Comparison Of Different Shared-Log Systems

Subhajit Gorai
sgorai2@illinois.edu
Univeristy of Illinois,
Urbana-Champaign

Chang Xu
changxu6@illinois.edu
Univeristy of Illinois,
Urbana-Champaign

Zixiang Xu
zx40@illinois.edu
Univeristy of Illinois,
Urbana-Champaign

ABSTRACT

Replicated shared-log systems have been used for many purposes from messaging queues to storing metadata and data structures distributedly across multiple machines. Different systems use different mechanisms for consensus and reliable storage, some use primary-backup replication based on consensus algorithms like Raft, and others use Chain replication. Additionally, there are various data-sharding mechanisms for scalability such as Static-Partitioning, Dynamic-Partitioning, etc. Given this myriad set of choices to build an entire system, different systems try to optimize different aspects of the problem leading to a wide variety of choices for system users who want to use this system for their use cases. There is no prior work that compares these log-based systems holistically. In this paper, we will compare two partial order systems- Kafka and BookKeeper, and two total order systems - Corfu and Scalog. The goal of this paper is to provide a benchmark that compares these systems on a variety of read-write workloads and presents clear and thorough latency and throughput values for each system.

ACM Reference Format:
Subhajit Gorai, Chang Xu, and Zixiang Xu. 2022. Comparison Of Different Shared-Log Systems. In . ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Replicated shared-log systems are often very complex internally, and understanding it completely with all the nuances is time-consuming. For a system user who wants to build systems on top of an existing shared-log system that solves a particular part of the problem, they have a wide variety of choices at their disposal. These shared-log systems can be used interchangeably to perform the task at hand. It is also a very common problem where one shared-log system can be used in a variety of different ways. Consider for instance Kafka, some might use them as a temporary messaging queue to transfer data from one process to another and delete them when done, while others can use them just for storage, to reliably store log-structured data.[2] These different usages all have different performance requirements. Given such a wide variety of choices and lack of uniform comparison between different systems, an end-user of the system has to go through a tedious and time-consuming process of understanding it thoroughly only to find out later in the future that there are even better systems that are available solving similar problems.

In order to solve this problem we aim to provide a single paper that can be referenced for all the metrics that one can potentially require before selecting any particular system. We compare two partially-ordered systems - Kafka[7] and BookKeeper[1]. These systems logically present a coherent view of different shards(partitions in Kafka and Ledger in BookKeeper) to one single unit, where inside each shard the data is totally-ordered but across different shards, it only provides partial-order guarantees. Next, we compare two totally ordered systems - Corfu[3] and Scalog[6], where there is a global order across each individual entry of the entire shared log. Since total-order systems provide stricter semantics and hence would be obviously slower, we are not comparing them with partial-order systems. However, the single-shard model for Kafka and BookKeeper can still be compared with total-ordered systems since both of them(Kafka and BookKeeper) will now provide total-ordering guarantees (with limited scalability).

In our study, we have created a uniform dataset with varying sizes that can be used consistently for each of the 4 systems that we are comparing. Additionally, we have kept the topology to be the same across all systems. In situations where we required an external consensus system for reliable storage of metadata like ZooKeeper[1, 7], we have used 3 servers. So both Kafka and BookKeeper use a 3-node ZooKeeper ensemble, where all three nodes are connected by a switch. In addition to it, each system is built on a 3-node cluster of servers with a data replication factor of 3. Therefore, Kafka and BookKeeper will have 3 Brokers and 3 Bookies respectively. For Corfu and Scalog we are using 3 servers connected to one another through a switch. Since these systems have an in-built consensus protocol[3, 6], they do not require any external systems like ZooKeeper. All these servers are homogenous in terms of their configuration. Each individual server has 10 core CPU, 192 GB RAM, and a link data rate of 1Gbps connected to one another with a switch of 10Gbps. The reader and writer clients that issue read and write queries respectively to these systems are run on a separate machine.

The rest of the paper will be organized as follows. We will first revisit the background and the motivation for our study in Section 2. Then we will present the high-level design of our study in Section 3. In Section 4, we will discuss some low-level implementation details and some nuances when conducting our measurements. We will then present our results in Section 5 and explain our findings. We will then present some related studies in Section 6. Finally, we will summarize our findings and discuss some future works in Section 7.

2 BACKGROUND AND MOTIVATION

As mentioned in the previous section, replicated shared-log systems are systems that provide an ordered logical log for clients. It is one of the building blocks for building distributed applications ranging from Log Aggregations for application-level offline analysis[2, 7, 8]

, ,

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

to infrastructural-level crash and transaction recovery[1, 2]. Different use cases may have different performance and scalability requirements. For example, messaging systems care more about the end-to-end latency for producing and consuming an entry. Usages for crash recovery such as building a WAL, on the other hand, care more about throughput. On top of this, there exists a wide variety of replicated shared-log systems which all differ in their design and make different trade-offs. Although each system provides its own set of benchmarks depicting latency and throughput numbers for workloads, they are not generic enough to be used in real-world application building. Additionally, they might not present a holistic comparison of different competing systems that already exist. One such reason for this might be that the system developed might have some characteristics that deviate significantly from the existing systems in their viewpoint, such that comparing them does not seem right to them. Therefore, there is a need to conduct a holistic comparison between replicated shared-log systems on their different guarantees, read and write APIs, and performance and scalability under different workloads.

3 DESIGN

The goal of our measurement is to evaluate the performance of the interchangeable shared-log systems with the same level of guarantees. We first split four of the systems into two groups 1. ones that only provide total ordering on a single shard: Kafka and BookKeeper 2. ones that provide total order across different servers in all shards: Corfu and Scalog. Since different systems have different terminologies, we will keep the same set of terminology for the rest of the paper. We will name the basic unit of ordered storage as shards (ledgers in BookKeeper and partitions in Kafka).

3.1 Single Shard

For partial order systems, we performed a sequential write workload on a single shard with three different entry payload sizes. We calculated each of their throughput and latency under all those settings. We then perform a sequential read on each of the payload sizes. For total order systems, we are only testing a single shard (stream in Corfu) with a single producer/consumer producing and consuming on the same shard with a similar setting as the partial order system experiments. In addition, we add an experiment that uses parallel producers producing at the same shard. Similarly, we perform a parallel uniform read that randomly reads all the records on the same shard.

3.2 Multi Shard

For partial order systems, we open 4 shards at the same time. Then we concurrently open 4 producers that each write to a shard sequentially. After that, we concurrently open 4 consumers to read from each of the shards sequentially. We then repeat this process and use 8 shards to test the system's overall throughput and scalability.

4 IMPLEMENTATION

For the study, our benchmark consists of a series of tests, where each test corresponds to a dataset and a set of interactions with the data.

4.1 Data Set

We generate the dataset with randomly generated strings with uppercase letters, lowercase letters, and digitals. By specifying the length of the string, we can get a record with specific size. For our benchmark, we have three record sizes: 100 bytes, 50 KB, and 1 MB.

4.2 API Definition

Both partial-ordered systems and total-ordered systems provide several APIs for basic operations like read and write.

4.2.1 Kafka. Kafka provides 2 variations of *send(record)* to append records in a Kafka topic, one with and without a callback function. It also provides *poll(timeout)* to consume records for a topic. By default, it will start consuming from the latest record. However, one can use *seek(partition_no, offset)* to set the partition and starting offset of the topic. Consumers also have flexibility of committing offset, so that Kafka knows which records have been successfully consumed.

4.2.2 BookKeeper. BookKeeper provides both API for adding an entry to a Ledger *addEntry (entryId, data)* which will first calculate all the servers within the ensemble and then issue a write to each of the server. BookKeeper also provide a read API textitreadEntries(firstEntry, lastEntry), which will read all the data from the range specified.

4.2.3 Corfu. CorfuDb only exposes high level APIs for different datastructures that are built over low level append and read opetation onto the shared log. To be consistent, we are using CorfuQueue, a persistent queue type abstraction built over CorfuTable in our benchmark. It provides two APIs: *enqueue(message)* and *get(corfu_record_id.* After the enqueue operation the client receives a corfu_record_id which can be later used to retrieve the same record.

4.2.4 Scalog. Scalog mainly provides two APIs, including *Append (record)* and *Read (global_seq_number, shard_id).* By calling the Append API in the library, the client will directly write the record into a data storage server and get a response with corresponding global sequence number and shard id. As global sequence number and shard id uniquely identify a record, we can call the Read API to get the specified record.

4.3 Tests for Partial-ordered Systems

For partial-ordered systems, we compared Kafka and BookKeeper with both synchronous clients and parallel clients. We performed sequential write-only and read-only workload under three different record size (100 Bytes, 50KB, 1MB) and measured their latency and throughput. We also performed a parallel write/read workload on 4/8 different shards using 4/8 different producers/consumers. Consumers for these systems only reads records contiguously starting from a particular offset. Therefore, we can not do multiple random reads at various locations of the log.

4.4 Tests for Total-ordered Systems

For total-ordered systems, we also used both synchronous tests and parallel tests to compare Corfu and Scalog. We performed a write-only workload for a single producer with three different

record sizes (100 Bytes, 50KB, 1MB). The only difference in testing with respect to Partial Ordered Systems is that all the concurrent reads and writes happen at the same shard to test the scalability limits. Additionally, in this scenario, Consumers performs reads at random locations of the log using the offset retrieved while doing append operations. Since, Scalog does not provide any batch get operations to retrieve bunch of records together, we are not testing CorfuQueue's batch get API *corfuQueue.entryList()* here for fair comparison.

5 RESULTS

5.1 Kafka vs BookKeeper

5.1.1 ***Producer Comparison.*** With a Single-Shard setup single producers are producing records one by one for each of the 3 different sizes of messages - 100B, 50KB, and 1MB. The producers are run one at a time, i.e, when only one producer is running at a given time. Table 1 shows the throughput and latency numbers for one-by-one synchronous writes. We see that the throughput for Kafka is 1.7x as compared to BookKeeper for 100B message size. Additionally, when the message size is increased to 50KB, the throughput rises to approximately 5x of BookKeeper and saturates to this value for 1MB size data. Moreover, the mean and tail latency of each individual write for Kafka, for each of the three message sizes, is much smaller than BookKeeper. Furthermore, similar to throughput numbers, the difference between latencies becomes more prominent as we increase the message sizes.

Table 2 shows performance numbers for Multi-Shard setup, where multiple concurrent producers are writing 50KB messages to their respective shards (different topics for Kafka and ledgers for BookKeeper). The "Combined Latency" denotes the merged latencies for each of the individual producers, and "Average Throughput" denotes the mean of the individual Producer's throughput. It can be seen that both throughput and latency for synchronous writes are better for Kafka.

The reason for this behavior can be analyzed by dissecting the write path of Kafka. The Producers first contact any one of the active brokers to fetch the address of the broker-leader specific to that partition. All active brokers will have information of partition to its broker leader mapping through Zookeeper. They might occasionally cache this information for efficiency. Once the Producer knows about the broker leader, it then sends data directly to it without any intervening routing tier.[7] Additionally, records are often batched together and written contiguously at the brokers' end.

In BookKeeper, the write path starts with the Producer contacting the Zookeeper for an active ensemble of bookies. Then for each individual entry, it will pick a subset of an ensemble, depending upon the write quorum, to send the write request and will wait for the majority before committing. Here we see that client itself tries to order a request, and thus latest entry can't be committed until previous entries are committed. This is similar to broker-leader in Kafka with the broker being able to better perform it because of batching. Also, for each bookie, every ledger entry is written to both a journal; a record describing the update, and long-term storage known as Ledger Storage (entry log). The response is sent back to the client only when an entry has been written to the journal and a fsync performed, and also written to Ledger Storage. This entire process should happen for the majority of bookies in order to commit the record by the client. Finally, every BookKeeper entry has a lot of metadata (Ledger number, Entry number, Last confirmed, Data, Authentication code) in it as compared to messages in Kafka, thereby increasing the overall size of the records as well as the CPU time (for coping and creating a new record with all the relevant fields instead of directly writing the message). [1]

5.1.2 ***Consumer Comparison.*** Consumers for Kafka and BookKeeper are similar. The Kafka consumer works by issuing "fetch" requests to the brokers leading the partitions it wants to consume. The address of the leading broker can be found by connecting any one of the brokers from the ensemble. The consumer specifies its offset in the log with each request and receives back a chunk of log beginning from that position. The BookKeeper Consumer works in a similar fashion, with the client contacting the Zookeeper to get the relevant "Ledger Storage" with a ledgerId and entryId as a parameter. The client then starts consuming from the Ledger Storage. Both Kafka and BookKeeper read in batches to improve throughput. [1, 7]

However, there are some optimizations in Kafka. First, in Kafka, each data file stores the contents of one partition only. Whereas, in BookKeeper entry log files might contain entries of different ledgers as well, thus blocking a contiguous read whenever such an entry is encountered. Second, both the producer and the consumer use the same message format, which allows for an important optimization: network transfer of persistent log chunks. Note that this is again because the entire file belongs to one partition and is thus meant for consumers of the same topic.

As we can see from Table 3, Kafka consumers perform better for 100MB, and 1MB message sizes. BookKeeper is slightly better for 50KB message sizes. Table 4 shows the result for running 4 and 8 concurrent consumers. These consumers are reading 50KB messages from their respective topics parallely. Like the single consumer case, we see that BookKeeper performs better here. It is due to manual tuning of batch size for BookKeeper Consumer since by default, it reads records one by one. We believe that if we fine-tune Kafka rather than using the default "off the shelf" values, Kafka's read throughput would be better than BookKeeper.

5.2 Corfu vs Scalog

5.2.1 ***Producer Comparison.*** We compare Corfu and Scalog in a similar single-shard setup like Kafka and BookKeeper. We run Producers one at a time for three different message sizes 100B, 50KB, and 1MB. Results are shown in Table 5. Corfu's latency for 100B and 50KB message sizes is better by a slight margin than that of Scalog. However, for large data sizes of 1MB Scalog perform better(nearly 2x speedup in latency and throughput). Before analyzing this behavior, we will look into details of append path for both Corfu and Scalog. Corfu appends data by getting the log tail from Centralized Sequencer, and then directly issues a write request to the corresponding storage servers through a locally cached mapping function.[3] Scalog performs appends by first selecting a shard according to the current sharding policy. If no policy is given it picks a shard randomly from the live shards set. Similar to Corfu, it then issues a write request directly to the storage server holding that shard. However, since the records need to be globally ordered

Metrics	100B		50KB		1MB	
	Kafka	BookKeeper	Kafka	BookKeeper	Kafka	BookKeeper
Mean Latency(ms)	2.00	4.00	3.00	20.00	36.00	139.00
Tail Latency(99.9% ms)	20.01	11.00	28.00	134.01	213.06	616.00
Throughput(MB/s)	0.038	0.023	13.47	2.37	27.21	7.15
Throughput(records/s)	400.00	250.00	280.00	50.00	27.00	7.00

Table 1: Comparison of Kafka and BookKeeper for Single-Shard Producer for three different message sizes

Metrics for 50KB	4 Concurrent Producers		8 Concurrent Producers	
	Kafka	BookKeeper	Kafka	BookKeeper
Combined Mean Latency(ms)	4.57	21.50	6.5	21.96
Combined Tail Latency(99.9% ms)	31.01	539.51	214.00	644.10
Average Throughput(MB/s)	10.63	2.27	7.5	2.22
Average Throughput(records/s)	217.77	46.5	153.7	45.5

Table 2: Comparison of Kafka and BookKeeper for Multiple Concurrent Producers

Metrics	100B		50KB		1MB	
	Kafka	BookKeeper	Kafka	BookKeeper	Kafka	BookKeeper
Throughput(MB/s)	0.78	0.48	64.00	78.00	78.00	38.00
Throughput(records/s)	8000.00	5000.00	1350.00	1550.00	78.00	38.00

Table 3: Comparison of Kafka and BookKeeper for Single-Shard Consumer for three different message sizes

Metrics for 50KB	4 Concurrent Consumers		8 Concurrent Consumers	
	Kafka	BookKeeper	Kafka	BookKeeper
Average Throughput(MB/s)	25.82	51.58	13.83	27.68
Average Throughput(records/s)	541.48	1056.53	290.18	567.06

Table 4: Comparison of Kafka and BookKeeper for Multiple Concurrent Consumers

Metrics	100B		50KB		1MB	
	Corfu	Scalog	Corfu	Scalog	Corfu	Scalog
Mean Latency(ms)	9.00	10.02	15.00	19.83	88.00	44.63
Tail Latency(99.9% ms)	29.00	19.00	65.01	24.00	257.01	78.00
Throughput(MB/s)	0.010	0.009	2.30	2.45	7.29	14.92
Throughput(records/s)	110.05	99.76	47.16	50.20	7.29	14.92

Table 5: Comparison of Corfu and Scalog for Single-Shard Producer for three different message sizes

Metrics for 50KB	4 Concurrent Producers		8 Concurrent Producers	
	Corfu	Scalog	Corfu	Scalog
Combined Mean Latency(ms)	16.21	26.05	17.23	26.24
Combined Tail Latency(99.9% ms)	55.01	144.00	88.0	234.0
Average Throughput(MB/s)	2.26	1.87	2.16	1.85
Average Throughput(records/s)	46.42	38.35	44.38	38.04

Table 6: Comparison of Corfu and Scalog for Multiple Concurrent Producers

Metrics	100B		50KB		1MB	
	Corfu	Scalog	Corfu	Scalog	Corfu	Scalog
Mean Latency(ms)	0.44	0.34	0.48	1.20	0.51	13.03
Tail Latency(99.9% ms)	2.00	1.30	2.00	2.43	2.00	20.96

Table 7: Comparison of Corfu and Scalog for Single-Shard Uniform Consumer for 3 different message sizes

Metrics for 50KB	4 Concurrent Consumers		8 Concurrent Consumers	
	Corfu	Scalog	Corfu	Scalog
Combined Mean Latency(ms)	0.38	0.73	0.40	0.75
Combined Tail Latency(99.9% ms)	2.00	1.64	2.10	1.91

Table 8: Comparison of Corfu and Scalog where Multiple Consumers are reading randomly from the same shard

as well, it waits for an acknowledgment from the Ordering layer, through periodic global cuts, before sending a successful response back to the client. [6]

The reason Corfu performs better for small and medium size messages is that it does not have to wait for any ordering layer to commit its message. The Global Sequencer already orders them before writing. In either case, 2RTT is required for successful append but in the case of Corfu there is no delay, once a tail is fetched, a subsequent call to storage servers to append the data is made instantaneously. Whereas, in Scalog after appending to a shard, a call to the ordering layer is made periodically. However, for larger-size records, Scalog can pipeline the work of appending data with a call to the ordering layer. Thus, for 1MB records, Scalog performs better.

The results for 4 and 8 concurrent producers producing at the same stream for 50KB records are shown in Table 6. Corfu's mean and tail latencies are better than that of Scalog because the global sequencer can process multiple requests fast. Therefore, Corfu is better scalable in terms of parallel appends.

5.2.2 ***Consumer Comparison.*** Corfu and Scalog perform read operations similarly. For reading, Scalog client library contacts any one of the storage servers holding the data for that shard. It then issues a read request with a global sequence number that it got originally while appending the entry. Corfu works in a similar way by issuing a read request with offset as a parameter to the storage server by using the mapping function. From Table 7 we see that the uniform random read latencies for 100B data are nearly the same for both of them, but latencies degrade for Scalog as we increase the message size. We haven't found anything conclusive to justify this behavior of Scalog.

Table 8 shows the latencies for 4 and 8 concurrent consumers reading uniformly randomly from the same stream. One important observation is that these latencies are getting smaller, both for Corfu and Scalog, due to caching. Since multiple consumers are reading from the same stream, there is a higher possibility that some records are read multiple times, leading to a drop in latencies.

6 RELATED WORK

Understanding the performance of replicated shared log systems has been a topic of interest for so long. Performance evaluation helps choose suitable systems according to different scenarios like big data workloads or transactional workloads. One of the most notable benchmark frameworks for shared log systems is the YCSB (Yahoo! Cloud Serving Benchmark) framework which provides a set of benchmarks for facilitating performance comparisons of cloud systems.[5]

As we mentioned before, for most research about shared log systems, researchers customize their own benchmarks based on the systems they are testing because shared log systems do not have a

unified protocol and provide different operations and functionalities. Jay Kreps compared Kafka with another two popular message broker services ActiveMQ and RabbitMQ by designing benchmarks of cumulative producing and cumulative consuming.[7] As Kafka provides batching functionality, he also compared two batching modes of Kafka with other systems. For Tango[4], Mahesh showed latency and throughput of read, write, and transaction on scenarios with different amounts of views and partitions. Cong proposed Scalog and compared write latency and read throughput for Scalog and Corfu with different amounts of data servers in each shard.[6]

7 CONCLUSION

In this study, we evaluate the performance of 4 replicated shared log systems which include partial-ordered shared log systems and total-ordered shared log systems: Kafka, a distributed messaging system, which provides ordering within each partition; BookKeeper, a storage service optimized for real-time workloads offering strong consistency and ordering within each ledger; Corfu, a distributed shared log on top of flash clusters providing strong consistency and total order; Scalog, a shared log system providing high throughput and total order.

For partial-ordered systems, latency and throughput for both synchronous and parallel writes are better for Kafka than for BookKeeper. As Kafka directly appends batched records to its broker leader instead of itself waiting for the majority before committing like BookKeeper, Kafka spends less time committing a record. Reading processes for Kafka and BookKeeper is similar and we have a lot of parameters to finetune them to make them better.

For total-ordered systems, Corfu performs better than Scalog when producing smaller records. As Corfu achieves total ordering with Global Sequencer and Scalog with Ordering Layer, waiting for Ordering Layer to handle a batch of recent writes would be the bottleneck for Scalog. Additionally, Corfu is more scalable to handle multiple parallel requests.

Besides the performance measure during our study, we realized projects like Kafka and BookKeeper have an amazing community and clear documentation, whereas CorfuDB and Scalog are poorly maintained with unclear documentation and noticeable software bugs. This difference is often another reason why end-user would choose one system over another despite a performant design. Therefore, we suggest system designers should also consider system maintainability and usability besides performance optimizations.

REFERENCES

[1] Bookkeeper concepts and architecture. https://bookkeeper.apache.org/docs/getting-started/concepts.
[2] kafka use cases. https://kafka.apache.org/uses.
[3] Mahesh Balakrishnan, Dahlia Malkhi, John D. Davis, Vijayan Prabhakaran, Michael Wei, and Ted Wobber. Corfu: A distributed shared log. *ACM Trans. Comput. Syst.*, 31(4), dec 2013.
[4] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D. Davis, Sriram Rao, Tao Zou, and Aviad Zuck. Tango: Distributed data structures over a shared log. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 325–340, New York, NY, USA, 2013. Association for Computing Machinery.
[5] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.
[6] Cong Ding, David Chu, Evan Zhao, Xiang Li, Lorenzo Alvisi, and Robbert Van Renesse. Scalog: Seamless reconfiguration and total order in a scalable shared log. In *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation*, NSDI'20, page 325–338, USA, 2020. USENIX Association.
[7] Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, volume 11, pages 1–7, 2011.
[8] Guozhang Wang, Joel Koshy, Sriram Subramanian, Kartik Paramasivam, Mammad Zadeh, Neha Narkhede, Jun Rao, Jay Kreps, and Joe Stein. Building a replicated logging system with apache kafka. *Proc. VLDB Endow.*, 8(12):1654–1655, aug 2015.