Nil-Ext Interface for Kafka

Aditya Kaushik Kota	Akshat Sinha	Himanshu Sagar
kota@cs.wisc.edu	asinha32@cs.wisc.edu	hsagar2@cs.wisc.edu

Section 1: Introduction

Apache Kafka, is an open-source distributed event streaming platform used by thousands of companies for high-performance data pipelines, streaming analytics, data integration, and mission-critical applications. Apache Kafka was born out of Linkedin's need for collecting and delivering high volumes of log data with low latency. Today, it is used for messaging, website tracking, metrics, logs aggregation, stream processing, and event sourcing.

Kafka has topics - partitioned queues of messages. A partition is replicated across various brokers(servers) and managed in named topics. Each partition has a leader and in-sync replicas which try to stay in sync with all writes on the leader. Only the leader can read and serve data for a particular partition in Kafka. We include an **appendix** section to cover Kafka's system details.

Improving the throughput of Kafka has been explored both in academia and by official Kafka maintainers. While the previous works have predicted server parameters for optimal performance, Kafka maintainers have used different methods such as batching, disk cache, reducing copy using$sendfile, and end to end compression of messages\cite{kafka_compress} to improve the overall performance. They fall short on one key bottleneck of the system, i.e. extra effort to sync all followers *synchronously* when the ack=all setting is used for acknowledgment. We propose a new method to sync followers lazily, without compromising on the consistency guarantees provided by the current system.

Storage interfaces provide several opportunities for improvement. More often the improvement can be in the form of performance coming with the expense of higher complexity or lower consistency. In this project, we exploit the nil-externality property and improve the performance of replicated storage in Kafka with a strong consistency guarantee. A nil-externalizing storage interface may modify the state of the internal system without externalizing the effects to other systems or the outside world.

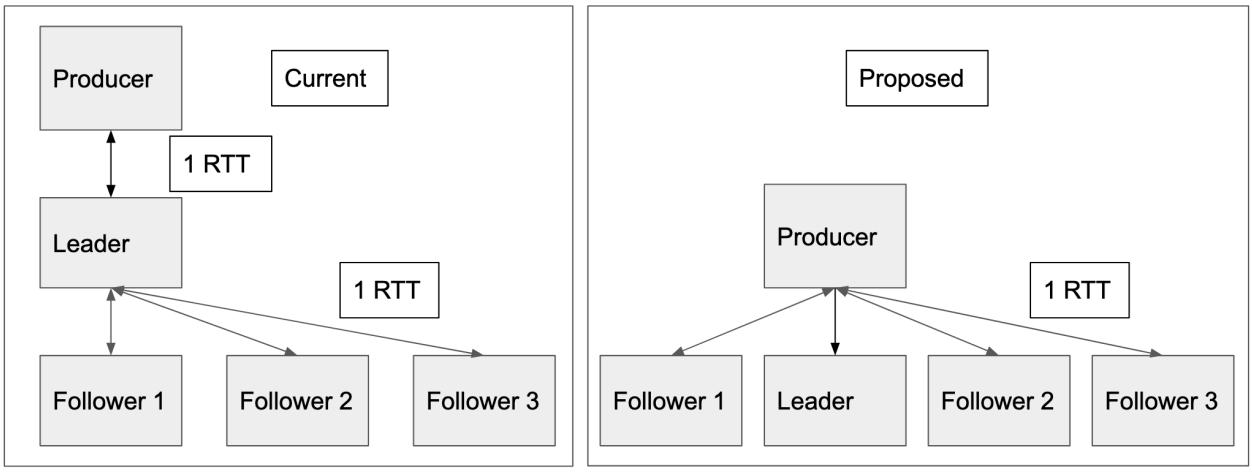


Figure 1: Current vs Proposed Architecture

In the new proposed architecture(fig 1), new messages are sent to all followers and the leader, but the order of messages is externalized (established) at a later time. In practice, data is read well after when it was written. Thus, in the meantime, we can modify the internal state of the storage system lazily\cite{nil}. This lazy update of the order reduces write round-trip-time (RTT) from 2 to 1, hence we expect a considerable increase in throughput.

Section 2: Design

We implemented a nil-ext interface over Kafka Produce API. In this interface, a produce request is sent to all replicas(leader and followers) and ack is received from at least a *supermajority* of nodes and successful write at the leader. *Supermajority* is defined as f + f/2 +1, where f is the number of failures we can tolerate. To implement nil-ext interface in Kafka, we solved the following problems:

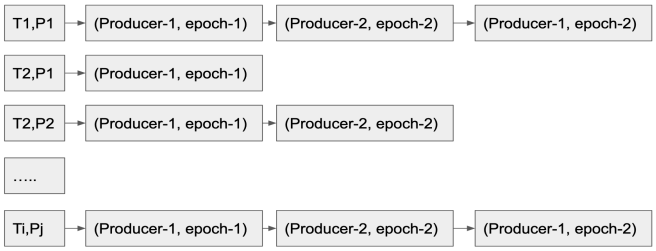
- Sending produce requests to leader and replicas
- The produce request will be processed differently at leader and followers:
    - On Leader:
        - Store global order of messages of the partition
        - Append entry to its log
        - Ack back producer
    - On Follower:
        - Store produce request in in-memory Data structure
        - Ack back producer and fetch order asynchronously
- Wait for ack from supermajority nodes(including the leader)
- Handle *Hole* issues on follower arising in case the message data is not present(due to network failure or crash)

In our system, we process the request at the leader, i.e. we append it to the respective partition before replying back to the producer but the request is kept in memory on the follower's side, which gets persists after fetching the right order is received from the leader asynchronously. This small deviation from the Nil-ext, Skyros(cite) helps us immensely in simplifying our read

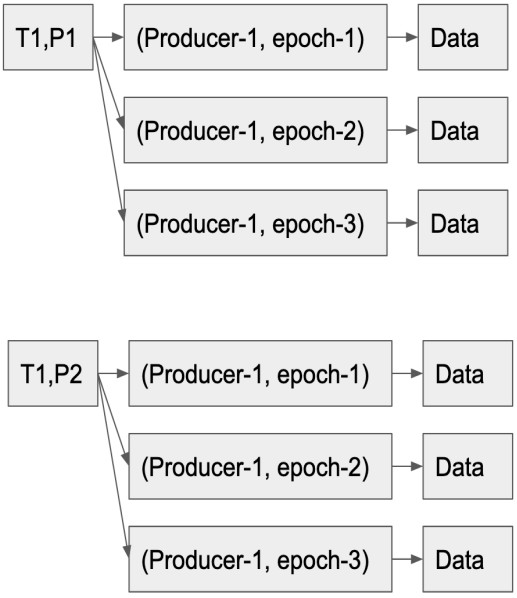
requests. All reads are processed by the leader, hence, we do not have a separate mechanism for recent reads and since Kafka is an append-only log structure, append on the leader is fast. Moreover, reads are not affected by slow nodes that can cause a slowdown in nil-ext interfaces.

Section 3: Implementation

The whole project is implemented in Java and Scala. We use a list of hashmaps(fig 2) on the leader to store the relative order of messages, which is synced using a separate fetcher thread running on all followers. The fetcher threads only bring global order of new messages and persist them accordingly. In case a follower does not has a particular message, it tries full fetch to get both message content and the global order. This fills a 'hole', which gets created in case a follower has not received the data of a produce request due to network failure or broker crash.



Part 2a: Structure of the HashMap stored at the leader side



Part 2b: Structure of the HashMap stored at the follower side.

Figure 2: Hashmaps stored on the leader and follower brokers. T1 is the topic name, while P1 is one of the partitions of that topic.

Overall, our implementation guarantees the following thing:

- Exactly once semantics: We use a tuple of producer ID, request sequence number, and leader epoch to uniquely identify a request. Producer ID is assigned when a producer connects to the leader, and the request sequence number is incremented at every successful produce request.
- Linearizability is guaranteed in a partition, as all followers will receive orders directly from the leader.
- Producer requests are marked successful after getting acks from a supermajority of nodes, including one from the leader.
- *Holes* of messages on followers are handled and failure of followers is handled using hole-filling logic

- Garbage collects old hashmap entries. This is done after a particular produce request is made durable on all followers.

Since the inception of this project, we have faced a lot of challenges, the most notable ones included how the network client is implemented for Kafka Producer. It has convoluted state management and does not allow sending more than one message at a time. We redesigned it to send produce requests to all replicas and mark successful when ack from supermajority nodes.

Testing: To test our system, we manually checked the md5 hash of log files at leader and follower to check if all the messages are replicated in the same order. This provides verification of data corruption and the global order of messages.

Section 4: Results

We use an open-source benchmarking tool (OpenBenchmark), to measure the performance of various workloads.

System Configuration: We run our experiment on 5 nodes, where each node runs on a c220g2 type server in CloudLab (Wisconsin). The initial Bandwidth of each connection is 1 GB/s. We scaled down the Bandwidth of each link to 100 MB/s using wondershaper.

We divide our results into 4 main sections to answer the following questions:

- What is the performance gain in latency of our nil-ext interface over the non-nil ext interface? (**4.1**)
- How does an increase in throughput in a limited network setting affect latency?(**4.2**)
- What is the scalability of a system with replication and partitions? (**4.3**)
- How do slow nodes affect our system? (**4.4**)

4.1 Nil-ext performance vs non-nil-ext interfaces

We compare the produce latency measured for nil-ext interface and standard Kafka ack modes for replication 3 setting. We observe a positive result, that for median latency nil-ext performance is similar to ack=1 mode, and even for tail latency, it is better than ack=all mode.
For the entire distribution, we observe at least 1.5x improvement over ack=all mode, with the graph becoming skewed for max values. However, the performance of nil-ext deteriorates as compared to ack=1 for higher percentiles, as we need to wait for supermajority nodes to mark a produce request success.

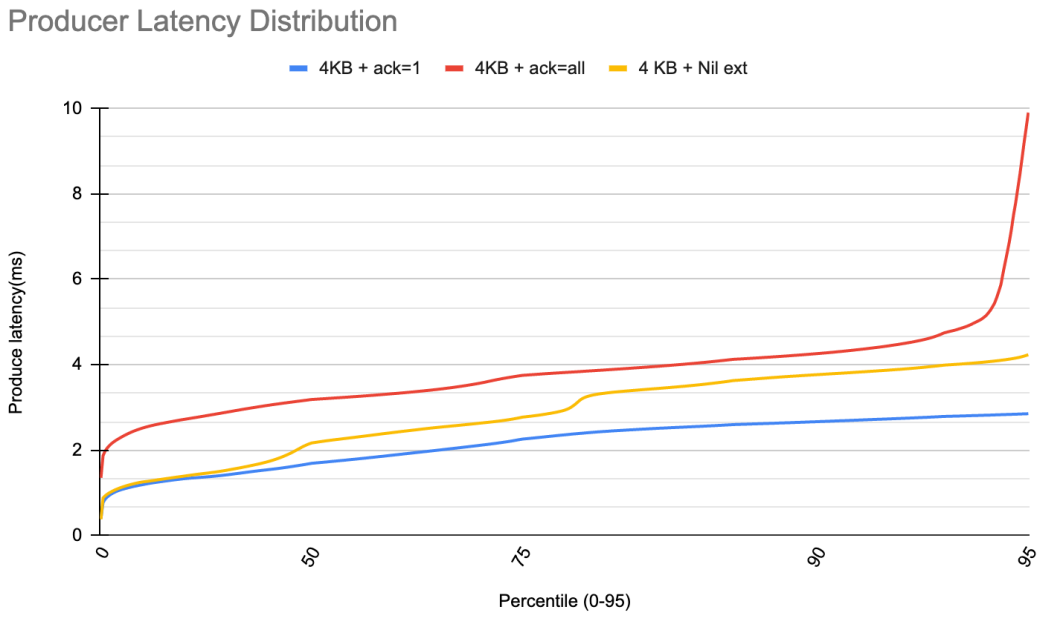


Figure 3: CDF plot of produce latency (ms) of different modes of Kafka with replication 3

4.2 Throughput and Latency

In this experiment, we reduced to network bandwidth to 100MB/s to stress test the system by increasing the production rate from 10 K/ops to 100 K/ops. The message size for each request is fixed to 1KB. The graph for this experiment is plotted in fig. 4. We make two important observations, first, we get latency benefit when there is ample network bandwidth, i.e. it is equivalent in performance to ack=1 mode. Second, even when under the stress of network bandwidth, nil-ext is performing better than highly consistent ack=all mode.

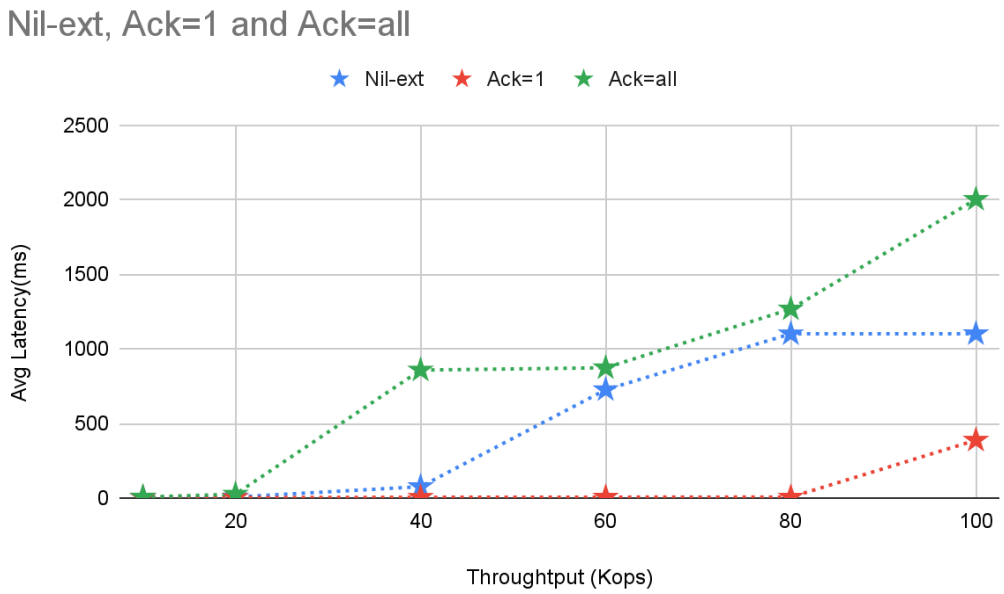


Figure 4: Latency vs Throughput plot, we increase produce rate till we hit the network limit of 100 MB/s

4.3 Scalability

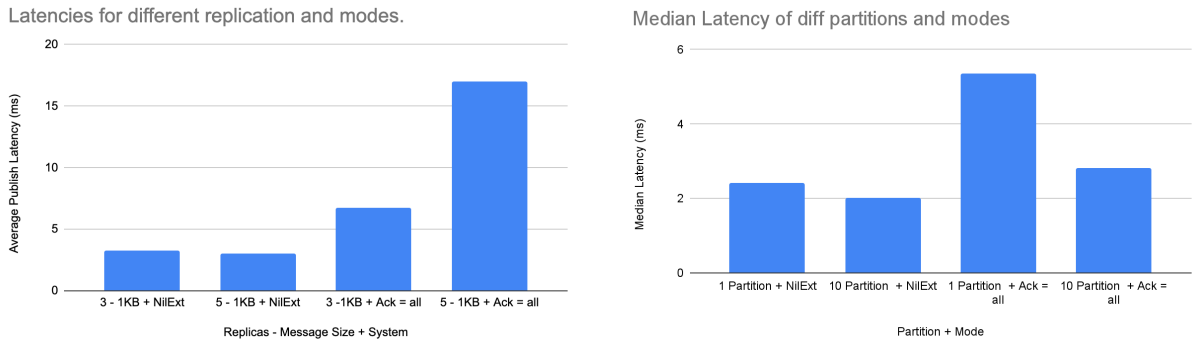


Figure 5: (left) 5a. Average latency with different replication and Kafka modes (right) 5b. Median latency with different number partitions(1 vs 10) and Kafka modes.

We set the message size as 1KB and ran workload with 10 Kops/sec on nil-ext and ack=all mode with a replication factor of 3 and 5(fig 5a). We observe that the average latency for both modes increases with an increase in the replication factor. However, we can observe that results are expected in both nil-ext and ack=all but performance degradation is not visible in nil-ext mode with a message size of 1KB. We also varied the number of partitions(fig 5b) by keeping the message size as 1 KB, throughput as 10 Kops/sec in nil-ext and ack=all mode. By

increasing the number of partitions, the kafka broker can accept more requests and we observe the average latency will be lower with a higher number of partitions. Figure 5b supports our hypothesis.

4.4 Impact of Slow Nodes

We slowed one node to half network bandwidth(50 MB/s) to calculate its impact of it on replication. As seen in fig 6, we were able to handle the situation much better than in ack=all mode. This is particularly helpful when there is a sudden network degradation in a data center, and ack = all mode can cause catastrophic slowdown.

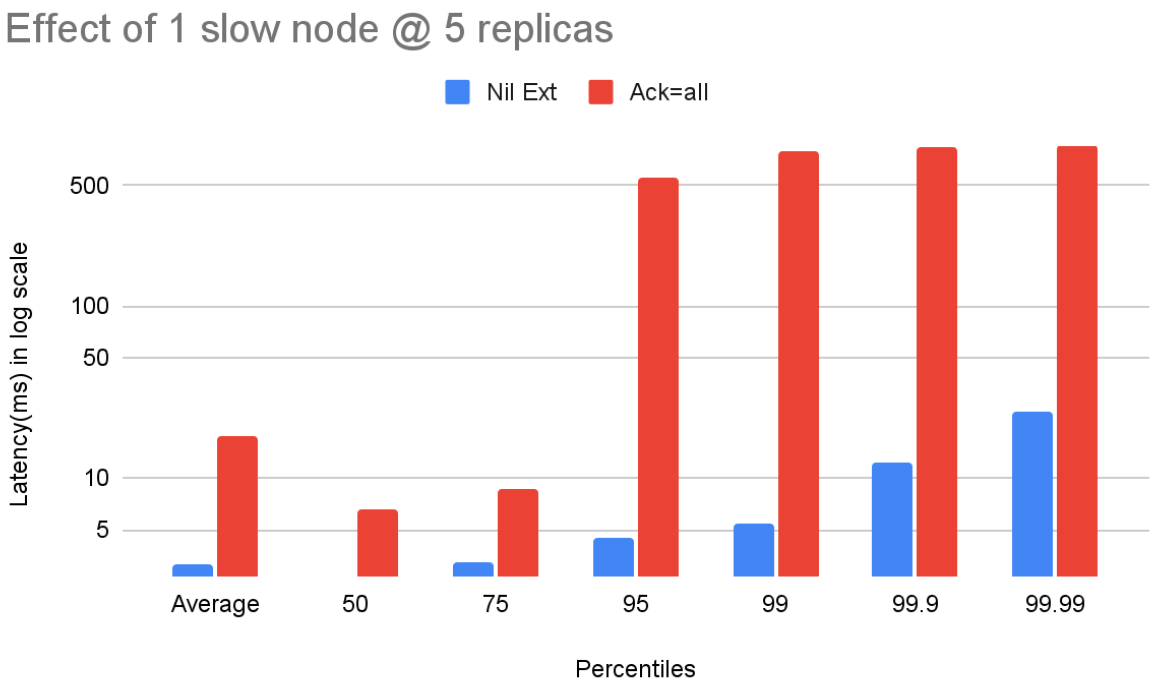


Figure 6: Measured latency when one node is slowed down

Section 4: Future Work

We plan to continue working on the project next semester, to complete the following features:

- Handle leader failure
- Try read from followers for externalized values

Section 5: Conclusion

We conclude this report with optimism on results obtained from our implementation of nil-ext interface on Kafka system. Although our system is not fully implemented we can still see improvement in performance for both latency and throughput, while maintaining a linearizability guarantee. We also show important protection of the system against a faulty slow node.

Code Submission and stats

We have submitted our fork of Kafka and benchmark, but given the extent of our implementation listing all changes are not feasible. Please let us know if we can submit it any other way.
Repos: Kafka and Benchmark

Appendix: Kafka Overview and Background

Apache Kafka is referred to as Kafka is an open-source, fast, scalable, fault-tolerant, highly available publish-subscribe messaging system and is mainly used for communication between large-scale distributed systems in the real world.

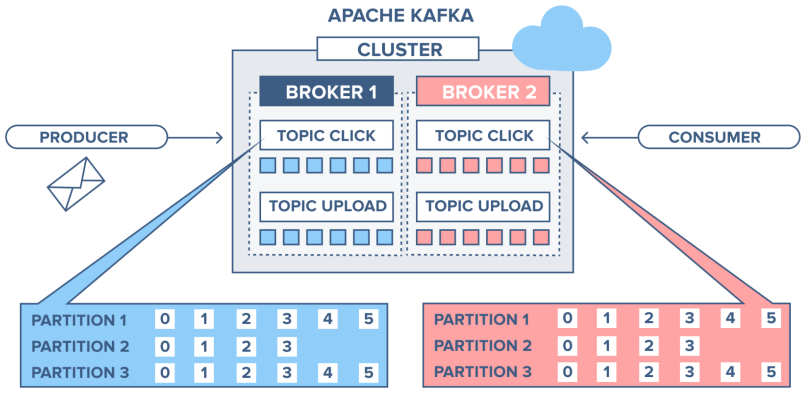


Fig1: Figure showing different components of Kafka

Kafka Overview:

In Kafka, the producer sends a message request to the cluster. Cluster store them and makes them available for the consumer if subscribed. The main components of Kafka are

1. Message
2. Topic
3, Producer
4. Consumer
5. Consumer Group
6. Cluster
4. Cluster
5. Consumer Group
6. Topic
7. Replicas
8. Partitions

Message: A message also known as a record, is the atomic unit of Kafka. It is usually associated with an event in the outside world. It usually contains a key, value, timestamp, and metadata of the event. An example of an event can be live location data. Ex: Fig 1 shows messages stored in Topics of Kafka Cluster

Topic: Kafka Cluster, organizes the messages received from different producers across systems as a collection of messages called Topics.

Replication: Each Topic is replicated in multiple brokers in Kafka Cluster, therefore it is highly resilient to failures. Whenever a broker goes down, topics of replicas can takeover and ensures data availability. The `replicationFactor` determines the number of copies of a topic

maintained in the Kafka Cluster. Ex: Fig 1 shows a topic is replicated on two brokers. It is recommended to have a minimum replication factor of 3.

Partitions: Each partition is divided into multiple partitions. Each partition is replicated across Kafka clusters based on the replication factor. Each partition in a topic can either be a leader or follower. For a partition, there will be a single leader and multiple followers. Each partition maintains an independent offset for storing and serving requests. Upon receiving a request Kafka stores the message in one of the partitions and increments the offset for that partition. Ex: Fig 1 shows the topic is divided into multiple partitions and each partition is replicated twice.

Producer: Kafka Producers are applications that publish messages to topics in the cluster. Producer Partitioner can map each message to a particular partition using the key in the message. Messages with the same key end up at the same partition. Each producer can publish to multiple topics. In Kafka, all writes are sent to the leader of the partition.

Cluster (Brokers): Kafka cluster comprises servers also known as brokers. The theory is responsible to accept message requests from producers and make them available for consumer/consumer groups. Ex: Fig 1 shows two brokers in Kafka clusters.

Consumer: Consumer is applications and subscribes for topics, read and process messages. A consumer can subscribe to messages individually or by batch. Each consumer reads the message, keeps track of the last read message, and updates the consumer_offset on the topic. Each consumer can subscribe to more than one partition.

Consumer Group: Consumer group is a set of independent consumers working together that subscribe to a topic. Each consumer in the group can read from more than one partition and are mutually exclusive i.e., two consumers in a group cannot read from the same partition. In the event of failure of the consumer, one of the remaining consumers will be assigned that partition and this is known as Consumer Rebalancing.

ACK Mode	Criteria
all	ACK from all replicas
1	ACK from leader partition
0	No ACK required

Table 1: Criteria for ack modes in Kafka

2.2 Kafka Interface:

Kafka producers consider a write based on the ack mode set before sending the request. Table 1 describes different criteria for a producer to consider a write successful. All

write requests are sent to the leader partition of the topic. When received a request, the master sends write requests to other replicas and waits for the acknowledgment based on the ack mode set by the producer. For a write to be successful in ack='all' mode, it takes a total of 2 RTTs whereas read requests take 1 RTTs since the message is already made durable.