The Block, Chained: Hyperledger Fabric Is Much Slower Than Redis as a Distributed Database

Mohamad Hadi Ajami
2227105
Département de génie informatique et de génie logiciel
Polytechnique Montréal
Montréal, Canada
mohamad-hadi.ajami@polymtl.ca

Roberto Molina
2317239
Département de génie informatique et de génie logiciel
Polytechnique Montréal
Montréal, Canada
roberto.molina@polymtl.ca

Thomas Lusignan
2078987
Département de génie informatique et de génie logiciel
Polytechnique Montréal
Montréal, Canada
thomas.lusignan@polymtl.ca

Xi-Zhen Wang
2313663
Département de génie informatique et de génie logiciel
Polytechnique Montréal
Montréal, Canada
xi-zhen.wang@polymtl.ca

***Abstract*—Modern applications use a large amount of data that needs to be constantly available. Distributed databases are a solution to these constraints of size and availability. We compared the throughput and latency performance, as well as the resource utilization, of two types of NoSQL distributed databases: Redis, an in-memory key-value store; and Hyperledger Fabric, a private, permissioned blockchain. Four 1000-operation workloads with varying proportions of reads and writes were tested on both databases. In each of them, Redis had higher throughput and lower latency, and completed the workloads faster. Blockchain features such as security and privacy were not measured.**

Keywords— Hyperledger Fabric, Redis, Blockchain, NoSQL, Databases, Throughput, Performance

I. INTRODUCTION

A. *Why use distributed databases?*

In the landscape of modern applications, the utilization of distributed databases has become imperative owing to several compelling reasons, each addressing pivotal constraints and requisites of contemporary data-driven environments.[8]

B. *Distributed database features and characteristics*

Here are the features and characteristics of distributed database:

1. *Scalability*: Distributed databases provide a horizontal scaling capability by dispersing data across multiple nodes or servers. This empowers these databases to effortlessly accommodate larger data volumes and increased concurrent transactions or users.
2. *High Availability*: Ensuring uninterrupted access to data is a paramount concern, and distributed databases achieve this by replicating data across multiple nodes. Even in the event of node failures, these databases sustain operational functionality, thereby averting downtime and data loss.
3. *Fault Tolerance*: Leveraging data distribution across multiple nodes, distributed databases exhibit heightened resilience to hardware failures or network disruptions. Redundancy mechanisms and data replication protocols fortify data integrity and continuity.
4. *Performance Optimization*: The distribution of data across diverse nodes can significantly enhance query performance by facilitating parallel processing and alleviating the burden on individual nodes.
5. *Geographic Distribution*: Distributed databases span across geographical locations, enabling swift and low-latency access to data for users dispersed across diverse regions.
6. *Consistency and Concurrency*: Equipped with robust mechanisms, these databases ensure data consistency and adeptly handle concurrent access, upholding integrity and reliability in multi-user environments.
7. *Support for Large Workloads*: As data volume burgeons and the need for real-time processing escalates, distributed databases provide a robust infrastructure to efficiently manage large and intricate workloads.

C. *Objectives and contributions of this work*

The primary objectives of this study were to evaluate and compare the performance metrics of two distinct types of NoSQL distributed databases—Redis, an in-memory key-value store, and Hyperledger Fabric, a private, permissioned blockchain. The contributions of this work are as follows:

1. *Performance Evaluation*: Conducting a comparative analysis of throughput, latency, and resource

utilization between Redis and Hyperledger Fabric under various workloads. The aim was to discern the operational efficiency and effectiveness of these databases in handling different read and write proportions within 1000-operation workloads.

2. *Comparative Metrics*: Assessing the speed, efficiency, and resource utilization patterns of Redis and Hyperledger Fabric in terms of throughput, latency, and workload completion times. This comparative analysis provides valuable insights into the relative strengths and weaknesses of each database system.
3. *Limitations and Scope*: The work acknowledges the limitations of the study, notably the exclusion of certain blockchain features such as security and privacy metrics. This acknowledgment contributes to delineating the scope of the comparative analysis, paving the way for future research directions to explore these unmeasured aspects.

II. BACKGROUND

This section introduces the pivotal technologies in focus: Hyperledger Fabric, a private blockchain framework, and Redis, an in-memory key-value store. It provides essential insights into their features, functionalities, and intended applications, laying the groundwork for a comparative analysis based on their distinct attributes and use cases.

A. Hyperledger Fabric: a private blockchain

Hyperledger Fabric stands as a prominent permissioned blockchain framework developed under the Linux Foundation's Hyperledger project. It differs significantly from public blockchains, such as Bitcoin or Ethereum, by offering a permissioned network model, making it suitable for enterprise applications requiring confidentiality, scalability, and high performance.[7]

Key Features and Characteristics:

1. *Permissioned Network*: Hyperledger Fabric operates within a permissioned network, where participants must be authenticated and granted specific access rights. This feature enhances privacy and control, crucial for businesses and consortia.
2. *Modular Architecture*: Fabric's architecture allows pluggable components for consensus, membership services, and smart contract execution. This modularity ensures flexibility and adaptability to diverse enterprise requirements.
3. *Smart Contracts (Chaincode)*: Fabric employs smart contracts, referred to as "chaincode," which execute business logic and enforce rules within the network. Chaincode supports various programming languages, increasing accessibility and developer adoption.
4. *Scalability and Performance*: Fabric's design focuses on scalability by supporting parallel execution of transactions and leveraging channels to enable private communication among subsets of network members. This architecture enhances throughput and performance.
5. *Data Privacy and Confidentiality*: Fabric incorporates private channels and off-chain data storage options, ensuring sensitive information remains accessible only to authorized parties.
6. *Consensus Mechanisms*: Fabric offers pluggable consensus mechanisms, allowing network participants to select a consensus algorithm that best suits their use case, balancing performance and trust requirements.
7. *Enterprise Integration*: Fabric provides interoperability with existing enterprise systems through a variety of SDKs (Software Development Kits) and standard integration mechanisms, facilitating smoother adoption within enterprise environments.

B. Redis: an in-memory key-value store

Redis, an acronym for Remote Dictionary Server, is an open-source, advanced in-memory key-value store. It is renowned for its speed, flexibility, and versatility in handling various data structures. Originally developed by Salvatore Sanfilippo, Redis has gained widespread adoption due to its performance characteristics and rich feature set.[6]

Key Features and Characteristics:

1. *In-Memory Data Storage*: Redis primarily stores data in memory, allowing for exceptionally fast read and write operations. This design choice makes it ideal for use cases that demand high-speed data access.
2. *Key-Value Data Model*: Redis employs a simple but powerful key-value data model, allowing storage and retrieval of data using keys as references. It supports various data types like strings, lists, sets, hashes, sorted sets, and more, enabling diverse data storage capabilities.
3. *Persistence Options*: While Redis primarily operates in-memory, it offers persistence options like snapshots and append-only files (AOF) to store data on disk, ensuring durability and data recovery in case of failures or restarts.
4. *Pub/Sub Messaging*: Redis includes a publish/subscribe messaging paradigm, enabling communication between different parts of an application or between multiple applications. This feature is beneficial for building real-time applications and event-driven architectures.
5. *Advanced Functionality*: Redis offers a rich set of commands and functionalities, including atomic operations, transactions, Lua scripting, and data eviction policies, which contribute to its flexibility and applicability across diverse use cases.
6. *High Performance*: Due to its in-memory architecture and optimized data structures, Redis excels in

delivering high throughput and low-latency performance, making it suitable for applications demanding real-time responsiveness.

7. *Use Cases*: Redis finds application in various scenarios such as caching, session management, message brokering, leaderboards, and real-time analytics where rapid access to data is crucial.
8. *Community and Ecosystem*: Redis benefits from an active community and a vast ecosystem of libraries, tools, and clients supporting multiple programming languages, enhancing its usability and integration capabilities.

C. How they compare

When comparing Hyperledger Fabric, a private blockchain framework, with Redis, an in-memory key-value store, several distinct aspects differentiate the two technologies, especially concerning their intended use cases, architecture, and functionalities:

1) Architecture and Purpose:
    a) *Hyperledger Fabric*: Built for enterprise-grade solutions, Hyperledger Fabric offers a permissioned blockchain infrastructure emphasizing confidentiality, scalability, and modularity. It focuses on supporting complex business logic through smart contracts while providing robust privacy features.
    b) *Redis*: Primarily an in-memory data store, Redis is designed for high-speed data access and manipulation. It excels in performance-critical applications that demand rapid read and write operations, offering a versatile key-value data model.
2) Data Model and Functionality:
    a) *Hyperledger Fabric*: Stores immutable transactional data across a network of nodes, enabling decentralized and secure data handling. Its main focus lies in executing smart contracts and ensuring data integrity across a distributed ledger.
    b) *Redis*: Offers a flexible key-value data model with support for various data structures, suitable for caching, session storage, pub/sub messaging, and real-time analytics. It emphasizes speed and efficiency in data retrieval and manipulation.
3) Consensus Mechanisms:
    a) *Hyperledger Fabric*: Provides pluggable consensus mechanisms, enabling network participants to choose consensus algorithms suitable for their use cases. It emphasizes achieving agreement among nodes while maintaining confidentiality.
    b) *Redis*: Does not inherently include a consensus mechanism as it operates primarily as an in-memory data store and does not involve decentralized consensus among nodes.
4) Security and Permissioning:
    a) *Hyperledger Fabric*: Incorporates robust security measures such as permissioned network access, cryptographic techniques, and private channels, ensuring data privacy and confidentiality.
    b) *Redis*: Offers access control features but is not inherently built for complex permissioning or cryptographic operations comparable to a blockchain framework like Hyperledger Fabric.
5) Use Cases:
    a) *Hyperledger Fabric*: Suited for applications requiring a shared, tamper-proof ledger among multiple entities, especially in industries like finance, supply chain, and healthcare where data confidentiality and trust are crucial.
    b) *Redis*: Ideal for applications that demand high-speed data caching, real-time analytics, session storage, and other scenarios where rapid data access and manipulation are critical.

III. EXPERMIMENT

This section provides a comprehensive overview of the steps taken to configure the experimental setup.

Our examination begins with an in-depth exploration of the deployment of the two databases. Subsequently, we delve into an analysis of the diverse workloads employed in the experiment. Finally, we examine the configuration of the benchmarks, completing our comprehensive overview of the infrastructure and experiment setup.

A. Deployment of Hyperledger Fabric

The configuration used in Hyperledger Fabric was already defined by Caliper. The configuration is specified in the fabric-go-tls-solo.yml, that can be observed in their GitHub [2]. This configuration uses the Hyperledger Fabric version 1.4.4, employing the Raft consensus algorithm for the orderer and utilizing the CouchDB database. It operates on a single host, with the network composed of two organizations, each contributing a single peer. A dedicated channel is established, incorporating four distinct smart contracts: marbles, drm, simple, and smallbank, all implemented in the Golang programming language.

Both organizations and their peers are equipped with a Certificate Authority (CA) to ensure secure communication between network components. TLS encryption is implemented for all communication channels within the network, bolstering data confidentiality and integrity.

The infrastructure we use was a virtual machine (VM) equipped with dual-core processing capabilities and 4GB of RAM, running on Ubuntu 21.04. In this environment we

installed the benchmark Caliper which, as we mentioned before, contained the Hyperledger Fabric.

B. Deployment of Redis DB

To configure the Redis database, we initially had to install Docker, Docker compose. Once the prerequisites were installed, we employed docker-compose to configure the database. The corresponding docker-compose.yml file with the configuration can be found on our GitHub repository [1]. The established database comprised four nodes: one master and three replicas. This configuration was implemented to replicate a real network, where they ensure redundancy and fault tolerance. The open port for the Redis was 6379.

To ensure a more meaningful comparison between Redis and Hyperledger Fabric, we maintained uniformity in the infrastructure. We employed the same setup that was utilized for Hyperledger Fabric, comprising a virtual machine (VM) with dual-core processing capabilities and 4GB of RAM. The VM operated on the Ubuntu 21.04 operating system. This consistency in infrastructure allows us to draw more insightful insights when evaluating the performance of these two databases. In this infrastructure we installed Redis with the benchmark YCSB.

C. Description of the workloads

During the experiment, we delineated distinct workloads to systematically examine the disparities between the two databases. The investigation will center on two primary types of operations—reading and writing—since these are the only operations that can be done in a blockchain database. Across all workloads, a consistent set of 1000 requests will be executed, with the percentage of reads/writes varying. The ensuing table presents a comprehensive overview of the diverse workloads under consideration:

TABLE I. WORKLOADS EMPLOYED

Workload name	**Proportion of operations**	
	Reads	***Writes***
Read-only workload	100 %	0 %
Balanced workload	50 %	50 %
Write-oriented workload	10 %	90 %
Write-only workload	0 %	100 %

D. Setup of the Benchmark tools

1) Setup for YCSB

The YCSB setup was conducted on the same virtual machine that housed the Redis database. Initial preparations involved the installation of essential tools such as Java, Docker, and Python 2.7. Once all prerequisites were in place, we proceeded to download the YCSB version 0.17.0.

Executing a workload on the Redis database involved running the following commands:

```
./bin/ycsb.sh load redis -s -P  workload -p "redis.host=ip" -p "redis.port=6379"
./bin/ycsb.sh run redis -s -P workload -p "redis.host=ip" -p "redis.port=6379"
```

The first command initializes the workload, while the second one executes it. The specifics of the workload to be executed

```
recordcount=1000
operationcount=1000
workload=site.ycsb.workloads.CoreWorkload

readallfields=true

readproportion=0.1
updateproportion=0
scanproportion=0
insertproportion=0.9
```

Fig. 1: Write-oriented (10/90) workload file for YCSB

must be outlined in the workload file. Additionally, it's imperative to specify the IP address and port of the Redis database for successful execution.

Fig. 1 presents a workload file for YCSB. In this figure, the parameter "operationcount" denotes the total number of operations slated for execution, while the term "xproportion" signifies the proportion of the “x” operation within the workload. In our specific case, the workload exclusively incorporates read and insert operations, aligning with the rationale elucidated in the dedicated section on workloads. All the workload files can be observed on our GitHub [1].

2) Setup for Hyperledger Caliper

As Caliper included the Hyperledger Worker, both were installed in the same VM. To utilize Caliper, we had to install essential tools such as Node.js (version 12.22.12), npm (version 6.14.16), Python 2.7, and Docker. The comprehensive sequence for executing a workload with Caliper is detailed in our GitHub repository [1]. Once the setup was complete, the following command needed to be executed to initiate a workload:

sudo npx caliper launch manager --caliper-workspace . --caliper-benchconfig benchmarks/samples/fabric/marbles/config.yaml --caliper-networkconfig networks/fabric/v1/v1.4.4/2org1peercouchdb_raft/fabric-go-tls-solo.yaml

In the config.yaml it is specified the workload that is being used and the fabric-go-tls-solo.yaml, as we have mentioned before, is the configuration of hyperledger Worker.

In Fig. 2, the config.yml file for the 10/90 workload is displayed. The execution involves two distinct script types: init.js and query.js. The former, init.js, is designed to generate and submit a single transaction to the "marbles" smart contract, thereby facilitating write operations on the blockchain. The latter, query.js, is responsible for simulating a read-only transaction on the "marbles" smart contract.

The txNumber parameter dictates the frequency of script execution. In this particular scenario, where the goal is to execute 10% reads, the value is set to 100, ensuring the execution of 100 read operations using query.js.

```yaml
test:
  workers:
    type: local
    number: 5
  rounds:
    - label: init
      txNumber: 900
      rateControl:
        type: fixed-rate
        opts:
          tps: 900
      workload:
        module: benchmarks/samples/fabric/marbles/init.js
    - label: query
      txNumber: 100
      rateControl:
        type: fixed-rate
        opts:
          tps: 200
      workload:
        module: benchmarks/samples/fabric/marbles/query.js
```

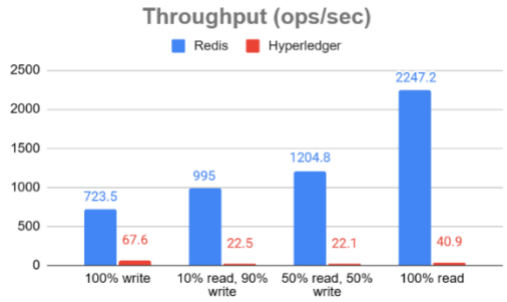
Fig. 2: Write-oriented (10/90) workload file for Hyperledger Caliper.

The tps parameter influences the rate at which transactions are dispatched per second. To comprehensively assess database behavior, we attempted to submit all requests simultaneously. Although Caliper did not attain the specified tps rate, it processed requests at its maximum capacity. It is noteworthy that we encountered challenges with the 50/50 workload. Setting a high tps value for read transactions resulted in database errors. Consequently, we conducted multiple iterations to determine the optimal tps value that the database could accommodate. This error was only reported in the 50/50 read operations. The remaining config files can be observed in our GitHub [1]

E. Results

1) Throughput and latency

Fig. 3 illustrates the comparative average throughput and latency metrics between Redis and Hyperledger Fabric. As can be seen,



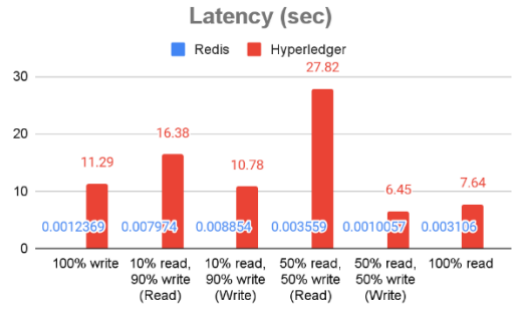


Fig. 3: Average throughput and latency for both database types. Top, throughput results. Bottom, latency results. The blue bars represent results for Redis; the red bars represent results for Hyperledger Fabric.

Redis exhibits superior performance by higher throughput and lower latency in contrast to Hyperledger Fabric. Notably, Redis demonstrates a remarkable throughput advantage, approximately 50 times higher than Hyperledger Fabric in read-intensive and mixed read/write workloads, whereas this margin is approximately 10 times in the write-intensive workload.

The latency analysis shows that Redis has millisecond-level responsiveness, which significantly outpaces Hyperledger Fabric. In addition, it is noteworthy that Hyperledger Fabric exhibits comparatively low write latency in the mixed read/write scenario.

2) Resource utilization

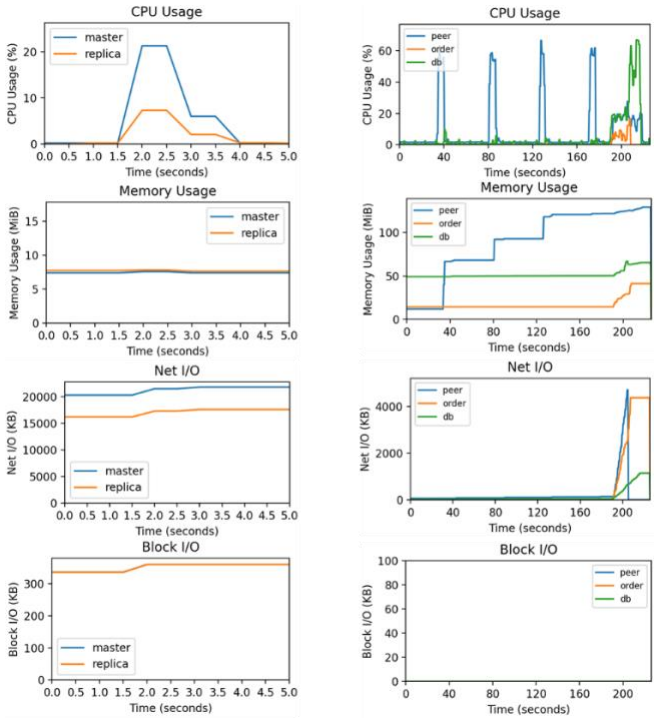
In order to understand the resource utilization throughout the execution phase, the command "docker stats" is employed to assess performance. This command fetches metrics data, including CPU usage, memory usage, network I/O, and block I/O, at intervals of half a second.

CONTAINER ID	NAME	CPU %	MEM USAGE / LIMIT	MEM %	NET I/O	BLOCK I/O	PIDS
c3685e10f9aa	redis-cluster_redis-replica_2	4.53%	6.75MiB / 3.835GiB	0.17%	10.3MB / 12.7MB	2.31MB / 19.8MB	7
4fd6ce3a7999	redis-cluster_redis-replica_3	3.90%	6.473MiB / 3.835GiB	0.16%	10.3MB / 12.7MB	1.47MB / 19.8MB	7
b2c96ac9b1c5	redis-cluster_redis-replica_1	4.29%	7.141MiB / 3.835GiB	0.18%	10.3MB / 12.7MB	3.92MB / 19.8MB	7
1d5fdb1b406b	redis-cluster_redis-master_1	12.16%	9.977MiB / 3.835GiB	0.25%	41.9MB / 33.4MB	14.9MB / 19.3MB	6

CONTAINER ID	NAME	CPU %	MEM USAGE / LIMIT	MEM %	NET I/O	BLOCK I/O	PIDS
480aad305f43	dev-peer0.org1.example.com-marbles-v0	0.00%	6.047MiB / 2GiB	0.30%	4.37kB / 2.67kB	401kB / 0B	6
959b91ef4c4b	dev-peer0.org2.example.com-marbles-v0	0.00%	5.848MiB / 2GiB	0.29%	4.25kB / 2.55kB	0B / 0B	6
b8e62a6e76b6	peer0.org2.example.com	55.57%	133.8MiB / 3.835GiB	3.41%	92.6kB / 47.6kB	0B / 303kB	10
4cf5b2f4143c	peer0.org1.example.com	25.19%	92.5MiB / 3.835GiB	2.36%	95.7kB / 49.3kB	0B / 303kB	10
90b0bd213b9d	orderer0.example.com	0.08%	14.77MiB / 3.835GiB	0.38%	22.7kB / 37.7kB	197kB / 279kB	10
5a8adf6ca64d	couchdb.peer0.org2.example.com	0.32%	48.15MiB / 3.835GiB	1.23%	30.4kB / 30.8kB	0B / 1.88MB	41
3793c3644c8b	ca.org2.example.com	0.00%	7.664MiB / 3.835GiB	0.20%	3.82kB / 6.01kB	0B / 811kB	8
8b3d838aced0	couchdb.peer0.org1.example.com	0.59%	48.59MiB / 3.835GiB	1.24%	31.3kB / 32.5kB	0B / 1.88MB	41
edc501edce5f	ca.org1.example.com	0.00%	7.992MiB / 3.835GiB	0.20%	3.69kB / 6kB	0B / 811kB	7

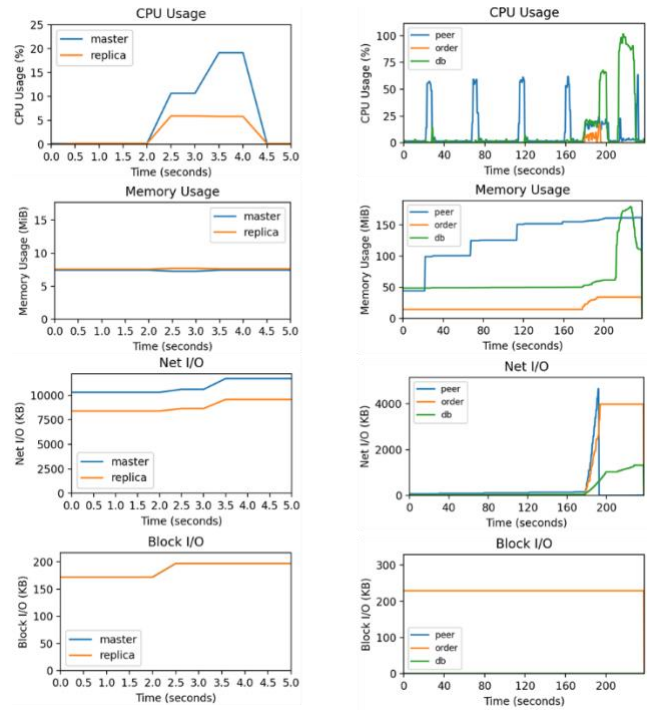
Fig. 4: Container resource usage. Top, Redis resource utilization. Bottom, Hyperledger Fabric resource utilization

Furthermore, we leverage *matplotlib.pyplot*, a prominent Python library, for the graphical representation of metric data which facilitates a more intuitive and visually insightful analysis of performance trends.



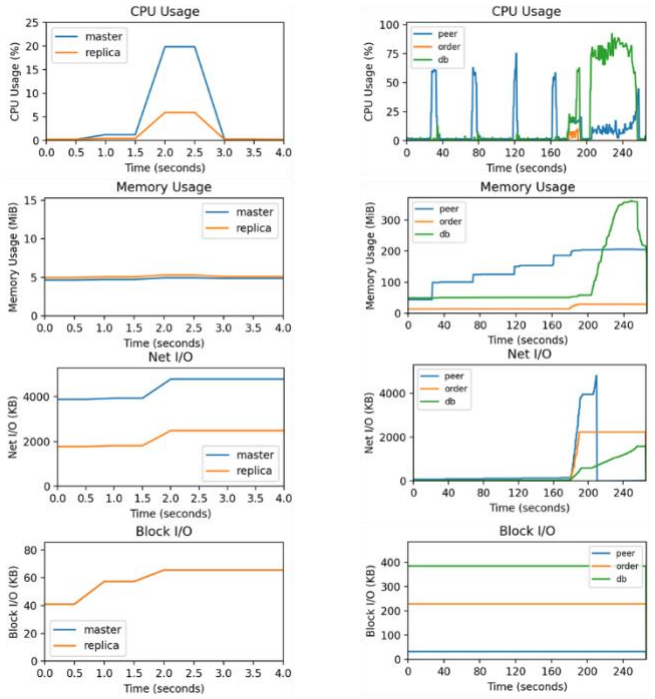
(a) Redis (b) Hyperledger

Fig. 5: Evolution of the resource usages for Redis and Hyperledger under 100% writing workload



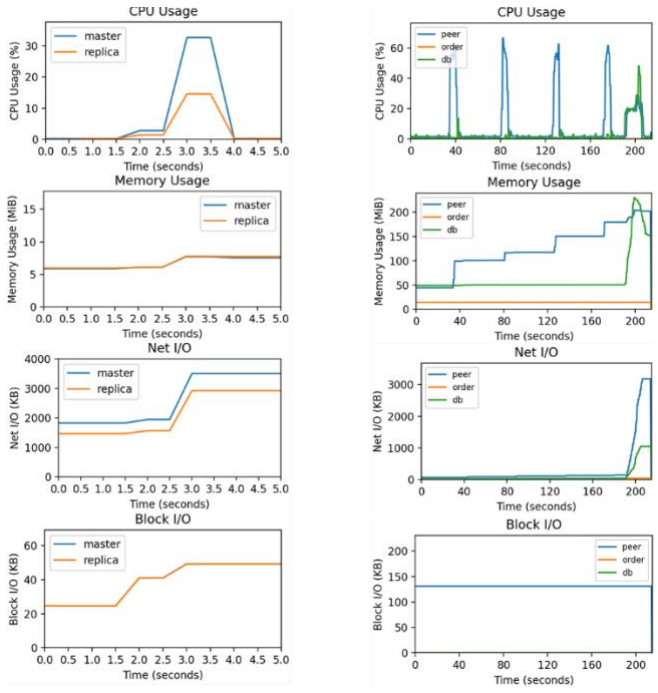
(a) Redis (b) Hyperledger

Fig. 6: Evolution of the resource usages for Redis and Hyperledger under 10% read / 90% write



(a) Redis (b) Hyperledger

Fig. 7: Evolution of the resource usages for Redis and Hyperledger under 50% read / 50% write



(a) Redis (b) Hyperledger

Fig. 8: Evolution of the resource usages for Redis and Hyperledger under 100% reading workload

The recording of master and replica nodes in Redis, as well as peer, order, and CouchDB nodes in Hyperledger, yielded noteworthy observations.

- Redis has a capacity to execute 1,000 operations within five seconds, whereas Hyperledger requires over 200 seconds.
- In Redis, the higher writing operations generate higher Network I/O and Block I/O.
- In Hyperledger, the CPU usage of the order node registers near zero in read-intensive workloads. Conversely, in write-intensive workloads within Hyperledger, there is an absence of block I/O traffic across all nodes.

These findings provide interesting insights into the distinctive performance characteristics and resource utilization patterns of Redis and Hyperledger nodes under varying operational workloads.

IV. Discussion

The following section compares results for each category of results and explain the observed differences.

A. Throughput and latency

The throughput and latency data shows many interesting differences between Redis and Hyperledger Fabric.

The first and most obvious is the disparity between the number of operations per second for each database type: Redis has a throughput of between 10x and 50x that of Hyperledger Fabric. This can be explained by the nature of each type of database. While Redis in an in-memory key-value store, Hyperledger is based on a blockchain. This differing nature is a theme that will come back multiple times throughout this section.

In this case, the high throughput of Redis is explained by its nature: reading data is extremely fast in a key-value database. Since every read operation is equivalent to a 'get' from a hash map, each is an operation of constant complexity (O(1)). The fact that the whole database stays in memory only accelerates this. As for writes, they can be appended at the end of a growing table without issue since links between entries are not encoded. They are slower on average, however, because of the need to periodically rehash the tables when it fills up [3]. This aligns with the slowdown observed in the data, proportional with the number of writes. It also explains the low latency in all workloads: there is very little overhead before any transaction is complete.

Hyperledger Fabric, on the other hand, is based on blockchain. Each transaction goes through an execution phase, an ordering phase, and a validation phase, each of which take time to complete [4]. These phases will be discussed in more depth later in this paper. The transaction is not complete until it is validated in a batch at the end of the validation phase. The complexity of the transactions explains both Fabric's low throughput and its high latency. The data indicates that it can take up to almost 30 seconds until a transaction goes through the validation phase.

All in all, Redis proved to have both a much higher throughput and much lower latency than Hyperledger Fabric.

B. Resource utilization

The resource utilization data consists of graphs representing the evolution of CPU, memory, network, and disk usage throughout each consistent with workload, for both database types. The patterns in these graphs are consistent with the data regarding throughput and latency.

The first phenomenon to point out is the length of the x-axis. Redis completes the 1000-operation workloads in five seconds or less, while Hyperledger takes near five *minutes* to process the same number of transactions. This speaks to Redis' efficiency and is consistent with its much higher throughput.

Moreover, both types of database display resource utilization patterns that are dramatically different from each other. However, both Redis and Hyperledger shows a pattern that is very similar across workloads.

1) Redis

In its short workloads, Redis displays allow CPU utilization that grows for one or two seconds in the middle if the task, before returning to a near-zero level. This is true for both the master and replica nodes, though the former always peaks at a higher utilization percentage. This is consistent with the high throughput we observed for Redis: the task is completed during the CPU usage "bump", while the beginning and end are buffer phases.

At the same time as CPU usage peaks, network usage also grows slightly, and then stays at its elevated level for the rest of the task. The same is true for disk usage, but only for the replica nodes. This implies that the nodes communicate with each other more during the operations, and the replica node is where the data is stored. This is surprising, since the data should have been stored in memory, negating the need for disk block reads.

As for memory usage, it stays remarkably low and consistent throughout, with the largest change being a raise of one or two MiB used, and only in the read-only workload. This is unexpected, since Redis stores data in memory: an increase in usage would have been expected considering 1000 entries are added in the write-intensive workloads. This might be explained by small-sized entries in a benchmark test like the one created by YCSB.

2) Hyperledger Fabric

The CPU usage of the different nodes of the Hyperledger Fabric network display a fascinating pattern, which is explained very well by the three phases of the transactions mentioned earlier [4].

In the execution phase, the peer node creates a transaction proposal and simulates the following reads and writes as if it were valid. This goes on until enough peers cryptographically endorse this transaction. This endorsement process is CPU-intensive and explains the periodic peaks in CPU usage observed on the blue peer node graphs. The accompanying memory-usage step increases are also consistent with the newly enlarged set of transactions the peer needs to keep simulating until its transactions are validated.

After the execution phase comes the ordering phase. This step is where the peer-endorsed transactions are sent to the ordering node (in orange on the graphs), which places them in order and mints them into a block. This is consistent with the increase in CPU, network, and memory usage of the orderer node at the end of the task. The transactions have all reached the endorsement threshold and are now hashed in a block. As expected, this doesn't happen in the read-only workload.

Then finally comes the validation phase, where a peer node validates the transactions and updates the ledger. This is where the DB node (in green) is activated. Note that the ledger includes read operations as well as transactions, which explains the activation in the read-only workload.

In the end, Redis used fewer resources, for a shorter time, and in a more consistent manner than Hyperledger Fabric.

C. *Limitations*

The main measures of this paper were the throughput and latency of two types of distributed databases. This is a major limitation because these are the specific characteristics for which Redis has been optimized. Hyperledger Fabric, on the other hand, is built for security, reliability, and inter-organization trust.

The measures chosen are akin to comparing how easily one can pry open a wooden crate using a crowbar and a screwdriver. Though both can get the task done, the crowbar was designed specifically for that end, while the screwdriver is also useful in many other ways once the crate is open. The same goes with our databases.

Though Redis beat Hyperledger in the metrics we measured, it lacks the consistency and trustworthiness of a blockchain-based solution. The advantages of Hyperledger are clear, but our test suite did not include ways to measure them.

Moreover, the Hyperledger documentation mentions some performance limitations that happen when using CouchDB in one's implementation [5]. Our early decision to use this default might have exacerbated the performance disparity between the database types.

V. CONCLUSION

This work compared the throughput and latency of two types of non-relational distributed databases, Redis and Hyperledger Fabric. These metrics were recorded in four 1000-operation workloads comprising varying proportions of reads and writes. The resource utilization of both database types during each workload were also measured and analyzed.

For each measure included, Redis was superior to Hyperledger Fabric: it had significantly more throughput, lower latency and completed the workloads much faster. Its resource utilization was also both lower and more consistent, leading to potentially easier hardware management.

If these are a priority for one's distributed database application, then Redis is undoubtedly a better choice. However, many advantages of a blockchain-based solution such as Hyperledger Fabric were poorly represented in our test suite, which might lead to a skewing of the conclusions drawn from the data.

Further studies testing the performance of both database types on matters of security, consistency and privacy would help get a more complete picture of the pros and cons of each. Moreover, adding other types of distributed databases to the table, like Cassandra, MongoDB, or even distributed SQL databases like Google Spanner, would help single out the best tool for many different use-cases.

REFERENCES

[1] Season06 (2023) season06/TP3/tree/main. GitHub. [Online]. Available: https://github.com/season06/TP3/tree/main

[2] Hyperledger Caliper hyperledger/caliper-benchmark. GitHub . *[Online]. Available:* https://github.com/hyperledger/caliper-benchmarks

[3] Nath, Kousik (2017) Medium. *A little internal on Redis hash table implementation* [Online]. Available : https://kousiknath.medium.com/a-little-internal-on-redis-key-value-storage-implementation-fdf96bac7453

[4] Guggenberger, Tobias; Sedlmeir, Johannes; Fridgen, Gilbert; Luckow, André (2022) Computers & Industrial Engineering. *An in-depth investigation of the performance characteristics of Hyperledger Fabric* [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0360835222007045#sec2

[5] Hyperledger project (2023) *Performance considerations* [Online] Available: https://hyperledger-fabric.readthedocs.io/en/latest/performance.html

[6] Redis Labs. (n.d.). Redis. [Online]. Available: https://redis.io/

[7] The Linux Foundation. (n.d.). Hyperledger Fabric. [Online]. Available: https://www.hyperledger.org/use/fabric

[8] Smith, J. (2020). "Scalability and Performance in Distributed Databases." Journal of Distributed Systems.