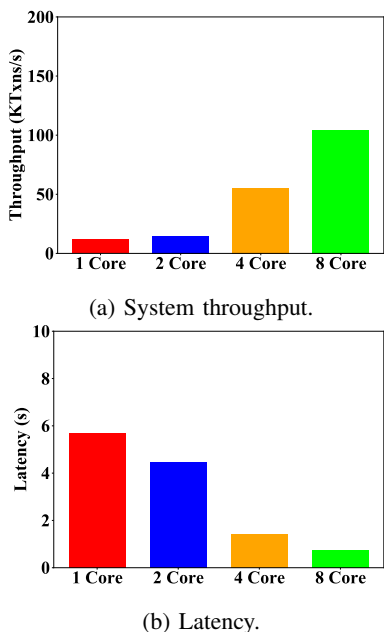


Fig. 16: System throughput and latency on varying the number of hardware cores. Here, 16 replicas participate in consensus.

observe that an increase in the number of clients from 16K to 80K helps the system to gain an additional 1.44% throughput but incurs $5\times$ more latency.

I. Effect of Hardware Cores

We now answer question Q10 by analyzing the effects of a deployed hardware on a PBC application. In specific, we want to deploy our replicas on different Google Cloud machines having 1, 2, 4 and 8 cores. We use Figures 16a and 16b to illustrate the throughput and latency attained by our ResilientDB system on different machines. For all these experiments, we require 16 replicas to participate in the consensus. These figures affirm our claim that if replicas run on a machine with fewer cores, then the overall system throughput will be reduced (and higher latency will be incurred). As our architecture (refer to Figure 6) requires several threads, so on a machine with fewer cores our threads face resource contention. Hence, ResilientDB attains maximum throughput on the 8-core machines. To **summarize**: deploying ResilientDB replicas on an 8-core machine, in comparison to the 1-core machines, leads to an $8.92\times$ increase in throughput.

J. Effect of Replica Failures

We now try to answer question Q11 by analyzing whether a fast BFT consensus protocol can withstand replica failures. This experiment also illustrates the impact of failures on a PBC. In specific, we perform a head-on comparison of Zyzzyva against PBFT, while allowing some backup replicas to fail.

In Figures 17a and 17b, we illustrate the impact of failure of *one* replica and *five* replicas on the two protocols. For this experiment we require at most 16 replicas to participate in consensus. Note that for $n = 16$, the maximum number of

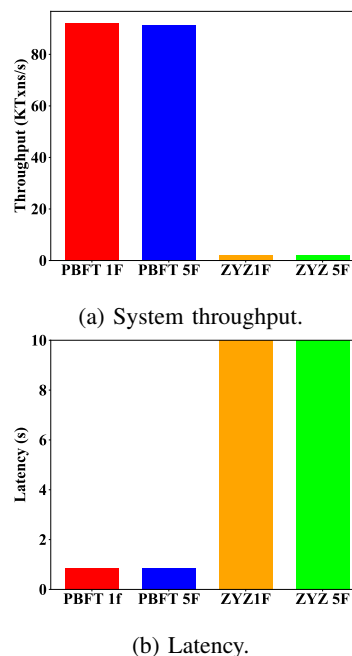


Fig. 17: System throughput and latency on failing non-primary replicas. Here, 16 replicas participate in consensus.

failures a BFT system can handle are $f = 5$. Hence, we evaluate both the protocols under minimum and maximum simultaneous failures.

On increasing the number of failures from one to five, there is a small dip in the throughput for both the protocols. This dip is not visible due to the high scaling of the graph. For PBFT, in comparison to the failure-free case, there is not a significant decrease in throughput as none of its phases require more than $2f + 1$ messages.

In case of Zyzzyva, the system faces a pronounced reduction in its throughput with just one failure. The key issue with Zyzzyva is that its clients need responses from all the replicas. So even one failure makes a client *wait* until it *timesouts*. This wait causes a significant reduction in its throughput. Note that finding an optimal amount of time a client should wait is a hard problem [30], [47]. Hence, we approximate this by requiring clients to wait for only a small time.

Protocols like Zyzzyva advocate for a twin path model [4], [48]. In these protocols, each replica achieves consensus by following a fast path until the system faces a failure. Once a failure happens, these protocols decide to switch to a slower path. Such a design heavily relies on the value of timeout. If the timeout is large, then these protocols face a large reduction in throughput. For example, in Zyzzyva, a larger timeout implies clients have to wait for a larger amount of time before initiating the next phase. If the network is dynamic, then the value of timeout can continuously change. Thus, finding the optimal value of timeout is hard. Another way to boost the throughput of these protocols is to assume there are a sufficient number of clients that can help offset the effects of timeout.

VI. OBSERVATION

Based on the results presented in the previous section, we make two high-level conclusions:

- A slow classical BFT protocol running on a well-crafted implementation (like ResilientDB), can outperform a fast BFT protocol implemented on a protocol-centric design.
- No single parameter can alone substantially improve the throughput (or reduce latency) of the underlying PBC. The key reason our ResilientDB framework can attain high throughputs and incurs low latency is that it attempts at optimally utilizing several parameters.

Threading and Pipelining. In Section II, we discussed several works that either present new protocols to improve the performance of a PBC or illustrate novel use-cases for blockchain. These works rarely focus on the implementation of a replica itself and can significantly gain throughput by adopting an architecture similar to our ResilientDB. Further, caution needs to be taken while introducing parallelism as unnecessary threads can cause resource contention or deadlocks (e.g., multiple execution-threads can cause data-conflicts).

Batching and Multiple Operations. Several works suggest batching client requests, while others have vetoed against such a choice. Our results show that the optimal use of batching can help to reduce the cost of consensus by merging multiple consensus into one. However, over-batching does introduce a communication trade-off. Hence, each PBC application should determine the optimal set of client requests to batch. Clients can also employ multi-operation transactions. In practice, such a transaction includes at most ten operations. Hence, employing operations per second as a metric to measure throughput may be a good idea.

Message Size and Payload. Depending on the application targeted by a PBC, the clients can send requests that have a large size. For example, a client can require the execution of a specific code. If multiple large requests are batched together, then the network may consume resources in splitting a message into packets, transmitting these packets, and aggregating these packets at the destination. Hence, depending on the application, batching just ten large requests may allow the system to return high throughput.

Cryptographic Signatures. Although the use of cryptographic signatures bottlenecks the system throughput, their use is essential for safety. We observe that a combination of MACs and DSs can help guarantee both safety and high throughput. For instance, digital signatures are only necessary for messages that need to be *forwarded*. Hence, in a PBC, only clients need to digitally sign their requests. For communication among the replicas, MACs suffice, as in most of the BFT protocols, no replica forwards messages of any other replica. Hence, the property of non-repudiation is implicitly satisfied.

Chain Storage. PBC applications need to store client records and other metadata. We observed that the use of in-memory data-structures is better than off-memory storage, such as SQLite. The key reason a PBC system can avoid frequent access to off-memory storage is that at all times,

at most f replicas can fail. Hence, if persistent storage is required, then it can be performed asynchronously or delayed until periods of low contention.

Replica Failures. We know that failures are common. Either a replica may fail, or messages may get lost. A PBC system needs to be ready to face these situations. Hence, the system design must not rely on a BFT protocol that works well in non-failure cases but attains low throughput under simple failures. We observed that designs employing protocols like Zyzzyva can have negligible throughput with just one failure. Further, some protocols suggest the use of two modes, fast path and slow path [48]. Although such protocols attain high throughputs in the fast path, they switch to the slow path on failures. Note that this switch happens when some replica or client timeouts. Determining the optimal value for timeouts is hard [30], [47]. Thus, twin path protocols may not be suitable if the network is dynamic.

VII. CONCLUSIONS

In this paper, we present a high-throughput yielding permissioned blockchain framework, ResilientDB. By dissecting ResilientDB, we analyze several factors that affect the performance of a permissioned blockchain system. This allows us to raise a simple question: *can a well-crafted system based on a classical BFT protocol outperform a modern protocol?* We show that the extensively parallel and pipelined design of our ResilientDB fabric does allow even PBFT to gain high throughputs (up to 175K) and outperform common implementations of Zyzzyva. Further, we perform a rigorous evaluation of ResilientDB and illustrate the impact of different factors such as cryptography, chain management, monolithic design, and so on. We envision the practices adopted in ResilientDB to be included in designing and testing newer BFT protocols and permissioned blockchain applications.

Acknowledgments. We would like to thank DOE Award No. DE-SC0020455 for funding our startup MokaBlox LLC.

REFERENCES

- [1] S. Gupta and M. Sadoghi, *Blockchain Transaction Processing*. Springer International Publishing, 2018, pp. 1–11.
- [2] T. T. A. Dinh, J. Wang, G. Chen, R. Liu, B. C. Ooi, and K.-L. Tan, “BLOCKBENCH: A framework for analyzing private blockchains,” in *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 2017, pp. 1085–1100.
- [3] M. Castro and B. Liskov, “Practical byzantine fault tolerance,” in *Proceedings of the Third Symposium on Operating Systems Design and Implementation*. USENIX Association, 1999, pp. 173–186.
- [4] 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*. ACM, 2007, pp. 45–58.
- [5] 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)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 146. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, pp. 44:1–44:3.
- [6] V. K. Garg, J. Bridgman, and B. Balasubramanian, “Accurate byzantine agreement with feedback,” in *OPODIS*, 2011.

- [7] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, S. W. Cocco, and J. Yellick, "Hyperledger fabric: A distributed operating system for permissioned blockchains," in *Proceedings of the Thirteenth EuroSys Conference*, ser. EuroSys 18. Association for Computing Machinery, 2018.
- [8] S. Gupta and M. Sadoghi, "EasyCommit: A non-blocking two-phase commit protocol," in *Proceedings of the 21st International Conference on Extending Database Technology*. Open Proceedings, 2018, pp. 157–168.
- [9] T. M. Qadah and M. Sadoghi, "QueCC: A queue-oriented, control-free concurrency architecture," in *Proceedings of the 19th International Middleware Conference*, 2018, pp. 13–25.
- [10] T. Qadah, S. Gupta, and M. Sadoghi, "Q-Store: Distributed, Multi-partition Transactions via Queue-oriented Execution and Communication," in *Proceedings of the 23rd International Conference on Extending Database Technology, EDBT*. OpenProceedings.org, 2020, pp. 73–84.
- [11] S. Gupta and M. Sadoghi, "Efficient and non-blocking agreement protocols," *Distributed and Parallel Databases*, vol. 38, pp. 287–333, 2020.
- [12] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2009. [Online]. Available: <https://bitcoin.org/bitcoin.pdf>
- [13] G. Wood, "Ethereum: a secure decentralised generalised transaction ledger," 2016, EIP-150 revision. [Online]. Available: <https://gavwood.com/paper.pdf>
- [14] N. Popper, "Worries grow that the price of bitcoin is being propped up," Jan. 2018. [Online]. Available: <https://www.nytimes.com/2018/01/31/technology/bitfinex-bitcoin-price.html>
- [15] J. R. Douceur, "The Sybil Attack," in *First International Workshop on Peer-to-Peer Systems*, ser. IPTPS '01. London, UK, UK: Springer-Verlag, 2002, pp. 251–260.
- [16] M. J. Amiri, D. Agrawal, and A. E. Abbadi, "CAPER: A cross-application permissioned blockchain," *PVLDB*, vol. 12, no. 11, pp. 1385–1398, 2019.
- [17] H. Dang, T. T. A. Dinh, D. Loghin, E.-C. Chang, Q. Lin, and B. C. Ooi, "Towards scaling blockchain systems via sharding," in *Proceedings of the 2019 International Conference on Management of Data*. ACM, 2019, pp. 123–140.
- [18] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, "Hot-Stuff: BFT consensus with linearity and responsiveness," in *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. ACM, 2019, pp. 347–356.
- [19] A. Bessani, J. Sousa, and E. E. P. Alchieri, "State machine replication for the masses with bft-smart," in *DSN*, 2014.
- [20] R. Buyya et al., "A manifesto for future generation cloud computing: Research directions for the next decade," *ACM Comput. Surv.*, 2018.
- [21] J. Gallard and A. Lebre et al., "Architecture for the next generation system management tools," *Future Generation Comp. Syst.*, 2012.
- [22] Z. Niu and B. He, "A study of big data computing platforms: Fairness and energy consumption," in *IC2E Workshop*, 2016.
- [23] M. Sadoghi and S. Blanas, *Transaction Processing on Modern Hardware*, ser. Synthesis Lectures on Data Management. Morgan & Claypool, 2019.
- [24] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, "Algorand: Scaling byzantine agreements for cryptocurrencies," in *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 2017, pp. 51–68.
- [25] M. Zamani, M. Movahedi, and M. Raykova, "RapidChain: Scaling blockchain via full sharding," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 931–948.
- [26] F. Nawab and M. Sadoghi, "Blockplane: A global-scale byzantizing middleware," in *35th International Conference on Data Engineering*. IEEE, 2019, pp. 124–135.
- [27] S. Gupta, J. Hellings, S. Rahnema, and M. Sadoghi, "An in-depth look of BFT consensus in blockchain: Challenges and opportunities," in *Proceedings of the 20th International Middleware Conference Tutorials*, 2019, pp. 6–10.
- [28] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie, "Fault-scalable byzantine fault-tolerant services," in *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*. ACM, 2005, pp. 59–74.
- [29] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira, "HQ replication: A hybrid quorum protocol for byzantine fault tolerance," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*. USENIX Association, 2006, pp. 177–190.
- [30] 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*. USENIX Association, 2009, pp. 153–168.
- [31] S. Gupta, J. Hellings, S. Rahnema, and M. Sadoghi, "Proof-of-Execution: Reaching Consensus through Fault-Tolerant Speculation," vol. abs/1911.00838, 2019. [Online]. Available: <https://arxiv.org/abs/1911.00838>
- [32] S. Gupta, J. Hellings, and M. Sadoghi, "Scaling blockchain databases through parallel resilient consensus paradigm," vol. abs/1911.00837, 2019. [Online]. Available: <https://arxiv.org/abs/1911.00837>
- [33] C. Li, P. Li, W. Xu, F. Long, and A. C. Yao, "Scaling nakamoto consensus to thousands of transactions per second," *CoRR*, vol. abs/1805.03870, 2018. [Online]. Available: <https://arxiv.org/abs/1805.03870>
- [34] Y. Sompolinsky, Y. Lewenberg, and A. Zohar, "SPECTRE: A fast and scalable cryptocurrency protocol," 2018. [Online]. Available: <https://eprint.iacr.org/2016/1159>
- [35] M. J. Amiri, D. Agrawal, and A. El Abbadi, "Parblockchain: Leveraging transaction parallelism in permissioned blockchain systems," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, 2019, pp. 1337–1347.
- [36] Y. Amir, C. Danilov, D. Dolev, J. Kirsch, J. Lane, C. Nita-Rotaru, J. Olsen, and D. Zage, "Steward: Scaling byzantine fault-tolerant replication to wide area networks," *IEEE Transactions on Dependable and Secure Computing*, vol. 7, no. 1, pp. 80–93, 2010.
- [37] S. Gupta, S. Rahnema, J. Hellings, and M. Sadoghi, "ResilientDB: Global Scale Resilient Blockchain Fabric," *Proc. VLDB Endow.*, vol. 13, no. 6, pp. 868–883, Feb. 2020.
- [38] G. Greenspan, "Multichain private blockchain," *CoRR*, 2015.
- [39] E. Buchman, J. Kwon, and Z. Milosevic, "The latest gossip on BFT consensus," *CoRR*, vol. abs/1807.04938, 2018.
- [40] G. E. Moore, "Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff." *IEEE Solid-State Circuits Society Newsletter*, vol. 11, no. 3, pp. 33–35, Sep. 2006.
- [41] Z. István, A. Sorniotti, and M. Vukolić, "Streamchain: Do blockchains need blocks?" ser. SERIAL'18. ACM, 2018, pp. 1–6.
- [42] M. P. Herlihy and J. M. Wing, "Linearizability: A Correctness Condition for Concurrent Objects," *ACM TOPLAS*, vol. 12, no. 3, pp. 463–492, 1990.
- [43] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fifth Edition: A Quantitative Approach*, 5th ed. Morgan Kaufmann Publishers Inc., 2011.
- [44] J. Katz and Y. Lindell, *Introduction to Modern Cryptography*, 2nd ed. Chapman and Hall/CRC, 2014.
- [45] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM Symposium on Cloud Computing*. ACM, 2010, pp. 143–154.
- [46] S. Developers, "Sqlite home page," 2019. [Online]. Available: <https://sqlite.org/>
- [47] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche, "Upright cluster services," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. ACM, 2009, pp. 277–290.
- [48] 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)*, 2019, pp. 568–580.