International Journal of Networking and Computing – www.ijnc.org, ISSN 2185-2847
Volume 15, Number 1, pages 32–50, January 2025

Cataphract: A Batch Processing Method Specialized for BFT Databases

Aoi Kida
Department of Media and Governance, Graduate School of Keio University
Fujisawa, Kanagawa, 252-0882, Japan

Hideyuki Kawashima
Faculty of Environment and Information Studies, Keio University
Fujisawa, Kanagawa, 252-0882, Japan

Received: July 29, 2024
Accepted: September 11, 2024
Communicated by Masahiro Shibata

Abstract

This paper presents Cataphract, a batch processing method specialized for BFT databases. Batch processing is a common technique for byzantine fault-tolerant state machine replication (BFT SMR) and distributed databases. However, no batch processing method is optimized for BFT databases, which possess characteristics of both BFT SMR and distributed databases. Cataphract optimizes cryptographic and communication processing, which are bottlenecks in BFT databases. We evaluate Cataphract with Basil (state-of-the-art BFT database) in experiments. In an environment where nodes are within an availability zone, Basil with Cataphract demonstrates up to approximately 5.6 times higher throughput and reduces latency by up to about 55% compared to the vanilla Basil. In an environment where nodes are within a region, Basil with Cataphract demonstrates up to approximately 13.8 times higher throughput and reduces latency by up to about 74% compared to the vanilla Basil. In an environment where nodes are geographically distributed, Basil with Cataphract demonstrates up to approximately 80.4 times higher throughput and reduces latency by up to about 76% compared to the vanilla Basil.

Keywords: Byzantine fault tolerance, Distributed database, Batch processing, Transaction processing

1 Introduction

1.1 Background

Data-intensive applications such as recommendation systems [6], online marketplaces [49], and internet search platforms [7] are producing enormous volumes of information at an unprecedented rate. Users of these services demand instantaneous access to data, regardless of the time or circumstance. Studies [50, 51] have demonstrated that delays in response time significantly impact the selection

[0]This is a new paper based on the paper of Aoi Kida and Hideyuki Kawashima: Accelerating BFT Database with Transaction Reconstruction. 26th Workshop on Advances in Parallel and Distributed Computational Models (APDCM 2024).

of search result pages and user satisfaction. The performance requirements for these applications are exceptionally demanding, often pushing beyond the capabilities of traditional, centralized systems. As a result, meeting these stringent needs within a single-system architecture has become increasingly challenging.

Distributed data stores have emerged as a critical solution for these complex demands. These systems efficiently manage and store massive data across multiple nodes, facilitating rapid data retrieval to ensure users can access information swiftly. Moreover, their inherent distributed architecture provides fault tolerance, maintaining system reliability even when individual nodes fail. In distributed data stores, systems designed for crash fault tolerance (CFT) are predominant. Notable examples in this category include Google's Spanner [13], Bigtable [55], Facebook's TAO [56], Amazon's DynamoDB [15], and PingCAP's TiDB [14]. These systems represent the current mainstream approach in distributed data management, primarily focusing on maintaining system integrity and availability in the face of node crashes.

Besides crash faults, Byzantine faults can be caused by malicious attacks, software bugs, hardware errors, and so on. It has been reported that bugs in distributed systems and database management systems (DBMS) lead to Byzantine faults [5, 57–60]. Byzantine fault-tolerant (BFT) distributed data stores can also operate without malicious participants, allowing secure online data sharing among mutually distrustful participants. Systems with BFT are utilized across diverse domains, including cryptocurrencies [8], Internet of Things [53], healthcare [9], financial services [10,54], supply chain [11], and international trade platforms [12]. Nevertheless, their practical adoption is limited due to the low performance of BFT systems, which is a trade-off for robust fault-tolerance [1–3].

BFT distributed data stores offer a higher level of resilience, maintaining operational integrity even in the presence of malicious actors. This capability enables secure data sharing among participants who may not fully trust each other. The application of BFT systems spans a wide array of sectors, including digital currencies [8], IoT ecosystems [53], healthcare information systems [9], financial services platforms [10, 54], supply chain management [11], and international trade networks [12]. Despite their security features, BFT systems face challenges in widespread adoption. The primary hurdle lies in the performance trade-off inherent to these systems [1–3]. The enhanced fault-tolerance that BFT provides comes at the cost of reduced system performance, a significant barrier to their practical implementation in many scenarios. In the Byzantine fault-tolerant systems, BFT databases are pushing the boundaries of performance optimization. These databases employ concurrent transaction processing as a critical strategy to boost their efficiency. A seminal work of this approach is Basil, which represents the cutting edge in BFT database technology. By leveraging concurrency control techniques, Basil has outperformed other notable BFT systems such as Hotstuff [18] and BFT-SMaRt [19].

1.2 Problem

While BFT databases demonstrate performance advantages over other BFT systems, they still lag behind non-Byzantine fault-tolerant systems. Our analysis of the characteristics of BFT databases compared to non-Byzantine fault-tolerant replicated systems reveals two primary factors contributing to this performance gap:

1. Cryptographic Overhead: BFT databases rely heavily on digital signatures to ensure message integrity. These signatures serve three crucial functions: they prevent tampering with message contents, verify the authenticity of the message source, and provide non-repudiation, ensuring that a replica cannot deny sending a message. These security measures are essential in Byzantine environments where replicas may be vulnerable to attacks or operated by malicious actors.

2. Communication Complexity: All distributed systems require inter-node communication during transaction processing. However, BFT systems require more extensive communication protocols than their CFT counterparts. This increased communication overhead stems from the need to account for potentially incorrect messages from cluster nodes in a Byzantine setting. For instance, when comparing representative distributed consensus algorithms, Raft [23] can commit client requests with just one round trip, while PBFT [20] requires two round trips.

These factors, the cryptographic processing overhead and the increased communication complexity, collectively contribute to the performance limitations of BFT databases compared to non-Byzantine fault-tolerant systems despite their advantages over other BFT implementations.

1.3 Contribution: Cataphract

This study aims to improve the performance of BFT databases by reducing the overhead caused by communication and cryptographic operations in transaction processing. We propose Cataphract, a solution to streamline the cryptographic and communication processes of BFT databases. Our proposal introduces a novel batch processing method for transaction processing of BFT databases:

1. The BFT database breaks down multiple incoming transactions into individual operations.
2. These operations are then reassembled into a single, consolidated transaction.
3. The reconstructed transactions are processed in the same manner as the conventional protocol of the BFT database.
4. Results are then disaggregated and returned to the application in the original transaction format.

This reconstruction technique aggregates cryptographic and communication processes previously performed separately for each transaction. By batching these processes, we can leverage the non-linear relationship between input size and processing time, thereby reducing the overall system overhead and improving performance.

To evaluate Cataphract, we implemented it, applied it to the state-of-the-art BFT database, Basil, and evaluated it by varying the geographical locations of the nodes. In a situation where nodes are within an availability zone, Basil with the proposed method achieved up to 5.6 times higher throughput and reduced latency by up to about 55% over the original Basil. In a situation where nodes are within a region, Basil with the proposed method achieved up to 13.8 times higher throughput and reduced latency by up to about 74% over the original Basil. In a situation where nodes are geographically distributed, Basil with the proposed method achieved up to 80.4 times higher throughput and reduced latency by up to about 76% over the original Basil.

1.4 Organization

The rest of this paper is organized as follows. In Section 2, we introduce the BFT databases and discuss the bottlenecks. In Section 3, we present Cataphract and its application to Basil, a state-of-the-art BFT database. In Section 4, we evaluate Cataphract with Basil. In Section 5, we present related work. Finally, Section 6 concludes this paper.

2 Preliminaries

The consensus in distributed systems where nodes can be Byzantine faulty was formulated as the Byzantine Generals Problem [27] by Lamport et al. One of the approaches to address the Byzantine Generals Problem is Byzantine fault-tolerant State Machine Replication (BFT SMR) [18–20, 28]. BFT SMR is used to build a distributed system that continues to operate accurately and consistently, even in the presence of Byzantine actors. BFT SMR ensures consistency between replicas by executing operations sequentially, although this approach comes with a significant overhead.

2.1 BFT Databases

BFT databases guarantee consistency among replicas even if they execute operations concurrently by exploiting the parallelism of transactions. BFT databases ensure both concurrency and consistency in transaction processing with Byzantine faults.

HRDB [5] was the first proposed BFT database, utilizing a unique concurrency control method called Commit Barrier Scheduling, which is based on Strict 2PL [26], to ensure serializability. HRDB requires only $2f+1$ replicas to tolerate f Byzantine faulty nodes, but it relies on a component named *Shepherd* to cooperate with clients between replicas.

Byzantium [21] consists of $3f + 1$ replicas and several clients. It employs PBFT protocol for Byzantine fault-tolerance. Unlike HRDB, Byzantium does not depend on trusted components. However, it adopts Snapshot isolation, a weaker transaction isolation level than Serializable. Under this isolation level, it is known that write skew and read-only anomalies can occur [33, 34]. These anomalies may make the system vulnerable, especially in an environment where Byzantine faults can occur.

BFT-DUR [22] is also composed of $3f+1$ replicas and several clients. It guarantees serializability without relying on trusted components. BFT-DUR executes transactions on a single server and performs deferred-update replication, atomically broadcasting the results to other servers. Moreover, read-only transactions can be committed by executing them on a single server without communication between servers. However, BFT-DUR cannot tolerate clients that are Byzantine faulty.

MITRA [32] is also composed of $3f+1$ replicas and several clients. It guarantees serializability without relying on trusted components. MITRA uses a deferred-update replication technique. MITRA handles a single version of data items, avoiding using multiple versions to support a broader range of DBMSs.

2.2 Basil

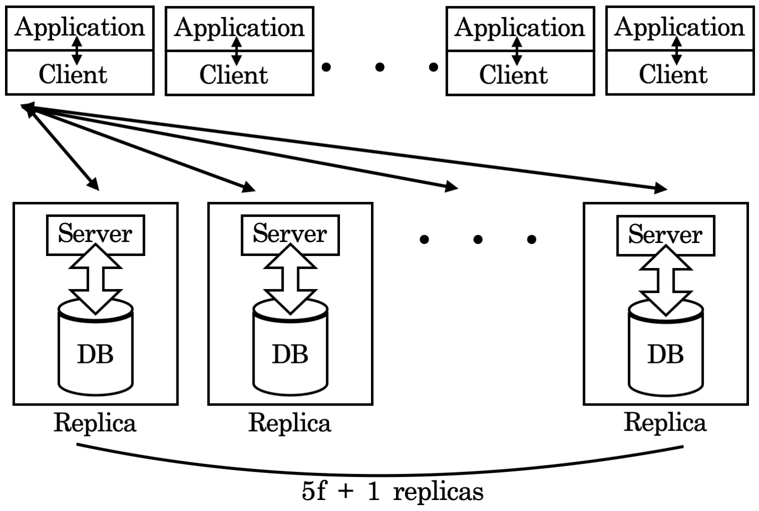


Figure 1: Basil architecture

Basil [3] is a state-of-the-art BFT database. Basil is a leaderless design BFT key-value store that guarantees serializability. Basil employs a consensus algorithm called Fast Byzantine Consensus [25]. Under certain conditions, the communication process between clients and replicas needed to commit transactions can be accomplished in a single round trip. However, Basil requires using $5f+1$ nodes to tolerate up to f Byzantine faulty nodes (Figure 1).

Basil stores data in the form of key-value pairs. The key is an identifier used to identify data uniquely. The value is the data associated with a key, and its content can be retrieved using the key. In Basil, operations manipulate these key-value pairs. A transaction is an indivisible set of some operations. Basil's transaction processing consists of three phases: Execution phase, Prepare phase, and Writeback phase (Figure 2).

Execution Phase Basil extends the multi-version timestamp ordering protocol (MVTSO) [29] to achieve both robustness against Byzantine faults and high transaction concurrency.

- **Begin()**: Basil's client assigns a timestamp $ts := (Time, ClientID)$ to transaction T. Basil protocol ensures serializability by using this timestamp for concurrency control in transactions.

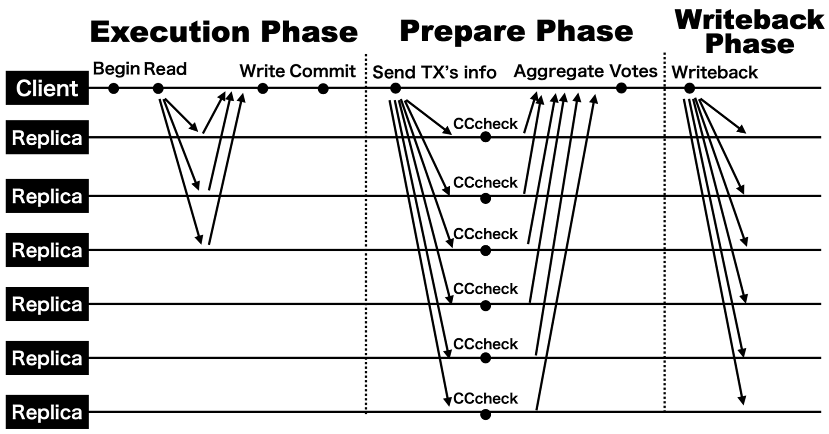


Figure 2: Basil transaction processing

- **Write(key, value)**: A client creates a new version of the key and assigns the value to this latest version. The client buffers the new version locally until the transaction has finished execution and reflects them to replicas during the protocol's Prepare phase. Once the version is created, the client adds a tuple consisting of the key and the newly developed version to $WriteSet_T$. $WriteSet_T$ is used to check the transaction's serializability during the CC-check (concurrency control check) in the Prepare phase.
- **Read(key)**: The client sends a *read request* message $m := \langle READ, key, ts_T \rangle$ to at least $2f + 1$ replicas. The replicas verify the *read request*; if deemed valid, they update the *Read Timestamp* for the key. Each replica selects a version in *committed* or *prepared* state with the maximum timestamp less than ts_T. The replica signs a message composed of that version and the version's metadata and replies to the client. The client waits for responses from at least $f + 1$ replicas and selects the version with the highest timestamp among the valid messages. The client adds a tuple consisting of the key and the selected version to $ReadSet_T$. If the selected version is not *committed* but *prepared*, the client adds a tuple to Dep_T composed of that version and the identifier of the transaction that created it. Both $ReadSet_T$ and Dep_T are used in the Prepare phase's concurrency control check, similar to $WriteSet_T$.
- **Commit()**: The client initiates the Prepare phase.

Prepare Phase The client sends a message to replicas containing the metadata of transaction T-ts_T, $WriteSet_T$, $ReadSet_T$, Dep_T, and hash values of these elements. Replicas perform the concurrency control check (Algorithm 1 in [3]) to check whether the transaction is serializable or not using the message elements. Based on the checking result, replicas sign a vote to commit or abort the transaction and reply to the client. The client verifies and aggregates the votes from replicas. If sufficient votes determine either to commit or abort the transaction, the client initiates the Writeback phase (Fast Path). If determining whether to commit or abort is impossible, the client communicates again with replicas to decide and begins the Writeback phase (Slow Path).

Writeback Phase The client notifies the application of the outcome of the transaction. The client sends a proof of the decision to replicas. In the case of commit, the client changes the state of the versions created by the transaction to *committed.*

2.3 Performance Bottlenecks

Using Basil as a case study, we illustrate the performance limitations of BFT databases. While Basil surpasses BFT SMR in performance, it falls short compared to Tapir [4], a distributed database designed for CFT. To analyze the reasons, we contrasted Basil's performance constraints with non-Byzantine fault-tolerant replicated systems. Two key factors contribute to its reduced efficiency.

Algorithm 1 CC-Check(T) cited from [3]

```
1: if $ts_T > localClock + \delta$ then
2:     return Vote-Abort
3: if $\exists$ invalid $dependency \in Dep_T$ then
4:     return Vote-Abort
5: for $\forall key, version \in ReadSet_T$ do
6:     if $version > ts_T$ then return MisbehaviorProof
7:     if $\exists T' \in Committed \cup Prepared : key \in WriteSet_{T'} \wedge version < ts_{T'} < ts_T$ then
8:         return Vote-Abort, $optional : (T', T'.\text{C-CERT})$
9: for $\forall key \in WriteSet_T$ do
10:     if $\exists T' \in Committed \cup Prepared :$
         $ReadSet_{T'}[key].version < ts_T < ts_{T'}$ then
11:         return Vote-Abort, $optional : (T', T'.\text{C-CERT})$
12:     if $\exists RTS \in key.RTS : RTS > ts_T$ then
13:         return Vote-Abort
14: $Prepared.add(T)$
15: wait $for\ all\ pending\ dependencies$
16: if $\exists d \in Dep_T : d.decision = Abort$ then
17:     $Prepared.remove(T)$
18:     return Vote-Abort
19: return Vote-Commit
```

The first factor is the need for cryptographic processing. In non-Byzantine fault-tolerant replicated systems, cryptographic processing is not necessary. On the other hand, Basil requires cryptographic processing in various scenarios because Basil needs to consider the possibility of external attacks. When replicas send messages to clients, they include a signature, and the client verifies them. When a transaction depends on another transaction, Basil validates the latter transaction to ensure it has not been fabricated. In the case of Basil's transaction processing, comparing the performance of signing and verifying messages with the performance of not executing these processes shows that the latter exhibits several times higher throughput [3]. Hence, cryptographic operations are considered a performance bottleneck in Basil's transaction processing.

The second factor is the need for more communication processing. Basil requires more communication processing than non-Byzantine fault-tolerant replicated systems because the former must consider the possibility of misbehavior of Byzantine faulty nodes. Considering the round trip time (RTT) between servers in a data center on the Google Cloud Platform, approximately 2 ms, and the RTT between servers in different data centers, which takes about 300 ms [24], we can see that the communication cost in geographically distributed systems becomes significantly expensive. In experiments with Basil, when comparing the performance of Basil with and without the Fast Path optimization, which reduces the round-trip count to 1 until commit, it was observed that introducing the Fast Path resulted in better performance [3]. Then, each time Basil executes a read operation, it must send read requests to a minimum of $2f + 1$ replicas and receive responses from at least $f + 1$ replicas to achieve Byzantine fault-tolerance. In experiments with Basil, varying the read quorum showed that as the quorum size increased, the performance degraded significantly [3]. Therefore, communication processes are considered a performance bottleneck in Basil's transaction processing.

3 Proposal: Cataphract

3.1 Concept of Cataphact

The previous section discussed cryptographic and communication processing as performance bottlenecks in BFT databases' transaction processing. We propose a novel batch processing method, Cataphract, to streamline these processes.

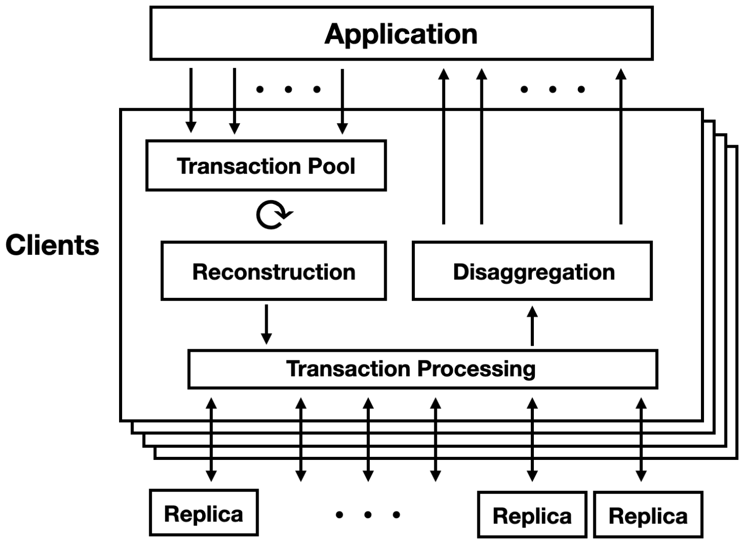


Figure 3: Flow of BFT database processing with Cataphract

BFT databases are composed of clients and replicas. When a client receives transaction requests from an application, the client stores the requests in the pool. In the conventional method, the client retrieves one transaction at a time from the pool, executes it, and then returns it to the application. In the BFT database with Cataphract (Figure 3), a client retrieves transactions repeatedly, disassembles these transactions into individual operations, and constructs a new consolidated transaction from these operations. After the client executes the transaction, it disaggregates it. Then, it returns the results of the original transaction units (before reconstruction) to the application. The number of original transactions that can be executed in a single transaction processing is increased by reconstructing transactions. Since cryptographic and communication processing occurs for each transaction and can be performed at the reconstructed transaction unit, the process becomes efficient.

The following explains why processing cryptographic and communication processes in larger units is more efficient than in smaller units. First, we explain cryptographic processing, which is divided into two parts: message signing and verification. The signing procedure consists of two parts: hashing the message and encrypting the obtained hash value. The latency of hashing the message is expressed by equation (1) [30]. The symbols used in equation (1) are presented in Table 1 [30].

Table 1: Explanation of the symbols in equation (1) cited from [30]

Symbol	Explanation
L	Latency of hash value generation
M	Input message size
B	Block size
I_{in}	Number of clock cycles to load a message
I_{out}	Number of clock cycles to store the hash value
I_{core}	Number of clock cycles to process hash function
I_{final}	Number of clock cycles for the finalization
f	Maximum clock frequency

$$L = \left(\frac{M}{B}(I_{in} + I_{core}) + I_{final} + I_{out} \right) \cdot \frac{1}{f} \tag{1}$$

Equation (1) reveals that even with an increase in the volume of data from the input messages, the counts of I_{in} and I_{core} may increase, but the counts of I_{final} and I_{out} remain unchanged. When comparing sending the same overall amount of data in small units N times versus sending it in a large unit once, sending in a large unit once reduces the counts of I_{final} and I_{out} to $\frac{1}{N}$. Similarly, the times the obtained hash value is encrypted are reduced to $\frac{1}{N}$. Since the hash value is of fixed length irrespective of the input data volume, the time taken for encryption per operation does not increase. Verification involves three processes: hashing the message again, decrypting the encrypted hash value, and comparing the results. Similar to the previous process, the hashing operation reduces the counts of I_{final} and I_{out} to $\frac{1}{N}$. For decryption and comparison, the count is also reduced to $\frac{1}{N}$, and the time does not increase due to the fixed hash value length. For the same amount of data, signing and verification are more efficient on larger data units than on smaller ones.

Second, we focus on latency in communication processing. The round trip time (RTT) between multiple servers is the sum of application, transmission, propagation, and queuing latency. As the input data volume increases, only the transmission delay among these factors increases latency. The transmission delay is affected by the amount of data because it is the time required to convert the data described by bits into electrical signals. When sending one large unit rather than N small units, the application delay, propagation delay, and queuing delay can be reduced to $\frac{1}{N}$. Therefore, sending a larger data unit is more efficient than sending the same amount in smaller units. Sending messages in large units is more efficient, especially in a wide-area network (WAN), where propagation delay accounts for most RTT.

From the above, it is more efficient to execute cryptographic and communication processes in larger units than in smaller units. Therefore, reconstructing a new transaction improves the performance of the BFT database because the communication and cryptographic processes can be executed in a larger unit.

3.2 Implementation of Cataphract

We applied Cataphract to Basil's transaction processing (Algorithm 2, Algorithm 3). Cataphract enhances the efficiency of processing in both the Execution phase and the Prepare phase of Basil. Given that efficiency improvements in these two processes are achieved through different means, we address each individually to provide a comprehensive understanding.

Our explanation begins with optimizations implemented in the Prepare phase. Basil's transaction processing requires the following operations in the Prepare phase. A client sends transaction information to all replicas. Replicas check whether to commit or abort the transaction, then sign and send the checking result to the client. Based on the replicas' messages, the client verifies the messages' signatures and decides whether to commit or abort the transaction. As the processing in the Prepare phase is carried out to commit the transaction, it is performed once per transaction in the vanilla Basil's transaction processing. Cataphract reduces the cryptographic and communication processes in the Prepare phase by reconstructing a single transaction from multiple transactions.

Next, We describe the optimizations implemented in the Execution phase. Basil conventionally processes operations synchronously to maintain data consistency, particularly when there are dependencies between operations. This synchronicity, while ensuring correctness, introduces latency. Our optimization strategy involves reordering operations during transaction reconstruction. This approach enables consecutive execution of homogeneous operations, facilitating asynchronous processing of read operations. This asynchronous execution of reads does not affect the results. Conversely, heterogeneous operations and write operations with potential dependencies are executed synchronously. By implementing this reordering technique, we significantly reduce operation wait times, thereby enhancing overall system efficiency.

Since changing the order of conflicting operations may alter their effects, it is essential to be cautious about reconstructing a transaction to ensure that conflicting transactions end up as different reconstruction targets (line 6, line 12 in Algorithm 2). Transactions not included in the current reconstruction process are automatically queued for the subsequent reconstruction cycle. This ensures that all transactions are eventually incorporated into reconstructed transactions. In the case of a write-write conflict, a newly constructed transaction involves both conflicting transactions because the order among operations of the same type is not modified when reordering.

Algorithm 2 Transaction Disaggregation in Cataphract

```
Input: batchSize                              ▷ The number of transactions used in reconstruction
Output: $ReadAfterWrite$, $WriteAfterRead$
Output: $ReadSet_{NewTx}$, $WriteSet_{NewTx}$
 1: function DisaggregateTransaction
 2:     for all transaction $\in$ transaction pool do
 3:         if loopCount $\geq$ batchSize then break
 4:         for all $op \in$ transaction do
 5:             if $op.type = Read$ then
 6:                 if $op \in WriteSet_{NewTx}$ then          ▷ Conflict with a prior loop transaction
 7:                     return
 8:                 if $op \in WriteSet_{Tx}$ then     ▷ Read-After-Write dependency in the transaction
 9:                     $ReadAfterWrite \leftarrow$ True
10:                 $ReadSet_{Tx} \leftarrow op$
11:             if $op.type = Write$ then
12:                 if $op \in ReadSet_{NewTx}$ then           ▷ Conflict with a prior loop transaction
13:                     return
14:                 if $op \in ReadSet_{Tx}$ then      ▷ Write-After-Read dependency in the transaction
15:                     $WriteAfterRead \leftarrow$ True
16:                 $WriteSet_{Tx} \leftarrow op$
17:         for all $op \in ReadSet_{Tx}$ do
18:             $ReadSet_{NewTx} \leftarrow op$
19:         for all $op \in WriteSet_{Tx}$ do
20:             $WriteSet_{NewTx} \leftarrow op$
21:         loopCount++
```

The reconstructed transaction basically executes all read operations after all write operations. The primary reason for this order is to shorten the intervals between read operations and the CC-Check. This approach leads to a lower probability of abort in the CC-Check compared to the reverse order.

When dealing with original transactions where read operations occur after write operations on the same data items, special handling is required (line 8 in Algorithm 2, line 2 in Algorithm 3). During the transaction reconstruction process, the client restructures the operation sequence, executing all read operations after completing write operations. This handling is essential because the subsequent read operations rely on the state modifications introduced by the preceding write operations. Conversely, when handling original transactions where write operations occur after read operations on the same data items, special handling is also required (line 14 in Algorithm 2, line 7 in Algorithm 3).

Reconstructing a new transaction from more transactions consistently leads to longer intervals between read operations and the CC-Check, resulting in a higher probability of abort than the vanilla Basil's transaction processing.

Algorithm 3 Transaction Reconstruction in Cataphract

```
Input: $ReadAfterWrite$, $WriteAfterRead$
Input: $ReadSet_{NewTx}$, $WriteSet_{NewTx}$
Output: $OpSet_{NewTx}$                                  ▷ Reconstructed transaction
 1: function ReconstructTransaction
 2:     if $ReadAfterWrite$ = True then                  ▷ Operations are ordered Read-After-Write
 3:         for all $op \in WriteSet_{NewTx}$ do
 4:             $OpSet_{NewTx} \leftarrow op$
 5:         for all $op \in ReadSet_{NewTx}$ do
 6:             $OpSet_{NewTx} \leftarrow op$
 7:     else if $WriteAfterRead$ = True then             ▷ Operations are ordered Write-After-Read
 8:         for all $op \in ReadSet_{NewTx}$ do
 9:             $OpSet_{NewTx} \leftarrow op$
10:         for all $op \in WriteSet_{NewTx}$ do
11:             $OpSet_{NewTx} \leftarrow op$
12:     else                                             ▷ Basically operations are ordered Read-After-Write
13:         for all $op \in WriteSet_{NewTx}$ do
14:             $OpSet_{NewTx} \leftarrow op$
15:         for all $op \in ReadSet_{NewTx}$ do
16:             $OpSet_{NewTx} \leftarrow op$
```

4 Evaluation

We evaluate Cataphract to answer the following questions:

Q1. How much does the performance of Basil improve when applying Cataphract?

Q2. How does batch size of Basil with Cataphract impact performance?

Q3. How much does the placement of nodes affect performance?

4.1 Experimental Setup

Table 2: m4.4xlarge instance specification.

Item	Specification
CPU	2.4 GHz Intel Xeon E5-2676 v3
CPUs	8
Memory	64(GiB)

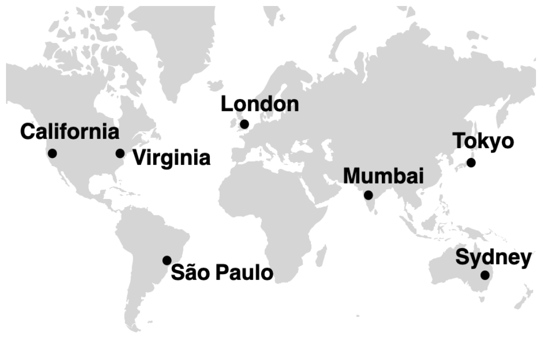


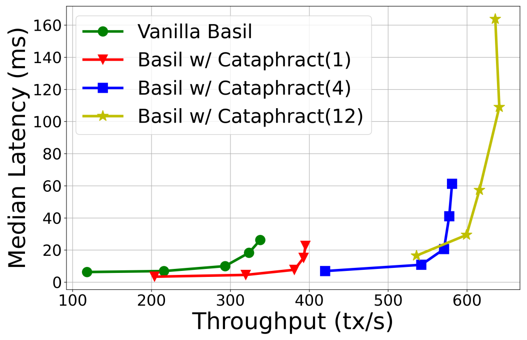
Figure 4: Placement of nodes in a geographically distributed environment.

To evaluate Cataphract, we applied Cataphract to Basil's implementation [16]. Our implementation is publicly available on GitHub [17]. We evaluated our system on Amazon Web Services (AWS). Each server was on an Amazon EC2 instance m4.4xlarge (Table 2). Basil was configured with one client node and six replica nodes ($f = 1$) for seven nodes. We ran the experiments within an availability zone (ap-northeast-1a), within a region (ap-northeast-1), and at the planet scale (Figure 4).

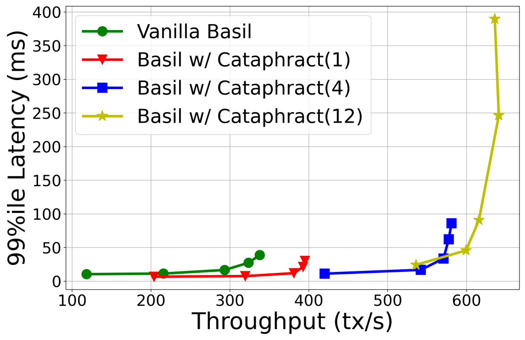
We used Yahoo ! Cloud Serving Benchmark (YCSB) [31] as the benchmark commonly used to evaluate NoSQL databases. YCSB-A represents a workload with a 50% read and 50% write operation ratio, YCSB-B consisted of 95% read and 5% write operation ratio, and YCSB-C consisted of 100% read operations and 0% write operations. Ed25519-donna [52] was used for the digital signature in all experiments. By default, a transaction had 10 operations, the dataset followed a uniform distribution, and the number of items in the database was 1 million. We ran experiments for 40 seconds (5 s warm-up/cool-down). We gradually increased the number of concurrent closed-loop clients from 1 to 12. Our performance metrics were throughput (tx/sec) and latency (ms). The latency included reconstructing transactions and reverting them to their original transaction units. Throughput and latency were the average of five measurements.

The discrepancy in Basil's performance results between the original Basil paper [3] and this paper can be attributed to differences in experimental setups. In Basil's original paper, experiments were conducted using 36 nodes (8-core 2.0 GHz CPU, 64 GiB RAM, 10 GB NIC, 0.15 ms ping latency), consisting of 18 clients and 18 replicas.

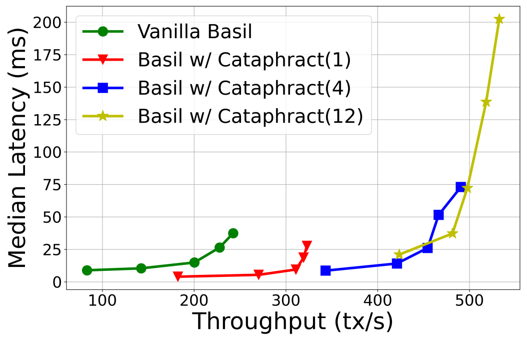
4.2 Basic Performance



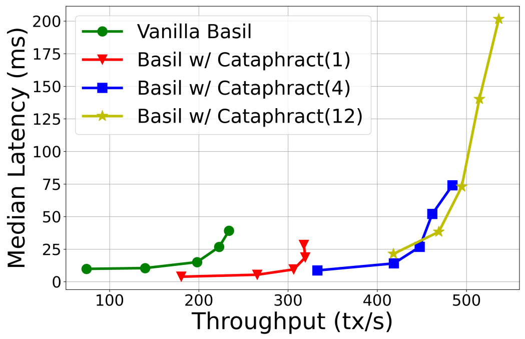
(a) Median Latency in YCSB-A.



(b) 99%ile Latency in YCSB-A.



(c) Median Latency in YCSB-B.



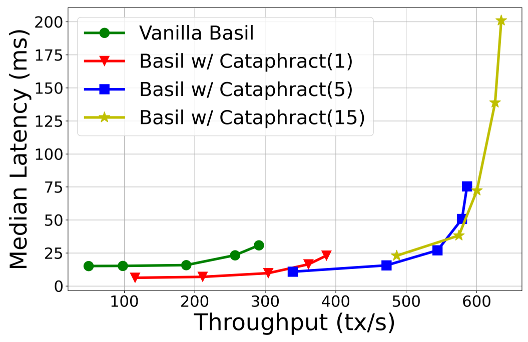
(d) Median Latency in YCSB-C.

Figure 5: Throughput vs. Latency for deployment within an availability zone.

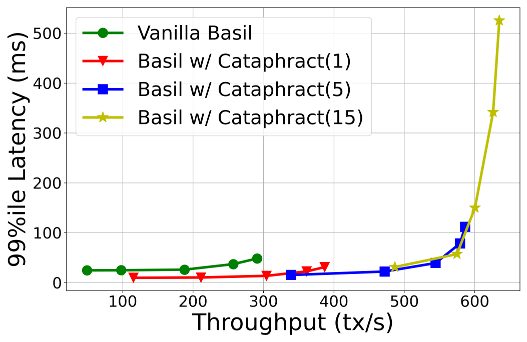
4.2.1 Evaluation in Intra-region Environment

First, we compared Basil with Cataphract against the vanilla Basil in an environment where all nodes were within a single availability zone (Figure 5). The numerical values enclosed in parentheses next to Cataphract represent the batch size, which refers to the number of transactions processed together as a single unit.

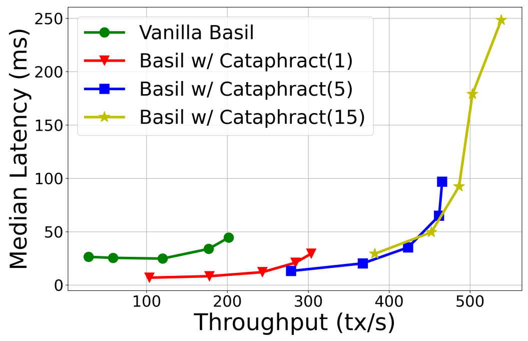
As can be seen from the results, Basil with Cataphract outperformed the original Basil in terms of throughput regardless of the batch size. There was a positive correlation between batch size and



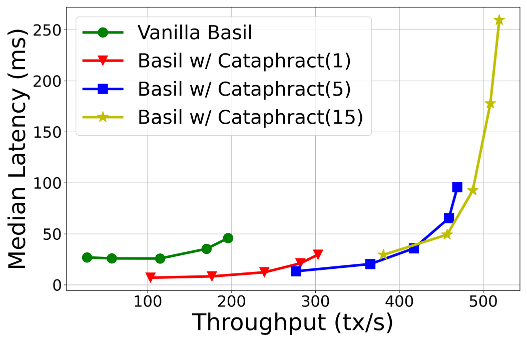
(a) Median Latency in YCSB-A.



(b) 99%ile Latency in YCSB-A.



(c) Median Latency in YCSB-B.



(d) Median Latency in YCSB-C.

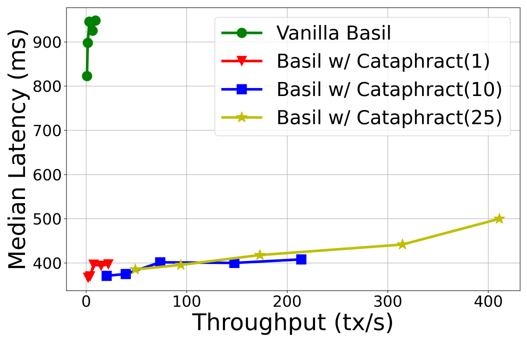
Figure 6: Throughput vs. Latency for deployment within a region.

throughput; as the batch size grew, the throughput consistently improved. The most significant performance difference was observed in YCSB-C workload. Basil with Cataphract measured approximately 5.6 times higher throughput than the baseline Basil when the number of clients was 1. When the batch size was large, the modest increase in throughput despite increasing client numbers was likely due to the number of requests exceeding the capacity of replica nodes to process them concurrently.

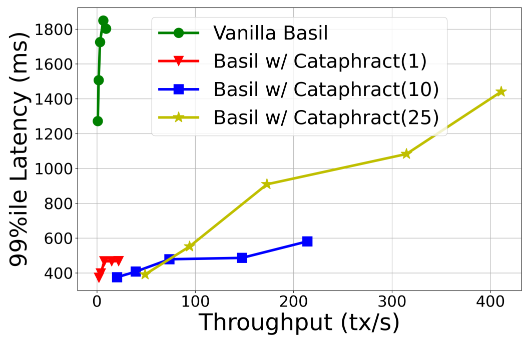
Basil with Cataphract demonstrated lower latency values than the vanilla Basil when operating with smaller batch sizes. Our experiments in YCSB-C benchmark demonstrated that, under minimal load conditions (1 client and 1 batch size), the proposed technique significantly improved performance by decreasing latency by 55% compared to the original Basil. The lower latency than the original Basil when the batch size was 1 was due to swapping the order of operations, which reduced the waiting time for communication due to read operations. Conversely, with larger batch sizes, we observed a steep increase in latency as the number of clients grew in every workload examined. The observed spike in latency could be attributed to the volume of requests surpassing the replica nodes' concurrent processing capacity. This latency increase aligned with our previous explanation for the limited throughput scalability, reinforcing the notion of system saturation.

The performance difference between Basil with Cataphract and the original Basil was most pronounced in YCSB-C (read-only workload). In the vanilla Basil, read operations required communication with replica nodes, while Cataphract reduced these overheads.

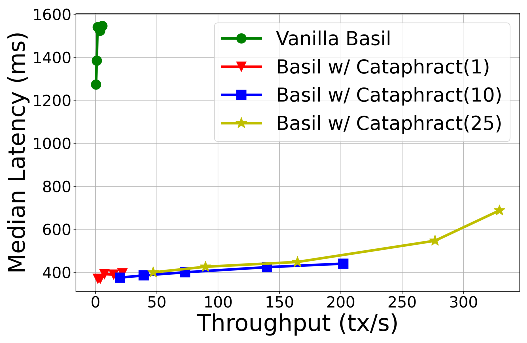
Next, we compared Basil with Cataphract against the vanilla Basil in an environment where nodes are across multiple availability zones (Figure 6). The overall shape of the graph remained similar to the experiments conducted within a single availability zone. However, the difference between Basil with Cataphract and the baseline Basil had become more pronounced. We attributed this phenomenon to the increased inter-node distances, which resulted in higher communication latencies. Consequently, the efficiency gains of our proposed method became more pronounced in this scenario.



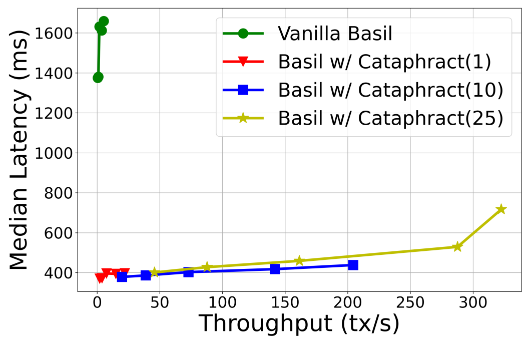
(a) Median Latency in YCSB-A.



(b) 99%ile Latency in YCSB-A.



(c) Median Latency in YCSB-B.



(d) Median Latency in YCSB-C.

Figure 7: Throughput vs. Latency for deployment in a geo-distributed environment.

In YCSB-C workload, Cataphract demonstrated significant performance enhancements. With a single client and a batch size of 12, we observed a peak throughput improvement of 8.6 times over the baseline. Concurrently, under minimal load conditions (1 client, 1 batch size), Basil with Cataphract reduced latency by 74% compared to the original Basil.

4.2.2 Evaluation in Geo-distributed Environment

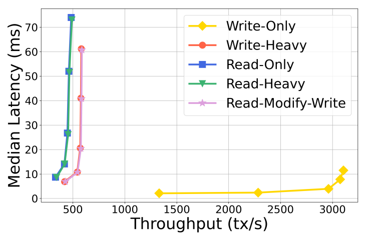
We compared Basil with Cataphract against the vanilla Basil in an environment where nodes were geographically distributed (Figure 7).

The results of this experiment revealed a pronounced deviation from the patterns observed in the two previously discussed experimental configurations. Of particular note was the substantially degraded performance exhibited by the original Basil, characterized by markedly reduced throughput and considerably elevated latency. These performance degradations could be primarily attributed to the prolonged communication times necessitated by the extensive geographical dispersion of nodes.

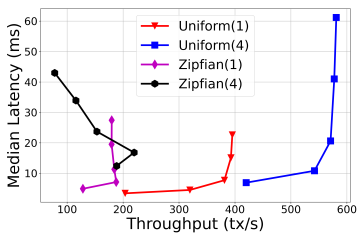
The geographically distributed environment highlighted Cataphract's strengths more prominently. Our experiments showed that Basil with Cataphract consistently achieved markedly superior performance metrics compared to the baseline Basil, both in terms of throughput and latency. A key distinction from our earlier findings in less distributed environments was the sustained performance advantage even at larger batch sizes. In this scenario, the Cataphract-enhanced Basil maintained lower latency than the vanilla Basil across all tested batch sizes.

The maximum performance difference in throughput was observed in YCSB-C when the number of clients was 1 and the batch size was 12, reaching approximately 80.4 times higher. Basil with Cataphract reduced latency by 76% relative to the original Basil for the YCSB-C workload with 12 clients and 1 batch size.

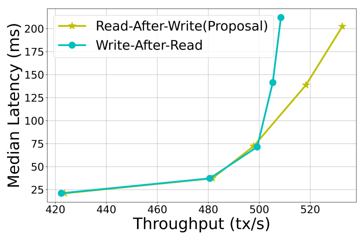
Our experiments provided empirical evidence of Cataphract's capacity to optimize throughput and latency in distributed systems. Our analysis revealed a clear correlation within the Cataphract



(a) Impact of workload.



(b) Impact of skew.



(c) Impact of ordering.

Figure 8: Performance analysis under varying system parameters.

framework: as batch size increased, we observed a corresponding improvement in throughput. However, this came with a trade-off in the form of increased latency.

A particularly noteworthy finding was the relationship between Cataphract's effectiveness and node distribution. While the method showed measurable benefits even in clusters with minimal geographical distribution, its impact became increasingly pronounced as the inter-node distances expanded.

4.3 Performance Analysis Under Varying System Parameters

4.3.1 Impact of Workload

Figure 8a demonstrated the impact of workload variations on the performance metrics of Basil with Cataphract (4). The workloads were defined as follows: Write-Only (100% writes), Write-Heavy (50% writes, 50% reads), Read-Only (100% reads), Read-Heavy (5% writes, 95% reads), and Read-Modify-Write (each operation reads then writes to the same key).

The experimental outcomes indicated a precise categorization of workload behaviors: read-modify-write and write-heavy workloads demonstrated similar performance trends, while read-only and read-heavy workloads also showed comparable patterns. The write-only workload exhibited substantially different characteristics, diverging significantly from these identified groups.

Performance degradation was directly correlated with the increase in the proportion of read operations due to the inherent communication processing and cryptographic operations associated with these reads.

4.3.2 Impact of Skew

Figure 8b demonstrated the impact of skew on the performance metrics of Basil with Cataphract. We compared cases where the data distribution followed a uniform distribution and cases with a Zipfian distribution with a skew of 0.9.

Our experiments revealed a significant performance decline when utilizing a Zipfian distribution with a skew factor 0.9. This performance degradation was primarily caused by an elevated rate of transaction aborts. This distribution's skewed data access pattern increased contention on frequently accessed data items, resulting in a higher probability of conflicting transactions and, consequently, more frequent transaction aborts.

The substantial performance deterioration observed when increasing the batch size from 1 to 4 could be attributed to the expanded number of operations within reconstructed transactions. This expansion resulted in extended intervals between read operations and CC-Check, subsequently elevating the probability of transaction aborts.

4.3.3 Impact of Operation Ordering

Figure 8c demonstrated the impact of operation ordering in a reconstructed transaction. The experimental configuration employed Basil with Cataphract (12) in YCSB-B.

The Read-After-Write sequence performed more effectively than the Write-After-Read sequence. This enhanced performance could be attributed to the reduced interval between read operations and CC-Check, consequently leading to a lower abort rate. The shorter the interval between the read operation and the CC-check, the lower the probability that the value read has been overwritten by other transactions and became stale when the CC-check occurred. The performance disparity became more evident as the number of clients increased due to the higher probability of transaction conflicts in high concurrency scenarios.

5 Related Work

5.1 BFT SMR

State machine replication (SMR) [46–48] approach has been widely studied as a solution to the classic problem, Byzantine Generals Problem. A state machine serves as a model for a series of server replicas working in concert to provide one or more services to a client. It typically consists of two primary components: variables and commands. The state variables represent the system's current state, while the commands represent transitions between different states. Byzantine fault-tolerant state machine replication (BFT SMR) [18–20, 28, 42–45] ensures that non-faulty replicas agree on the order of execution of operations from clients and that despite Byzantine replicas, non-faulty replicas execute the operations sequentially in the same order and generate the same responses to each operation. However, this approach is inefficient for commutative workloads as it processes operations sequentially for agreement and execution. Recently, considerable literature has grown around the theme of directed acyclic graph-based (DAG-based) BFT SMR [38–41]. In DAG-based BFT SMR, a single designated leader server is not involved when delivering transactions to each replica. Since the process depends on a leader server and is only for consensus formation, message propagation, and consensus formation are performed independently. DAG-based BFT SMR achieves substantially higher throughput compared to traditional BFT SMR approaches. This performance improvement is attributed to transaction pipelining, which facilitates parallel processing.

5.2 BFD database

Scalar DL [35], a Byzantine fault detection database (BFD database) can detect Byzantine faults and execute transactions concurrently. While the BFD database cannot continue providing services unaffected in the event of a Byzantine fault, it can detect the occurrence of such faults. BFD database can operate in smaller administrative domains [36, 37] compared to the BFT database. Managing multiple domains within a single organization can be challenging, so BFD database is more practical when dealing with Byzantine faults in a single organizational context.

6 Conclusions

This paper introduces Cataphract, an innovative batch processing method tailored for Byzantine fault-tolerant (BFT) databases. Our research delved into the performance limitations of BFT databases, highlighting cryptographic operations and communication processes as the primary bottlenecks. Cataphract's novel approach involves breaking down multiple transactions into their constituent operations and reconstructing transactions from them. This strategy aims to streamline the communication and cryptographic processes typically associated with individual transactions. To evaluate Cataphract's efficacy, we integrated it into Basil, a cutting-edge BFT database, and conducted comprehensive experiments. The experimental results indicate that Cataphract enhances Basil's performance. Our findings reveal a trade-off between throughput and latency as batch size increases. Moreover, we observed that the efficacy of Cataphract becomes more significant with greater inter-node distances.

In future work, we will focus on the application of Cataphract to other BFT databases to further validate its effectiveness and explore its potential in various contexts.

Acknowledgment

This paper is based on results obtained from the project ”Research and Development Project of the Enhanced Infrastructures for Post-5G Information and Communication Systems (JPNP20017) “ and JPNP16007 commissioned by the New Energy and Industrial Technology Development Organization (NEDO), and from JSPS KAKENHI Grant Number 22H03596, and from JST CREST Grant Number JPMJCR24R4 and from SECOM Science and Technology Foundation.

References

[1] S. Liu, P. Viotti, C. Cachin, V. Quema, M. Vukolic. 2016. XFT: Practical Fault Tolerance beyond Crashes. USENIX Symposium on Operating Systems Design and Implementation. 485-500.

[2] A. Barger, Y. Manevich, H. Meir, Y. Tock. 2021. A Byzantine Fault-Tolerant Consensus Library for Hyperledger Fabric. IEEE International Conference on Blockchain and Cryptocurrency. 1-9.

[3] F. Suri-Payer, M. Burke, Z. Wang, Y. Zhang, L. Alvisi, and N. Crooks. 2021. Basil: Breaking up BFT with ACID (Transactions). In Proceedings of the ACM Symposium on Operating Systems Principles. 1–17.

[4] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. K. Ports. 2015. Building consistent transactions with inconsistent replication. In Proceedings of the ACM Symposium on Operating Systems Principles. 263–278.

[5] B. Vandiver, H. Balakrishnan, B. Liskov, and S. Madden. 2007. Tolerating Byzantine Faults in Transaction Processing Systems Using Commit Barrier Scheduling. In Proceedings of the ACM Symposium on Operating Systems Principles. 59–72.

[6] A. Sharma, J. Jiang, P. Bommannavar, B. Larson, J. Lin. 2016. GraphJet: Real-Time Content Recommendations at Twitter. In Proceedings of the VLDB Endowment. 1281–1292.

[7] P. Venetis, Alon Halevy, J. Madhavan, M. Paşca, W. Shen, F. Wu, G. Miao, C. Wu. 2011. Recovering Semantics of Tables on the Web. In Proceedings of the VLDB Endowment. 528–538.

[8] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. Decentralized Business Review (2008), 21260.

[9] K. Fan, S. Wang, Y, Ren, H, Li, Y, Yang. 2018. MedBlock: Efficient and Secure Medical Data Sharing Via Blockchain. J Med Syst.

[10] IBM. Transform cross-border payments with IBM blockchain world wire. `https://www.ibm.com/blockchain/solutions/worldwire`.

[11] Maciel M Queiroz, Renato Telles, and Silvia H Bonilla. 2019. Blockchain and supply chain management integration: a systematic review of the literature. Supply Chain Management: An International Journal (2019).

[12] Yanling Chang, Eleftherios Iakovou, and Weidong Shi. 2020. Blockchain in global supply chains and cross border trade: a critical synthesis of the state-of-the-art, challenges and opportunities. International Journal of Production Research 58, 7 (2020), 2082–2099.

[13] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J.J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, D. Woodford. 2012. Spanner: Google's Globally-Distributed Database. In 10th USENIX Symposium on Operating Systems Design and Implementation. 261-264.

[14] D. Huang, Q. Liu, Q. Cui, Z. Fang, X. Ma, F. Xu, L. Shen, L. Tang, Y. Zhou, M. Huang, W. Wei, C. Liu, J. Zhang, J. Li, X. Wu, L. Song, R. Sun, S. Yu, L. Zhao, N. Cameron, L. Pei, X. Tang. 2020. TiDB: a Raft-based HTAP database. In Proceedings of the VLDB Endowment. 3072–3084.

[15] M. Elhemali, N. Gallagher, N. Gordon, J. Idziorek, R. Krog, C. Lazier, E. Mo, A. Mritunjai, S. Perianayagam, T. Rath, S. Sivasubramanian, J. C. Sorenson III, S. Sosothikul, D. Terry, A. Vig. Amazon DynamoDB: A Scalable, Predictably Performant, and Fully Managed NoSQL Database Service. In Proceedings of the 2022 USENIX Annual Technical Conference. 1037-1048.

[16] Code of Basil. `https://github.com/fsuri/Basil_SOSP21_artifact`.

[17] Code of Proposed method to Basil. `https://github.com/aoikida/Basil_with_ReTx`.

[18] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham. 2019. HotStuff: BFT consensus with linearity and responsiveness. In Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC). 347–356.

[19] A. Bessani, J. Sousa, and E. E. Alchieri. 2014. State machine replication for the masses with BFT-SMaRt. In Proceedings of the International Conference on Dependable Systems and Networks (DSN). 355–362.

[20] M. Castro, B. Liskov. 1999. Practical Byzantine fault tolerance. In Proceedings of the USENIX Symposium on Operating Systems Design and Implementation. 173–186.

[21] R. Garcia, R. Rodrigues, and N. Preguiça. 2011. Efficient Middleware for Byzantine Fault Tolerant Database Replication. In Proceedings of the ACM European Conference on Computer Systems (EuroSys). 107–122.

[22] F. Pedone and N. Schiper. 2012. Byzantine fault-tolerant deferred update replication. Journal of the Brazilian Computer Society. 3–18.

[23] D. Ongaro, J. Ousterhout. 2014. In search of an understandable consensus algorithm. In USENIX Annual Technical Conference. 305–320.

[24] H. Zare, V. R. Cadambe, B. Urgaonkar, C. Sharma, Praneet Soni, N. Alfares, A. Merchant. 2022. LEGOStore: A Linearizable Geo-Distributed Store Combining Replication and Erasure Coding. In Proceedings of the VLDB Endowment. 2201-2215.

[25] J.-P. Martin, L. Alvisi. Fast Byzantine consensus. 2005. International Conference on Dependable Systems and Networks (DSN'05). 402-411.

[26] G. Weikumand, G.Vossen. 2001. Transactional Information Systems : Theory, Algorithms, and the Practice of Concurrency Control and Recovery. Morgan Kaufmann Publishers Inc.

[27] L. Lamport, R. E. Shostak, M. C. Pease. 1982. The Byzantine Generals Problem. ACM Trans. Program. Lang. Syst. 382–401.

[28] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, E. L. Wong. 2009. Zyzzyva: Speculative Byzantine Fault Tolerance. ACM Trans. Comput. Syst. 1–39.

[29] P. A. Bernstein and N. Goodman. 1983. Multiversion concurrency control-theory and algorithms. ACM Transactions on Database Systems (TODS). 465–483.

[30] M. Knezevic, K. Kobayashi, J. Ikegami, S. Matsuo, A. Satoh, Ü. Kocabas, J. Fan, T. Katashita, T. Sugawara, K. Sakiyama, I. Verbauwhede, K. Ohta, N. Homma, T. Aoki. 2012. Fair and Consistent Hardware Evaluation of Fourteen Round Two SHA-3 Candidates. IEEE Transactions on Very Large Scale Integration (VLSI) Systems. 827-840.

[31] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In Proceedings of the ACM Symposium on Cloud Computing (SOCC). 143–154.

[32] A. F. Luiz, L. C. Lung, M. Correia. 2011. Byzantine fault-tolerant transaction processing for replicated databases. In Proceedings of the IEEE International Symposium on Network Computing and Applications (NCA). 83-90.

[33] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, P. O'Neil. 1995. A critique of ansi SQL isolation levels. In Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data. 1–10.

[34] A. Fekete, E. O'Neil, P. O'Neil. 2004. A read-only transaction anomaly under snapshot isolation. ACM SIGMOD Record. 12-14.

[35] H. Yamada, J. Nemoto. 2022. Scalar DL: Scalable and Practical Byzantine Fault Detection for Transactional Database Systems. In Proceedings of the VLDB Endowment (PVLDB). 1324-1336.

[36] T. Distler. 2021. Byzantine Fault-Tolerant State-Machine Replication from a Systems Perspective. ACM Comput. Surv. 38 pages.

[37] J. LiandD. Maziéres. 2007. Beyond One-Third Faulty Replicas in Byzantine Fault Tolerant Systems. In Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI).

[38] I. Keidar, E. Kokoris-Kogias, O. Naor, A. Spiegelman. 2021. All You Need is DAG. Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC). 165–175.

[39] G. Danezis, L. Kokoris-Kogias, A. Sonnino, and A. Spiegelman. 2022. Narwhal and Tusk: a DAG-based mempool and efficient BFT consensus. In Proceedings of the ACM European Conference on Computer Systems (EuroSys). 34–50.

[40] A. Spiegelman, N. Giridharan, A. Sonnino, and L. Kokoris-Kogias. 2022. Bullshark: DAG BFT Protocols Made Practical. In Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS). 2705–2718.

[41] A. Spiegelman, B. Aurn, R. Gelashvili, Z. Li. 2023. Shoal: Improving DAG-BFT Latency And Robustness. arXiv preprint arXiv:2306.03058 (2023).

[42] A. Miller, Y. Xia, K. Croman, E. Shi, D. Song. 2016. The honey badger of BFT protocols. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. 31–42.

[43] G. G. Gueta, I. Abraham, S. Grossman, D. Malkhi, B. Pinkas, M. Reiter, D-A. Seredinschi, O. Tamir, A. Tomescu. 2019. SBFT: a scalable and decentralized trust infrastructure. In 2019 49th Annual IEEE/IFIP international conference on dependable systems and networks (DSN). IEEE, 568–580.

[44] P-L. Aublin, S. B. Mokhtar, V. Quéma. 2013. Rbft: Redundant byzantine fault tolerance. In 2013 IEEE 33rd International Conference on Distributed Computing Systems. IEEE, 297–306.

[45] T. Crain, V. Gramoli, M. Larrea, M. Raynal. 2018. DBFT: Efficient leaderless Byzantine consensus and its application to blockchains. In 2018 IEEE 17th International Symposium on Network Computing and Applications (NCA). IEEE, 1–8.

[46] N. A. Lynch, M. J. Fischer. 1981. On describing the behavior and implementation of distributed systems. Theoretical Computer Science 13, 1 (1981), 17–43.

[47] F. B. Schneider. 1990. Implementing fault-tolerant services using the state machine approach: A tutorial. ACM Computing Surveys (CSUR) 22, 4 (1990), 299–319.

[48] S. Owicki, L. Lamport. 1982. Proving liveness properties of concurrent programs. ACM Transactions on Programming Languages and Systems (TOPLAS) 4, 3 (1982), 455–495.

[49] G. Huang, X. Cheng, J. Wang, Y. Wang, D. He, T. Zhang, F. Li, S. Wang, W. Cao, Q. Li. 2019. X-engine: An optimized storage engine for large-scale e-commerce transaction processing. in Proc. Int. Conf. Manag. Data, Jun. 651–665.

[50] I. Arapakis, X. Bai, and B. B. Cambazoglu, "Impact of response latency on user behavior in web search," in Proceedings of the 37th international ACM SIGIR conference on Research & development in information retrieval, 2014, pp. 103–112.

[51] E. Schurman, J. Brutlag. Performance related changes and their user impact. In Velocity – Web Performance and Operations Conf. 2009

[52] Code of ed25519-donna. `https://github.com/floodyberry/ed25519-donna`.

[53] R. Wang, X. Wang, W. Yang, S. Yuan, Z. Guan. 2022. Achieving fine-grained and flexible access control on blockchain-based data sharing for the Internet of Things. China Communications, vol. 19, no. 6, pp. 22–34.

[54] W. -T. Tsai, R. Blower, Y. Zhu, L. Yu. 2016. A System View of Financial Blockchains. IEEE Symposium on Service-Oriented System Engineering (SOSE). 450-457.

[55] F. Chan, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A.Fikes, R. E. Gruber. 2006. Bigtable: A Distributed Storage System for Structured Data. IN 7th USENIX Symposium on Operating Systems Design and Implementation. 205-218.

[56] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov D. Petrov, L. Puzar, Y. J. Song, V. Venkataramani. 2013. TAO: Facebook's Distributed Data Store for the Social Graph. In USENIX Annual Technical Conference. 49-60.

[57] Pramod Bhatotia, Alexander Wieder, Rodrigo Rodrigues, Flavio Junqueira, and Benjamin Reed. 2010. Reliable data-center scale computations. In Proceedings of the 4th International Workshop on Large Scale Distributed Systems and Middleware (LADIS). 1–6.

[58] Diogo Behrens, Marco Serafini, Sergei Arnautov, Flavio P. Junqueira, and Christof Fetzer. 2015. Scalable error isolation for distributed systems. In Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation (NSDI). 605–620.

[59] Sushant Mane, Fangmin Lyu, and Benjamin Reed. 2023. Verify, And Then Trust: Data Inconsistency Detection in ZooKeeper. In Proceedings of the 10th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC). 16–22.

[60] Haoze Wu, Jia Pan, Peng Huang. 2024. Efficient Exposure of Partial Failure Bugs in Distributed Systems with Inferred Abstract States. In Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI). 1267-1283.