The Key Ideas Behind Boki's Shared Logs

Zhipeng Jia
Google LLC
Seattle, WA, USA
zhipengjia@google.com

Emmett Witchel
The University of Texas at Austin
Austin, TX, USA
witchel@cs.utexas.edu

Abstract

The shared log approach has emerged as an attractive state management option for distributed systems. A shared log not only serves as persistent, strongly consistent, and fault-tolerant storage, its ability to provide a total order enables fine-grained state machine replication. Boki is a recent shared log system that includes an intuitive LogBook abstraction and novel shared log design choices. Despite Boki being designed as storage for serverless functions, its design principals are applicable to other distributed systems that disaggregate storage from compute.

1 Introduction

Distributed, shared, fault-tolerant logs [6, 9, 19] have emerged as a powerful tool in distributed systems to solve several difficult problems with a single, elegant abstraction. Logs provide persistent and fault-tolerant storage, but they also provide fine-grained state machine replication which forms the basis for important distributed services like consensus [5] and transactional data management [8]. A fault-tolerant distributed log enables distributed services to be built quickly and correctly, while also providing a single target for low-level, system optimizations.

Boki [11] brings shared logs to the serverless paradigm. For serverless applications, the total order provided by the shared log enables serverless functions to agree on the order of state updates, eliminating the need for complex coordination protocols. Moreover, the shared log acts as a reliable and durable storage layer, ensuring that state updates are persisted even in the presence of failures. While Boki was developed to support serverless computation, it is effective in any environment where compute can be scaled independently from storage, e.g., by adding additional virtual machines to a processing cluster.

Boki's physical shared log is divided into logical LogBooks that support an API (§ 3) similar to previous shared log systems [5, 6, 19]. However, Boki adds important new features including log tags and auxiliary data to speed up log-structured protocols. Log tags provide selective reads which are used to skip irrelevant records during log replay (§ 4.1). Auxiliary data is used as cache to store materialized states from log replay, so that new functions do not always need to replay the log from the beginning (§ 4.1.3).

Logs are a write-optimized data structure, so systems that use them often require read optimizations for higher performance. For example, the log-structured file system [15] kept its index entirely in memory, and subsequent systems have added features like substreams [7, 19] to reduce the work of reconstructing state from a large number of updates. Boki's log tags are metadata in the form of a list of strings. We show that log tags are a simple mechanism not only to make selective reads faster, they also enable atomic updates to multiple, independent logical streams that are stored in the same shared physical log.

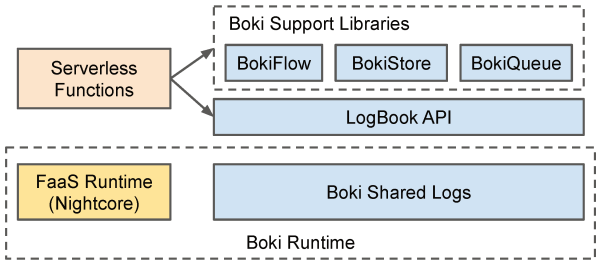


Figure 1. Boki overview (§ 2).

Boki takes inspiration from previous shared log systems [5–7, 9, 19], but also proposes two novel design choices to scale shared logs while providing consistency guarantees. The first design choice is using a log-structured approach for maintaining Boki's internal metadata. Boki internally uses a data structure, called the *metalog*, to simultaneously provide mechanisms for log ordering, read consistency, and fault tolerance (§ 5.1). The second design choice is to build separate indices for the log. Boki's log index is the key enabler for log tags and LogBook multiplexing. Moreover, the log index design integrates with the metalog to provide read consistency guarantees (§ 5.2).

The remainder of this paper is structured as follows: Section 2 provides an overview of Boki components. Section 3 explains LogBook API in details. Section 4 demonstrates how to use the LogBook API for two fault-tolerance paradigms: exactly-once execution via write-ahead logging and state machine replication. Section 4 also explains how to speed up log-structured protocols via log tags and auxiliary data. Section 5 describes key design elements in Boki: the metalog and the log index. Section 6 concludes.

2 Overview of Boki

Boki's design combines a FaaS system with shared log storage (depicted in Figure 1). Boki's implementation is based on Nightcore [12], a state-of-the-art FaaS system for microservices, while adding new components to support shared logs.

```
struct LogRecord {
    uint64_t seqnum;          string data;
    vector<tag_t> tags;       string auxdata;
};

// Append a new log record.
status_t logAppend(vector<tag_t> tags, string data,
                   uint64_t* seqnum);

// Read the next/previous record whose seqnum >=
// `min_seqnum`, or <= `max_seqnum`. Log reads guarantee
// "monotonic reads" and "read-your-writes" semantics.
status_t logReadNext(uint64_t min_seqnum, tag_t tag,
                     LogRecord* record);
status_t logReadPrev(uint64_t max_seqnum, tag_t tag,
                     LogRecord* record);

// Alias of logReadPrev(kMaxSeqNum, tag, record).
status_t logCheckTail(tag_t tag, LogRecord* record);

// Trim the LogBook until `trim_seqnum`, i.e., delete
// all log records whose seqnum < `trim_seqnum`.
status_t logTrim(uint64_t trim_seqnum);

// Set auxiliary data for the record of `seqnum`.
status_t logSetAuxData(uint64_t seqnum, string auxdata);
```

Figure 2. Boki's LogBook API (§ 3).

Boki provides a LogBook abstraction for its serverless functions to access shared logs (§ 3). Boki applications create LogBooks shared between functions to manage states. Internally, Boki maintains a small number of physical, independent, totally ordered logs. Application-facing LogBooks are multiplexed on these physical logs for better resource efficiency.

To simplify stateful serverless applications using shared logs, Boki also includes support libraries on top of the LogBook API aimed at three different serverless use cases: BokiStore for durable object storage, BokiQueue for message queues, and BokiFlow for fault-tolerant workflows with exactly-once execution.

3 LogBook API

Boki provides a distributed shared log accessible via a simple API, shown in Figure 2. Like previous shared log systems [5, 6, 9, 19], Boki exposes *append*, *read*, and *trim* APIs for writing, reading, and deleting log records. These APIs apply to logical LogBooks, which are sequences of data records contained in a physical log. Applications or application components that wish to coordinate application state with strong consistency and fault tolerance use the same LogBook.

Two important features of the API that distinguish Boki from previous work is the list of tags and the auxiliary data stored with each log record. Discussed in detail below, the list of tags enables atomic writes to multiple LogBooks. The auxiliary data allows Boki clients to share materialized views of the log. Previous systems like Tango [7] built materialized views, but only in thread-local memory.

Seqeunce numbers (seqnum). In a LogBook, every log record has a unique sequence number (referred to as seqnum) establishing the total order of records. As illustrated in Figure 2, the `logAppend` API returns the seqnum for the newly appended log record. Seqnums are monotonically increasing within a LogBook but *not* guaranteed to be consecutive. `logReadNext` is used for forward log read, while `logReadPrev` is used for backward log read. Given the non-consecutive nature of seqnums, these read APIs necessitate specifying the lower or upper bound of the seqnum for the read operation.

Log tags. Tagging log records with arbitrary strings is a unique feature of Boki shared logs that enables selective log reads. Each log record in the LogBook can be associated with a set of string tags. Log tags are immutable metadata of a log record, which can only be specified when appending the log, i.e. the `logAppend` API (see Figure 2). To selectively read the log via tags, both `logReadNext` and `logReadPrev` APIs accept a tag argument, meaning only log records with the given tag are considered. Although a single tag is sufficient for selective reads, every log entry can have multiple tags. Supporting multi-tagging enables certain important use cases, e.g., atomic group updates (explained in § 4.1.1).

Consistency guarantees. When a LogBook is shared by multiple clients, Boki ensures *sequential consistency* for LogBook operations. Guarantees of sequential consistency include total write order, monotonic reads/writes, and read-your-writes. A LogBook's total write order is reflected in the monotonically increasing seqnums of log records. Access to a total order of state updates allows different clients to agree on the final outcome of deterministic computations. Sequential consistency does not guarantee real-time visibility of new log records, but read-your-writes ensures immediate visibility in the producing client. Moreover, monotonic reads ensure that once a client reads some log record, all log records having a smaller seqnum in the same LogBook become visible. Monotonic reads means that any computation invoked by a client sees at least as much of the log as its invoker could see, which is often an implicit assumption within applications.

Auxiliary data. LogBook's auxiliary data is designed as per-log-record cache storage, which is set by the `logSetAuxData` API. When performing log reads, auxiliary data is returned if was ever set and is still resident in the cache (see the return struct of `logReadPrev` and `logReadNext` APIs). Auxiliary data can cache object views in a shared-log-based object storage. These cached object views will significantly reduce log replay overheads (§ 4.1.3).

Auxiliary data is only used as a cache, so Boki does not guarantee its durability. It provides only best effort support. Moreover, Boki does not maintain the consistency of auxiliary data, i.e., Boki trusts applications to provide consistent auxiliary data for the same log record. The desired usage pattern of auxiliary data is to cache states related to log replay,

```python
# Increase a single counter by delta
def counter_inc(name: str, delta: int):
  logAppend(tags: [name],
            data: {name: delta})

# Increase multiple counters atomically
def counter_multi_inc(names: list[str],
                      deltas: list[int]):
  data = {}
  for name, delta in zip(names, deltas):
    data[name] = delta
  logAppend(tags: names, data: data)

# Get the value of a single counter
def counter_get(name: str) -> int:
  value = 0
  pos = 0
  while True:
    record = logReadNext(tag: name,
                         min_seqnum: pos)
    break if record == None
    value += record.data[name]
    pos = record.seqnum + 1
  return value
```

Figure 3. Implementing counters using LogBook APIs (§ 4.1). In this example, the log record stores counter increase commands that modify one or multiple counters. The log record is tagged with names of modified counters, and its data field stores a map where keys are counter names and values are deltas to counters.

which should be uniquely determined by a deterministic process. Relaxing durability and consistency allows Boki to store auxiliary data in node-local in-memory caches without any mechanism for consistency.

4 Using the LogBook API

In the literature of distributed systems, many log-structured protocols are proposed to achieve strong consistency and fault tolerance [5, 7, 8, 14, 17–20]. Boki's LogBook API is designed to implement common log-structured protocols with minimum development effort.

In this section, to demonstrate the usability of the LogBook API, we use two important paradigms as examples: (1) state machine replication (SMR) for shared data structures, and (2) write-ahead logging (WAL) for exactly-once execution. For SMR-based protocols, we further explain three key techniques unique to Boki: (1) how LogBook's multi-tagging enables atomic group updates, (2) how to achieve linearizability, and (3) how auxiliary data speeds up log replay.

4.1 State machine replication (SMR)

State machine replication (SMR) [16] is a paradigm for fault tolerance, where application state is replicated across servers using a log of commands. The command log is traditionally backed by consensus algorithms such as Paxos [13, 18] or Raft [14], but recent studies show a shared log can provide an efficient abstraction to support SMR-based data structures [7, 19] and protocols [5, 8]. When using a shared log to implement SMR, state machine commands are ordered and persisted by the shared log.

Figure 3 illustrates an SMR-based counter implementation backed by a LogBook. In this example, a counter is an integer which can be increased or decreased by a delta value. The implementation allows multiple counters, which are identified by string names. The *counter_inc* function increases a counter by an integer delta. Using a negative delta value effectively decreases the counter. The *counter_inc* function appends a new log record representing the execution of the counter increase command using the delta value. The *counter_get* function re-constructs the current state of the counter by replaying the log. Log records for counter increase commands are tagged with the name of the counter being modified, which enables selective log reads in the *counter_get* function.

Boki includes two support libraries, BokiStore and BokiQueue, to provide SMR-based data structures for serverless applications. Shared data structures like maps and queues are common in distributed applications. Strong consistency provided by SMR means that any code that uses these data structures will work correctly. Code often makes implicit assumptions about the semantics of data structures, for example, that an item enqueued after another is also dequeued after that item.

BokiStore implements a strongly consistent object store, similar to Tango [7], which enables applications to store data structures with strong consistency and fault tolerance. Objects are identified by unique string names and are represented as JSON objects. All object update commands are stored in the shared LogBook, and the total order provided by the LogBook becomes the single source of consistency. Similar to the previous counter example, log records are tagged with object names to allow selective log reads for re-constructing objects. BokiStore also supports multi-object transactions, with a transaction commit protocol derived from Tango but taking advantages of log tags.

BokiQueue provides a simple push and pop API for sending and receiving messages. Like BokiStore, BokiQueue uses the log to store all push and pop commands, and the outcome of each operation is determined by replaying the history stored in the log. Applications make sure that results of log replay are deterministic.

4.1.1 Atomic group updates via multi-tagging.

Log tags enable selective reads, which improves log replay performance. But when we want to update multiple SMR-backed objects together, it becomes challenging to achieve atomicity. Consider the counter implementation in Figure 3. Suppose there are two counters a and b that represent two different bank balances. We might want to effect a transfer of funds by decrementing a by 100 while atomically incrementing b by

the same amount. We must take care to do the entire action or none of it to ensure that funds are not lost or (erroneously) created.

If we use the *counter_inc* function in Figure 3 to independently change both counters, there will be two log records in the LogBook. The LogBook read API cannot guarantee these two log records are visible atomically for the reader, i.e., the reader might see the update for counter *a* but not counter *b*.

Boki's multi-tagging feature (i.e., allowing one log record to have multiple tags) provides the mechanism to address such a challenge. The *counter_multi_inc* function in Figure 3 demonstrates the approach. In this case, a single log record corresponds to updates of multiple counters, and the record is tagged with names of all involved counters. When a reader reads counters *a* and/or *b*, they will get this single record that has the update of both counters. The single-record nature of the update ensures the atomicity of the multiple updates.

The multi-tagging feature also plays a critical role in BokiStore's transaction protocol. In the protocol, the *txn_commit* record is tagged with object names in the transaction write set, which ensures the commit record appears atomically in the update streams of all involved objects. At a high level, Boki's multi-tagging feature provides similar functionality to multi-stream appends in vCorfu [19], but without any coordination protocol.

4.1.2 Linearization challenge.

Updates to our counter (Figure 3) only record the delta, making them conflict-free or commutative. If a single counter is incremented by 10 and then by 20, the final value is independent from the update order.

A lock is an important data structure whose operations are not commutative. The lock state machine has two states: *empty* and *acquired*. An *acquire* command can only succeed when the lock is in the *empty* state, and vice versa. Suppose a program wants to acquire the lock. It must first replay the log to reconstruct the current lock state. If the lock is in the *empty* state, the next step is to append an *acquire* command to the log. But if there are multiple contending threads, they will append multiple *acquire* commands. Which participant should obtain the lock? and how do all of the threads (efficiently) agree on the identity of the winner?

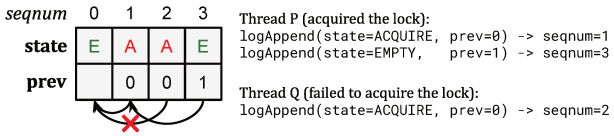
To obtain a correct lock implementation using only log operations, the system must be able to linearize [10] the operations. Surprisingly, we can achieve linearizability for locks without complicating the log API. Intuitively, the solution is to include the tail seqnum of the current state machine when appending the proposed update commands. If every participant records their view of the tail, that ensures that there is enough information in the log records for all participants to agree on a unique outcome. While replaying the log, participants choose only the first of any updates that were concurrently proposed.

```
def check_lock_tail(lock_key: str):
  tail, pos = None, 0
  while True:
    record = logReadNext(tag: lock_key,
                         min_seqnum: pos)
    break if record == None
    if record.data["prev"] == tail.seqnum:
      tail = record
    pos = record.seqnum + 1
  return tail

def try_acquire_lock(lock_key: str) -> bool:
  tail = check_lock_tail(lock_key)
  if tail.data["state"] == EMPTY:
    seqnum = logAppend(
        tags: [lock_key],
        data: {"state": ACQUIRED,
               "prev": tail.seqnum})
    tail = check_lock_tail(lock_key)
    if tail.seqnum == seqnum:
      return True  # Lock succeeded
  return False     # Lock failed

def release_unlock(lock_key: str):
  tail = check_lock_tail(lock_key)
  assert tail.data["state"] == ACQUIRED
  logAppend(tags: [lock_key],
            data: {"state": EMPTY,
                   "prev": tail.seqnum})
```

(a) Pseudocode of lock operations.



(b) An example log for the above lock implementation. Two threads P and Q are contending to acquire a lock. Thread P wins due the its log record has smaller seqnum.

Figure 4. Implementing locks using LogBook APIs (§ 4.1.2).

A linearizable lock is shown in Figure 4. In this example, there are two concurrent lock acquire attempts, resulting two log records, both having their *prev* fields equal to 0. The first record in the log (seqnum=1) wins the lock. The second record (seqnum=2) is a failed acquire attempt, and will be ignored when replaying the log. Later, the lock is released by the log record with seqnum=3.

We note this solution not only applies to two-state state machines (e.g., locks), but it provides a generic approach to linearize commands for any state machine when using LogBook APIs.

4.1.3 Speeding up log replay using auxiliary data.

Reads in BokiStore are handled by replaying the log to re-construct object state. This naive approach makes read latency proportional to the number of relevant log records, i.e., the

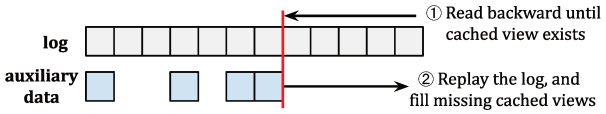


Figure 5. Use auxiliary data to cache object views in BokiStore, which can avoid a full log replay (§ 4.1.3).

number of object writes. Tango [7] optimizes log replay by caching local object views, such that only new records from the shared log are replayed. However, this thread-local, in-memory cache has two disadvantages: it is not present in some environments like serverless, and it is not amortized over multiple threads.

In BokiStore, object writes are logged as update commands in the shared log. For each update command, its auxiliary data stores a snapshot view of the modified object. When reading an object, BokiStore seeks back from the log tail to find the first relevant record having a cached object view in its auxiliary data. Then BokiStore replays the log from this position to re-construct the target object state. Figure 5 demonstrates this accelerated replay process. During replay, for records missing cached object views, threads use the `logSetAuxData` API to fill the auxiliary data with the updated object views. Note that there is no coordination for setting the auxiliary data because SMR by itself ensures object views are always same for the same log position.

txn_commit records use auxiliary data to cache the decided commit outcome. If the transaction commit succeeds, the auxiliary data of the *txn_commit* also caches a view of modified objects.

4.2 Exactly-once execution

Workflows composing multiple steps are an important paradigm in distributed computing (e.g., serverless workflows [4]). A workflow is a directed graph of computations where data flows between nodes via the log. Workflows often interact with cloud data storage (e.g., Amazon S3 [3] or DynamoDB [1]) to manage their application state. While workflows can be connected with fault-tolerant queues (e.g., Amazon SQS [2]), fault tolerance becomes challenging in such scenario, because functions of a workflow can fail in the middle of execution, leaving partial application states in cloud storage. These partial states can cause records to be processed multiple times or not at all when failures occur.

The key guarantee of stateful workflows is exactly-once execution semantics, meaning even in face of failures, a workflow instance will eventually get executed exactly once as if no failure has happened. Previous work proposes write-ahead logging (WAL) to achieve exactly-once execution semantics: Olive [17] proposes a client library interacting with cloud storage, where a write-ahead redo log is used for recovery in case of failures. Beldi [20] extends Olive's log-based techniques for transactional serverless workflows.

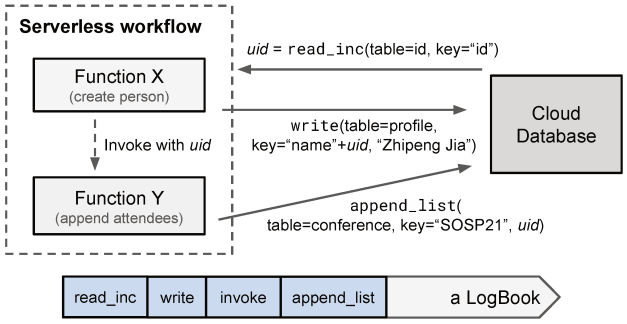


Figure 6. Use logging to achieve exactly-once execution for serverless workflows (§ 4.2).

To better support serverless workflows, Boki includes a library, BokiFlow, which provides exactly-once execution semantics with fault tolerance by using LogBook APIs. BokiFlow resembles Olive and Beldi's logging techniques, which log every operation having externally visible effects (e.g., a database write). When failure happens, the log is used to provide exactly-once guarantees, e.g., by avoiding duplicated database updates.

Figure 6 shows an example serverless workflow for conference registration. This example has two serverless functions: one to create a person's profile, and the other for adding a person to the conference attendance list. Consider a failure that happens in between the two functions. If we simply re-execute the workflow, the database may end up with duplicated people profiles. To provide exactly-once semantics, BokiFlow uses a LogBook to log database updates and function invocations before and after these operations get executed. When failure happens, the log is used to re-execute the workflow while guaranteeing exactly-once execution semantics.

5 Boki efficiency techniques

The API to append to a shared, distributed, fault-tolerant log is simple, but the underlying implementation has to achieve high throughput with global total order, as well as support flexible log reads with consistency guarantees. To achieve high throughput, concurrent log appends by independent threads and nodes are buffered, then replicated to distributed storage. At the same time, sequencers order the log and make the total order visible to log readers [9]. To support flexible log reads, Boki builds indices for the log, while providing a mechanism for consistent log reads.

5.1 Metalog for log ordering

At the core of the log ordering mechanism, Boki employs a log-structured approach: Boki maintains a data structure called the *metalog* for ordering metadata updates to a physical log. Each metalog entry stores a cut vector which determines the global total order between multiple, concurrent

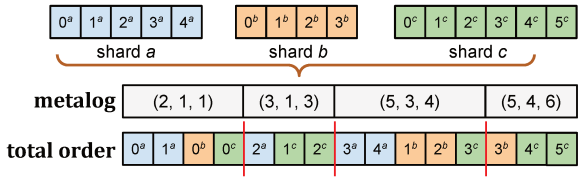


Figure 7. An example showing how the metalog determines the total order. A physical log in Boki is sharded, and the metalog uses cut vectors to order records across log shards. Each element of a cut vector corresponds to a log shard. In the figure, log records between two red lines form a delta set, which is defined by two consecutive vectors in the metalog (§ 5.1).

log shards. Shown in Figure 7, each log shard maintains a local order of its log records, and a cut vector encodes the progress of all shards, which are periodically appended to the metalog. In this way, new updates to the metalog essentially extend the underlying physical log, which also enables log indices to be built by subscribing to metalog updates.

To achieve fault tolerance, every metalog is replicated on a fixed number of sequencer nodes (3 in the prototype). One of the sequencers is configured as primary, and only the primary sequencer can append to the metalog. To append a new metalog entry, the primary sequencer sends the entry to all other sequencers for replication. Once acknowledged by a quorum, the new metalog entry is considered as successfully appended. If any sequencer node fails, reconfiguration is triggered.

5.2 Log index for selective log reads with consistency

Boki builds log indices to enable multiplexing LogBooks on internal physical logs and selective reads via log tags. The structure of the log index is designed to fit the semantic of LogBook read APIs (see Figure 2). The log index first groups records by their LogBook identifiers (*book_id*) and log tags, because a read can only target a single LogBook and a single log tag. For each (*book_id*,*tag*) pair, it builds an index row that includes seqnums of log records matching *book_id* and *tag*. Given that LogBook read APIs seek for records sequentially using lower or upper bounds for seqnums, the index row is a sorted list of seqnums, allowing a binary search to locate the target seqnum. When a log record has multiple tags, its seqnum will appear in multiple index rows. Figure 8 depicts the workflow of LogBook reads using the index.

From the LogBook read workflow, read consistency is determined by the log index, because the index is first used to locate the seqnum of log record to read. To scale read throughput, Boki builds multiple index replicas, creating the challenge to enforcing read consistency among replicas. To achieve read consistency, Boki takes advantage of the fact that log indices are built by subscribing the metalog, meaning the metalog position of an index replica determines its relative freshness. Recall that LogBook reads should provide monotonic reads and read-your-writes (§ 3). For a log reader,

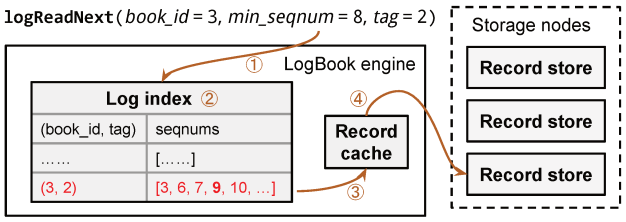


Figure 8. Workflow of LogBook reads (§ 5.2): ① Locate a LogBook engine stores the index for the physical log backing *book_id* = 3; ② Query the index row (*book_id*, *tag*) = (3, 2) to find the metadata of the result record (seqnum = 9 in this case); ③ Check if the record is cached; ④ If not cached, read it from storage nodes.

to guarantee monotonic reads, it must use log indices with monotonically increasing metalog positions. Similarly, to guarantee read-your-writes, a log reader must only use the log index that already caught up to its latest log writes, i.e., the index includes the seqnum of the latest write.

To realize such mechanism, a log reader maintains a metalog position to represent its most recently read position in the log. Before reading from an index replica, the log reader will wait for the index to catch up if the replica's metalog position is before the reader's. After the read, the log reader will update its metalog position to guarantee monotonic reads for subsequent reads. Log writes also result in updating the metalog position to include log records just appended, in order to guarantee read-your-writes for future reads. Figure 9 illustrates this mechanism.

6 Conclusion

State management remains to be a key challenge for distributed systems, especially when fault tolerance and strong consistency are required. Shared logs prove to be an effective solution in such challenging scenario. A shared log not only scales to high write throughput, its total order property naturally enables state machine replication [16], the generic, widely-used paradigm for fault-tolerant services.

Modern distributed systems are moving to disaggregation architectures, where machines for data storage are often disaggregated from machines for compute. One prominent example is function-as-a-service (FaaS) in cloud computing. Boki is an attempt to provide shared logs in a FaaS environment, aiming at helping stateful serverless applications to manage their state with fault tolerance and strong consistency.

Boki designs an intuitive LogBook abstraction for applications to adopt log-structured protocols including write-ahead logging and state machine replication. To speed up log-structured protocols, the LogBook abstraction introduces log tags for selective reads, and auxiliary data to cache log replay states. The LogBook abstraction also allows multiple tags to be associated with one log record, serving as an elegant solution for atomic group updates and complex

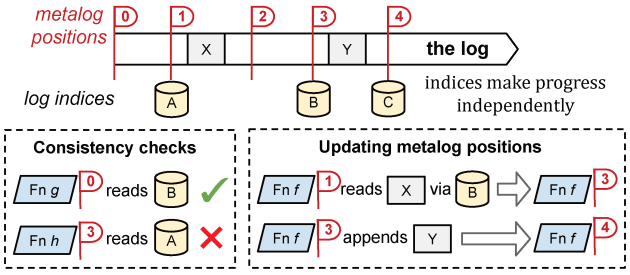


Figure 9. Consistency checks by comparing metalog positions (§ 5.2). For a log reader (serverless function in this example), if reading from a log index whose progress is behind its metalog position, it could see stale states. For example, function *h* have already seen record *X*, so that it cannot perform future log reads through index *A*.

log-based protocols like transaction commits. Despite the simplicity of the LogBook API, it can achieve strong consistency properties including linearizability.

To scale shared logs while providing consistency guarantees, Boki internally employs a log-structured mechanism, the metalog, to manage its own metadata. The metalog design provides a unified mechanism to address log ordering, read consistency, and fault tolerance. To enable selective log reads while scaling read throughput, Boki builds indices for shared logs. Boki's log index serve as the first step to read any log data, such that the index actually ensures read consistency such as monotonic reads and read-your-writes.

References

[1] Amazon DynamoDB | NoSQL Key-Value Database | Amazon Web Services. [Accessed Apr, 2022]. URL: https://aws.amazon.com/dynamodb/.

[2] Amazon SQS | Message Queuing Service | AWS. [Accessed Apr, 2022]. URL: https://aws.amazon.com/sqs/.

[3] Cloud Object Storage | Store and Retrieve Data Anywhere | Amazon Simple Storage Service (S3). [Accessed Jan, 2021]. URL: https://aws.amazon.com/s3/.

[4] Workflows | Google Cloud. [Accessed Apr, 2022]. URL: https://cloud.google.com/workflows.

[5] Mahesh Balakrishnan, Jason Flinn, Chen Shen, Mihir Dharamshi, Ahmed Jafri, Xiao Shi, Santosh Ghosh, Hazem Hassan, Aaryaman Sagar, Rhed Shi, Jingming Liu, Filip Gruszczynski, Xianan Zhang, Huy Hoang, Ahmed Yossef, Francois Richard, and Yee Jiun Song. Virtual consensus in delos. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 617–632. USENIX Association, November 2020. URL: https://www.usenix.org/conference/osdi20/presentation/balakrishnan.

[6] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobbler, Michael Wei, and John D. Davis. CORFU: A shared log design for flash clusters. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 1–14, San Jose, CA, April 2012. USENIX Association. URL: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/balakrishnan.

[7] 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. doi:10.1145/2517349.2522732.

[8] Mahesh Balakrishnan, Chen Shen, Ahmed Jafri, Suyog Mapara, David Geraghty, Jason Flinn, Vidhya Venkat, Ivailo Nedelchev, Santosh Ghosh, Mihir Dharamshi, Jingming Liu, Filip Gruszczynski, Jun Li, Rounak Tibrewal, Ali Zaveri, Rajeev Nagar, Ahmed Yossef, Francois Richard, and Yee Jiun Song. Log-structured protocols in delos. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 538–552, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3477132.3483544.

[9] 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 *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 325–338, Santa Clara, CA, February 2020. USENIX Association. URL: https://www.usenix.org/conference/nsdi20/presentation/ding.

[10] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, jul 1990. doi:10.1145/78969.78972.

[11] Zhipeng Jia and Emmett Witchel. Boki: Stateful serverless computing with shared logs. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 691–707, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3477132.3483541.

[12] Zhipeng Jia and Emmett Witchel. Nightcore: Efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, page 152–166, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3445814.3446701.

[13] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 358–372, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2517349.2517350.

[14] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, Philadelphia, PA, June 2014. USENIX Association. URL: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro.

[15] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, February 1992. doi:10.1145/146941.146943.

[16] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990. doi:10.1145/98163.98167.

[17] Srinath Setty, Chunzhi Su, Jacob R. Lorch, Lidong Zhou, Hao Chen, Parveen Patel, and Jinglei Ren. Realizing the fault-tolerance promise of cloud storage using locks with intent. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 501–516, Savannah, GA, November 2016. USENIX Association. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/setty.

[18] Robbert Van Renesse and Deniz Altinbuken. Paxos made moderately complex. *ACM Comput. Surv.*, 47(3), February 2015. doi:10.1145/2673577.

[19] Michael Wei, Amy Tai, Christopher J. Rossbach, Ittai Abraham, Maithem Munshed, Medhavi Dhawan, Jim Stabile, Udi Wieder, Scott Fritchie, Steven Swanson, Michael J. Freedman, and Dahlia Malkhi. vcorfu: A cloud-scale object store on a shared log. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 35–49, Boston, MA, March 2017. USENIX Association. URL: https://www.usenix.org/conference/nsdi17/technical-

Zhipeng Jia, Emmett Witchel„
sessions/presentation/wei-michael.

[20] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu. Fault-tolerant and transactional stateful serverless workflows. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1187–1204. USENIX Association, November 2020. URL: https://www.usenix.org/conference/osdi20/presentation/zhang-haoran.