JEPSEN

Bufstream 0.1.0

Kyle Kingsbury
2024-11-12

Bufstream is a Kafka-compatible streaming system which stores records directly in an object storage service like S3. We found three safety and two liveness issues in Bufstream, including stuck consumers and producers, spurious zero offsets, and the loss of acknowledged writes in healthy clusters. These problems were resolved by version 0.1.3. We also characterize four issues related to Kafka more generally, including the lack of authoritative documentation for transaction semantics, a deadlock in the official Java client, and write loss, aborted read, and torn transactions caused by the lack of message ordering constraints in the Kafka transaction protocol. These issues affect Kafka, Bufstream, and (presumably) other Kafka-compatible systems, and remain unresolved. A companion blog post from Buf is available as well. This report was funded by Buf Technologies, Inc. and conducted in accordance with the Jepsen ethics policy.

1 Updates

2024-11-13: An earlier version of this report asserted that consumers with auto-commit enabled could lose data by automatically committing offsets returned by the most recent poll. This claim was based in part on Confluent's documentation, which states:

> When auto-commit is enabled, every time the poll method is called and data is fetched, the consumer is ready to automatically commit the offsets of messages that have been returned by the poll. If the processing of these messages is not completed before the next auto-commit interval, there's a risk of losing the message's progress if the consumer crashes or is otherwise restarted.

This perspective is echoed by Red Hat and New Relic. Jepsen observed data loss in tests using auto-commit, and assumed that this was, in fact, intended behavior.

However, several readers objected that auto-commit consumers do not in fact commit the offsets of the most recent poll. Rather, they commit the offsets of the *penultimate* poll—the poll before the most recent poll. Brief experiments with the Java Kafka client appear to support this claim. To further complicate matters, Buf indicates that some clients commit offsets of the penultimate poll, and other clients use the most recent poll. We have removed specific claims regarding auto-commit from this report; more research is warranted.

2 Background

Kafka is a popular streaming system which provides replicated, sharded, append-only logs. Bufstream is a drop-in replacement for Kafka designed to prioritize data governance and cost efficiency in cloud environments.

Like Kafka, Bufstream provides a collection of named, partially ordered logs called *topics*. Each topic is divided into *partitions*.[1] Each partition is a totally ordered, append-only list of *records* (also called *messages* or *events*). Within a partition, each record is uniquely identified by a monotonically advancing integer *offset*. Offsets may be sparse: some offsets are used for storing internal metadata and are invisible to clients.

Bufstream works with standard Kafka clients. There are two main types of clients in Kafka-compatible systems. *Producers* append records to partitions by calling `producer.send()`. *Consumers* read those records. Consumers are first bound to partitions via `consumer.assign()` or `consumer.subscribe()` operations.[2] Once bound, one repeatedly calls `consumer.poll()` to receive records from any of those partitions. Each consumer can belong to a *consumer group*, which shares responsibility for processing records from a set of topics.

Each partition has a *last stable offset (LSO)*, which is the highest offset below which every transaction has completed. It also has a *committed offset* for each consumer group, which is the highest offset below which

[1] In this article, the word *partition* can refer to either a network omission fault ("network partition") or an ordered log within a Bufstream topic ("topic-partition").

[2] The `assign` operation informs a consumer that it should poll records from a specific set of topic-partitions. The consumer starts at some position in each of those partitions and advances sequentially through them. The `subscribe` operation takes only a topic, and allows Kafka to automatically determine which partitions are bound to the consumer. These allocations are dynamically balanced between all active, subscribed consumers.

that consumer group has processed all records in the partition.[3]

As in Kafka, records are opaque blobs of bytes by default. However, Bufstream can integrate with the Buf Schema Registry to introspect Protocol Buffer records. This allows Bufstream to validate records before committing them, enforce field-level access control policies, and reformat data to interoperate with other systems. Unlike Kafka, which stores data on local disks and has its own replication protocol, Bufstream writes its data directly to an object storage service. By relying on object storage, which often bundles the cost of replication traffic, Bufstream aims to reduce costs. This also allows Bufstream nodes to run as stateless, auto-scaled VMs.[4]

Bufstream comprises three subsystems: an *agent*, an *object store*, and a *coordination service*. The agent is a stateless service which provides the Kafka API. Clients connect to agents to publish and consume records. The object store (e.g. S3) stores chunks of records as they are written, and makes them available to readers. The coordination service (presently etcd) helps agents establish which chunks in storage are committed, and what the order of records should be.

As of October 2024, Bufstream was deployed only with select customers. Its documentation claimed to be "a drop-in replacement for Apache Kafka," and listed compatibility with Kafka's transactions and exactly-once semantics. However, there were few specific safety claims beyond Kafka compatibility. Throughout this work we use Kafka's documentation as our benchmark for evaluating Bufstream.

2.1 Client Safety

Like Kafka, Bufstream is intended for a variety of streaming applications with different throughput, latency, and safety tradeoffs. As in Jepsen's previous work on Kafka-compatible systems, we set a variety of client configuration options to obtain safer behavior.

We generally used the default `acks = all` for our producers. In Bufstream, `acks = 0` allows the server to acknowledge a write immediately without waiting for storage. As in Kafka, this may lose committed writes. Both `acks = 1` and `acks = all` block until Bufstream is certain the write is durably persisted.

Kafka producers can automatically retry writes. We used the default setting `enable.idempotence = true` to prevent appending multiple copies of a record to the log.

By default, Kafka consumers may automatically mark offsets as committed. Some documentation indicates that this could lead to data loss. We generally tested with `enable.auto.commit = false` to avoid this possibility.

When a consumer subscribes to a topic, it starts at the last committed offset. If no offset has been committed, it defaults to the most recent offset. This is another way in which Kafka's defaults do not ensure at-least-once delivery. We used `auto.offset.reset = earliest` to make sure consumers had a chance to observe the entire log.

2.2 Transactions

Bufstream supports Kafka's transaction system. Kafka transactions have complex semantics determined by a wide array of configuration settings and the specific calls executed by producer and consumer. As we discussed two years ago, Kafka's official documentation largely omits any description of transaction invariants. Instead, documentation remains scattered across various blog posts, Wiki pages, Kafka Improvement Proposals, Google Docs, introductory guides, and the Java client's API documentation. These resources are often confusing, underspecified, contradictory, or outright wrong. They generally focus on implementation mechanics rather than invariants. We have inferred Kafka's intended transaction semantics as best we can from these documents and observed behavior.

In broad terms, a Kafka transaction comprises a set of records sent by a producer and a map of partitions to the maximum offsets polled by a consumer. Transactions provide a weak form of atomicity. If and only if the transaction commits, every sent record is durable and eventually visible to `read_committed` consumers in their respective partitions, and the committed offset for each partition is at least as high as the corresponding offset specified in the transaction. If the transaction does not commit, committed offsets do not advance, and some (or all) writes may (or may not) be visible depending on consumer configuration.

With the right implicit assumptions—for instance, that consumers process every record seen, that every record is processed in the scope of a transaction, that every transaction commits its highest consumed offsets, and so on—the committed offsets of a transaction can be understood as the set of records it consumed. With care, one can theoretically use transactions to obtain what Kafka terms "exactly-once semantics."

Consumers can run in one of two isolation levels: `read_uncommitted` or `read_committed`. At the default `read_uncommitted`, consumers may read values written by transactions which actually aborted. This is known as *aborted read (G1a)*. Transactions may observe none, part, or all of an aborted transaction's writes. Two or more read-write transactions may also observe each other's writes: a form of *circular information flow (G1c)*. Kafka's documentation

[3] Kafka's documentation varies in whether the committed offset is the offset of the last committed record, or the uncommitted offset immediately *after* the last committed record.

[4] One can also imagine that using a common format, like Apache Iceberg, would allow Bufstream data to be read directly from S3 by query engines like Spark, Trino, Google BigLake, or Amazon Redshift—without storing redundant copies of the data.

says that `read_committed` prevents G1a, and sort of[5] guarantees[6] that either all or none of a transaction's writes are eventually visible. In our tests of Kafka, Redpanda, and Bufstream, `read_commmitted` also prevented G1c involving only write-read dependencies.

On the other hand, our tests of Kafka, Redpanda, and Bufstream all found write cycles analogous to phenomenon G0 in normal operation. One transaction's writes can appear in the middle of a second transaction's writes. The major formalisms for Read Uncommitted generally prohibit G0, and the Kafka wiki claims it should not occur at `read_committed`:

> Since X2 is committed first, each partition will expose messages from X2 before X1.

Nevertheless, all three systems exhibited G0 both at `read_uncommitted` and `read_committed`. Either the documentation is wrong or Kafka's transaction isolation is broken. We identified this problem in our 2022 Redpanda analysis, but it remains unaddressed.[7]

Moreover, our tests show that Kafka, Redpanda, and Bufstream, even at `read_committed`, allow a different form of G1c proscribed by Read Committed. Specifically, they allow cycles of transactions linked by at least one write-write dependency. For instance, transaction T_1 can write a to topic-partition P before T_2 writes b to P. Then T_1 can read b, forming a cycle. Kafka's documentation still declines to state whether this should be legal.

Finally, a word about producer identifiers. Users can provide a *client ID*, which is helpful for logging but has no semantic effects. Producers also have an internal *producer ID*, which is used to de-duplicate sent records. Third, producers have a user-specified *transactional ID*,[8] which identifies a logical producer across multiple client instances. When a logical producer crashes and restarts, it provides its transactional ID to the server, which increments an *epoch* associated with that ID. Transactional writes from older epochs are rejected.

3 Test Design

We tested Bufstream 0.1.0 through 0.1.3, including several release candidate builds. We based our Bufstream test harness on Jepsen's previous work on Redpanda and Kafka. We used the Jepsen testing library and the Java Kafka Client at version 3.8.0. Since Bufstream implements the Kafka API, we were able to reuse much of the Redpanda/Kafka test as a library.

We ran our tests on three to five Debian Bookworm nodes, both as LXC containers and EC2 VMs. We reserved one node for etcd, one for Minio (an S3-compatible object store), and the remainder for Bufstream agents. Producers, consumers, and admin clients were always initialized with a single node for `bootstrap_servers`, but we did not interfere with smart client discovery: clients could talk to any node freely. As with all smart clients, this may have reduced our chances to observe safety violations.

All consumers shared a single consumer group, and committed their offsets manually after each non-transactional poll operation. For transactional workloads we gave each producer a unique transactional ID and used `sendOffsetsToTransaction` for any transaction which performed a poll—including read-only transactions.

We applied several configuration changes to clients in order to achieve faster recovery during failures, and to ensure safety. Our consumers ran with significantly shorter timeouts (generally under 10 seconds), and with tunable `isolation_level`, `auto_offset_reset`, and `enable_auto_commit`. Producers also ran with shorter timeouts, and configurable `acks`, `enable_idempotence`, and `retries`. In general we tested with the safest possible settings: auto-commit false, acks `all`, retries 1,000, idempotence enabled, isolation level `read_committed`, `auto_offset_reset` of `earliest`, and automatic creation of topics on the server disabled. We tested both with and without transactions.

Like most Jepsen tests, we injected the usual suite of faults into Bufstream: process pauses (via `SIGSTOP`), crashes (via `SIGKILL`), clock skew (via `clock_settime`) and partitions (via `iptables`). Because Bufstream comprises three distinct subsystems, we designed new subsystem-aware DB automation, client, and fault injection tools for Jepsen. This allowed us to target faults to a particular subsystem, such as just crashing Bufstream nodes or pausing only the etcd coordinator. We mixed these faults together in shifting combinations over time, producing, say, 30 seconds of partitions, 30 seconds of no faults, then Bufstream crashes with storage pauses, and so on.

3.1 Queue

In prior work on Redpanda and Kafka we designed a queue workload which performed sophisticated safety analysis geared towards Kafka's data model. That workload is now part of the core Jepsen library, and we updated it for use in Bufstream.

In this workload each logical process launched a producer, a consumer, and an admin client. For concision, we defined a logical numeric *key* which uniquely identified a specific topic-partition. Topics were created

[5] Atomicity in Kafka is complicated. Consumers are allowed to seek, or can be automatically reassigned, to arbitrary offsets—even in the middle of a transaction. Consumers using `subscribe` may be reassigned in between polls, causing them to skip over records. Compaction can destroy part, but not all, of a transaction's records. We aren't sure whether a call to `poll` can stop short of the end of a transaction, and so on.

[6] As we show later in this report, neither Kafka nor Bufstream actually ensure these invariants, thanks to a flaw in Kafka's transaction protocol.

[7] Specifically, see Write Cycles, Transaction Isolation, and Exactly-Once Semantics.

[8] Not a transaction!

dynamically on first use. Keys were selected with exponential frequency: some keys accessed quite often, while others only infrequently. After sending a configurable number of records to a key, we abandoned it and moved on to a new one.

The queue workload performed three basic kinds of operations. The first, `crash`, simulated a client failure: it terminated the logical process, closing all three clients. Jepsen would then create a fresh process with new clients to take its place. The second class of operations, `subscribe` or `assign`, updated the set of topics or partitions the consumer received records from when calling `poll`: either assigning a specific set of keys (topic-partitions), or subscribing to the set of topics which covered the requested keys.

The third class included `txn`, `poll`, and `send` operations. Each contained a sequence of `poll` or `send` micro-operations. Those which performed only polls or sends were labeled `poll` or `send`, rather than `txn`, but their structure was otherwise identical. Each `send` micro-operation sent a single *value* (a unique integer) to a specific key, and returned an `[offset, value]` pair, based on the offset Bufstream returned. Each `poll` micro-operation called `consumer.poll` once, and returned a map of keys to sequences of `[offset, value]` pairs observed for that key. For example:

```
[[:poll {1 [[2 3] [4 5]]}]]
 [:send 6 [7 8]]
```

This transaction polled key `1` and received two records back: at offset `2`, value `3`; and at offset `4`, value `5`. Then it sent a single record `8` to key `6`, which was assigned offset `7`.

For non-transactional workloads we constrained every `send` or `poll` operation to contain exactly one micro-operation. For transactional workloads, we allowed multiple micro-operations and wrapped them all in a Kafka transaction.

To analyze histories of these operations we first constructed, for each key, a mapping of offsets to sets of values observed at that offset, either via `send` or `poll`. If we observed multiple values at a single offset, we called it an inconsistent offset. Since every value was unique within a key, we also expected to observe each value at most once. If we observed the same value at multiple offsets, we called that a duplicate error.

If our mapping between a key's values and offsets was bijective, we could construct a total order over values. This order might not cover all values: calls to `send` and `poll` might not have returned offsets. Nor could we necessarily tell which offset an indeterminate, unobserved `send` might have produced. Moreover, not every offset contained a value: like Kafka, Bufstream uses some log offsets to store transaction metadata. We therefore collapsed our sparse offset logs into a dense version order which mapped each value to a unique *index* 0, 1, 2, ….

From this version order we looked for several additional errors, which came in symmetric flavors. We checked subsequent pairs of send micro-operations, and subsequent pairs of polls as well, to see if the offsets for their values were strictly monotonic and did not skip over intermediate indices. We looked for non-monotonic or skipped offsets both within a single transaction and between successive transactions by the same process.

For aborted reads, we searched for any poll which returned a value sent by a failed operation. We verified that transactions never observed their own writes, even if they later committed: we called this phenomenon *pre-committed read*.

When Bufstream confirmed receipt of a record but that record was never observed, we called that record *lost* or *unseen*. Lost records were those where some poller observed a higher offset. Since consumers start at a known-consumed offset and proceed linearly, it should be impossible to poll index n unless some poller has already seen index $n-1$, and by induction, all lower indices.[9]

Typically, a small tail of the log has just been written but has not yet been polled. After the main body of the test, we ceased transactions, resolved all faults, waited for recovery, then began a *final reads* phase. Each process assigned its consumer to every topic-partition, rewound to offset 0, and polled until it reached the highest offset known to have been written to that partition. If our final read phase timed out, leaving some acknowledged records unobserved, we called those records *unseen*.

3.2 Abort

Following unexpected results from the queue workload, we designed an abort workload which aborted transactions and kept track of which offsets are polled. We limited each topic to a single partition, process, producer, and consumer. Each process would create a new topic, subscribe to it, then perform a series of transactions against that topic, each involving a single poll and a variable number of sends. Once one of those transactions had polled some records, the process shifted to intentionally aborting transactions. After some time, the process returned to committing transactions.

We examined the offsets returned by polls after an aborted transaction, and classified them into four categories. An *advance* meant that the next transaction began its polls at an offset higher than the last one polled in the previous, aborted transaction. A *rewind* meant that the next transaction started polling at the same offset the aborted transaction started at. A *rewind-further* meant that it started at some earlier offset. Any other behavior we called *other*.

4 Bufstream Results

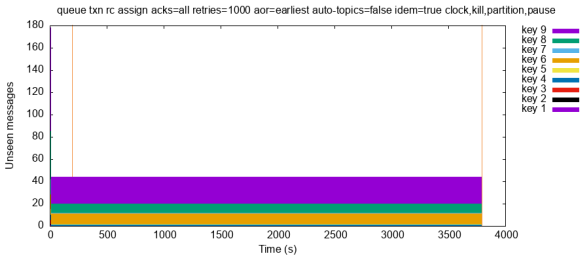
We begin with five issues in Bufstream proper, from liveness failures to duplicate offsets and data loss.

[9] We say index, rather than offset, because offsets are sparse.

4.1 Stuck Consumers (#1)

In 0.1.0 through 0.1.3-rc.8 our final read phase often stalled. Calls to `consumer.poll()` would return immediately with no records, even though thousands of acknowledged records remained in the log. This would continue for tens of seconds to over an hour, until either the test gave up waiting or Bufstream decided to deliver records to consumers again. This weakened our tests: those unseen records may have had safety issues, but without observing them we had no way to tell.

For instance, this test run sent 691 acknowledged records in the first 120 seconds, then shifted to final reads. At that time, 40 of those acknowledged writes had never been observed by any poller. This situation persisted for over an hour as calls to `consumer.poll()` repeatedly returned no results. Finally, the test timed out.



In other cases consumers would get stuck waiting for thousands of unseen records. Then after hundreds of seconds—for no apparent reason—Bufstream would rapidly produce the remaining values. We saw this behavior in all kinds of situations, including healthy clusters, but it was most pronounced with faults.

Bufstream made two patches in 0.1.3-rc.6 to help with this issue. First, when restarted, a Bufstream node could sometimes return stale, cached values for the last stable offset and high watermark.[10] This caused some client libraries to stall, assuming no later records were available (#1). Refreshing the cache on startup resolved this issue. We discuss the second patch (#3) later in this report, as it also had safety consequences.

4.2 Stuck Producers & Consumers (#2)

Unfortunately we continued to see regular issues with unseen writes on 0.1.3-rc.6, in response to pauses, crashes, or partitions affecting the coordinator, storage, or Bufstream nodes. In some cases, a coordinator pause could cause every Bufstream node to enter a state where the process was running, but clients would time out waiting for calls to `InitProducerId`. In other cases calls to `listOffsets` would fail with messages like `node 2008741112 being disconnected` or, alternatively because they `timed out waiting for a node assignment`. Calls to `poll` would complete but return no results. Killing and restarting Bufstream nodes resolved these issues. We call these metastable failures: a brief interruption in (e.g.) connectivity to the coordinator could cause long-lasting partial (or total!) unavailability in Bufstream agents.

Bufstream uses etcd leases to keep track of active Bufstream agents. Each agent subscribed, via its etcd client, to updates affecting a set of leased keys. Despite setting long timeouts, brief pauses or partitions caused etcd to delete keys tied to an agent's lease, but updates reflecting those deletions were not necessarily relayed by the client to the agent itself. In essence, an agent would be unaware that it had lost its lease. The Bufstream team added additional polling logic to work around this problem, and unseen writes were largely resolved by 0.1.3-rc.8.

4.3 Spurious Zero Offsets (#3)

In versions 0.1.0 through 0.1.3-rc.2, a sent value could be assigned offset `0` (even if offset `0` had already been assigned far earlier in the test), then appear at a higher, more reasonable offset. Only the sender observed offset zero; pollers always observed the higher offset. This occurred when either etcd or Bufstream paused, or crashed, or a network partition occurred between the two.

Consider this two-minute test run with a single Bufstream node, wherein we induced brief pauses in the etcd process. Six writes were assigned offset `0`, then appeared at a second, higher offset. On key `6`, value `26` was assigned offset `0` on send, then appeared at offset `25` in polls. On key `9`, `224` was assigned offset `0`, then appeared at `223`, and so on. Our checker reported these as duplicates:

```
:duplicate
{:count 6,
 :errs [{:key 6,
         :value 26,
         :count 2,
         :offsets [0 25]},
        {:key 9,
         :value 224,
         :count 2,
         :offsets [0 223]},
        {:key 9,
         :value 69,
         :count 2,
         :offsets [0 66]},
        ...]}
```

Key `9` actually had five values at offset `0`: value `1` (which was stable), `67`, `68`, `69`, and `224`. This caused our checker to report inconsistent offsets:

```
:inconsistent-offsets
{:count 3,
 :errs [{:key 6,
         :offset 0,
         :values #{1 26}},
        {:key 8,
         :offset 0,
         :values #{1 110}},
```

[10] The high watermark marks the prefix of the log known to be fully replicated and durable.

```
{:key 9,
 :offset 0,
 :values #{1 67 68 69 224}}]}
```

We could reproduce this behavior readily—it occurred roughly every five minutes. However, Bufstream initially struggled to reproduce it.

This issue was caused by Bufstream omitting a field from the error responses sent to clients. Imagine that Bufstream sent a message to etcd to commit a new record in the log, and etcd processed that request. However, due to a process pause or network partition, Bufstream might time out waiting for a response from etcd, and send an error back to the client. While Bufstream's response included an error code, it did *not* set the offset for the sent record to the special value `-1`, which some clients relied on as a signal of an error. Consequently, the official Java client we used in our tests interpreted this error as a successful response with offset `0`. Bufstream's test suite used Franz-go, which interpreted these messages as errors. This meant that Bufstream's test suite did not encounter this issue.

Bufstream fixed this issue in version 0.1.3-rc.6, and we have not observed it since.

4.4 Lost Transaction Writes (#4)

Version 0.1.2 exhibited frequent write loss. Records written as a part of a committed transaction could vanish, never to be seen again. Readers would simply skip over those records as if they had never existed. For example, consider this test run. In just 100 seconds and 6,761 write transactions, 240 records written by committed transactions were lost. Key `5`, for instance, had a successful write of value `141`:

```
{:type :ok,
 :process 7,
 :f :send,
 :value [[:send 11 [83 48]]
         [:send 5 [274 141]]]}
```

Since this transaction committed successfully, we know that key `5` should have stored value `141` at offset `274`. However, every call to `consumer.poll()` would skip over that offset, returning values like:

```
[[272 140]
 [273 142]
 ; Missing [274, 141]
 [277 144]
 [278 145]
 [282 146]
 ...]
```

This behavior happened regularly in healthy clusters, even with just a single Bufstream node. It was caused by a new concurrency safety mechanism added in 0.1.2 to mitigate the lack of idempotence in Kafka's transaction protocol. Bufstream nodes assigned a unique number to each transaction performed by a producer within a given epoch, allowing Bufstream to safely retry some internal operations. However, a bug in the logic for tracking transaction numbers caused some transaction commits to be erroneously ignored when multiple transactions were committed across multiple epochs. This caused transactions which appeared to commit to actually abort, or vice versa.

Bufstream's internal integration and unit tests initially missed this bug—we only caught it with the Jepsen test suite because of our choice of an unusually low (one second) transaction timeout. Luckily we identified the problem within a few hours of 0.1.2's release. Bufstream took action to prevent customers from upgrading to 0.1.2, and none did. The issue was fixed in 0.1.3-rc2.

4.5 Lost Writes Due to Server-Side Filtering (#5)

In version 0.1.3-rc.8, we regularly found short windows of write loss in response to minor faults, like pausing a Bufstream process or the coordinator, or a partition between the two. Data loss occurred both with and without transactions. Take this five-minute test run in which 22 out of 16,770 records were acknowledged, but never polled by any consumer. On key 12, value 663 was written at offset 662:

```
{:type :ok,
 :process 38,
 :f :send,
 :value [[:send 12 [662 663]]]}
```

However, every poller skipped over value 663 (and 664, which was also acknowledged). They read 665, then missed 666, 667, and so on:

```
{:type :ok,
 :process 143,
 :f :poll,
 :value [[:poll
          {...
           12 [... [660 661]
                   [661 662]
                   ; No 663
                   ; No 664
                   [664 665]
                   ; 6 missing records
                   [671 672]
                   ; 5 more missing
                   [677 678]
                   ...]}]]}
```

Sometimes records would be visible to pollers for a time, then missing from later polls. In this test run with process crashes, Bufstream acknowledged the writes of values `101` through `106` at roughly 1.37 seconds into the test. These records were visible to pollers until 1.88 seconds. After that, pollers simply skipped over the records as if they had never existed.

To work around a bug in a popular Kafka web GUI, Bufstream introduced logic in 0.1.3-rc.8 to more strictly limit the size of responses to the fetch API. However, a bug in that filtering logic caused Bufstream to hide records from some lagging consumers—which manifested as write loss. Bufstream fixed this issue in 0.1.3-rc.12.

5 Kafka Results

In the course of our research we uncovered several issues with the Kafka Java client, documentation, and protocol design. We present four of these issues here. These affect Kafka, Bufstream, and presumably any Kafka-compatible system.

5.1 A Misleading Error Message (KIP-588)

During our testing we encountered frequent errors like `ProducerFencedException: There is a newer producer with the same transactionalId which fences the current one.` This was particularly vexing in tests where every producer received a unique transactional ID. We spent a good deal of time verifying that producers were initialized at most once, that data from past runs was not leaking into the present, and so on. Finally, the Bufstream team identified KIP-588, which notes that a ProducerFencedException is *also* thrown for a transaction timeout:

> When the producer goes back online and attempts to proceed, it will receive the exact `ProducerFenced` even though a conflicting producer doesn't exist.

Kafka's Java client uses a dedicated `TimeoutException` for most timeouts, but throws `ProducerFencedException` for this particular kind of timeout instead. When this happens, the error message lies to the user, asserting a second instance of the producer exists when none actually does. KIP-588 has been open for two years; we recommend the Kafka team change this error message.

5.2 Closing a Consumer Can Block Indefinitely (KAFKA-17734)

Our tests against both Bufstream and Kafka got stuck every few hours thanks to a bug in the Java client. Calls to `Consumer.close()` block on network IO by default. There is a timeout parameter which is supposed to prevent calls to `close()` from blocking indefinitely, but it doesn't work. Neither does spawning a separate thread specifically to call `consumer.wakeup()`, which is intended to safely interrupt a consumer stuck in IO.

Long-running programs should be robust to network errors. This means they should be able to reliably tear down clients and their associated resources—connections, threads, allocated memory, and so on—in a reasonable amount of time. We filed KAFKA-17734 to track this issue.

5.3 Unpredictable Consumer Offsets After Transaction Failure (KAFKA-17582)

Kafka's official documentation is largely silent about the intended behavior for consumer offsets when a transaction fails to commit. Should the consumer rewind, such that the next transaction's polls begin with the first records observed by the aborted transaction? Or should it continue advancing, polling subsequent records? The original transaction proposal, KIP-98, says:

> Further, since consumer progress is recorded as a write to the offsets topic, the above capability is leveraged to enable applications to batch consumed and produced messages into a single atomic unit, ie. a set of messages may be considered consumed only if the entire 'consume-transform-produce' executed in its entirety.

Confluent's Kafka design documentation goes on to say:

> If the transaction is aborted, the consumer's position reverts to its old value and you can specify whether output topics are visible to other consumers using the `isolation_level` property.

It makes sense that consumers should rewind on abort. After all, aborting a transaction in any system generally undoes its effects. More critically, Kafka users typically want at-least-once delivery—advancing to later offsets could mark records from the aborted transaction as committed even though they had never been processed. In fact the official Java client does rewind, but only *sometimes*.

We first encountered this behavior in the queue workload, where it manifested as write loss during a variety of faults, both with Bufstream and Kafka. We designed the abort workload to follow up and found that even in healthy clusters, aborts led to unpredictable outcomes. For instance, here are results from a five-minute abort test with no fault injection. Most pairs of transactions advanced to later offsets, but some rewound to earlier ones. Some rewound to the start of the transaction, and others rewound even further. All rewinds were associated with a rebalance event, and all advances had no rebalances.

```
{[:advance :none]             17048,
 [:rewind-further :rebalance] 1745,
 [:rewind :rebalance]         476}
```

For example, process 0, interacting with topic-partition `t374`, aborted a transaction which polled records 12 through 14. Then it went on to poll and commit later records. Offsets 12 through 17 were effectively lost.[11]

```
{:process 0,
 :type    :fail,
 :f       :poll,
 :value   {:topic   "t374",
           :offsets [12 14 15 17],
           :abort?  true}}
{:process 0,
 :type    :ok,
 :f       :poll,
```

[11] Throughout this work we omit some fields (e.g. timestamps, operation indices, etc.) from records for concision and clarity.

```
 :value   {:topic   "t374",
           :offsets [18 19 21 23]}}
```

On the other hand, if a rebalance occurred consumers could rewind to earlier offsets, preventing data loss. For example, process 15, consuming from topic-partition `t1208`, was reassigned to that same topic. That rebalance event reset the consumer's position from offset 5 to offset 0, causing it to rewind further than the most recent aborted transaction:

```
{:process 15,
 :type    :fail,
 :f       :poll,
 :value   {:topic "t1208",
           :abort? true,
           :offsets [4]}}
{:process 15,
 :type    :ok,
 :f       :poll,
 :value   {:topic "t1208",
           :abort? true,
           :offsets [0 1 2 4]},
 :rebalance-log
 {:during
   [{:type :assigned,
     :partitions [{:topic "t1208",
                   :partition 0}]}],
   :before []}}
```

We opened KAFKA-17582 to ask for clarification, and learned that this behavior is intentional. Consumers continue advancing, unless they happen to be rebalanced, in which case they might rewind to an arbitrary point—whatever happens to be committed. Users are supposed to manually rewind the consumer's position on transaction abort. Indeed, one of Kafka's demonstration programs includes a rewind method for exactly this reason.

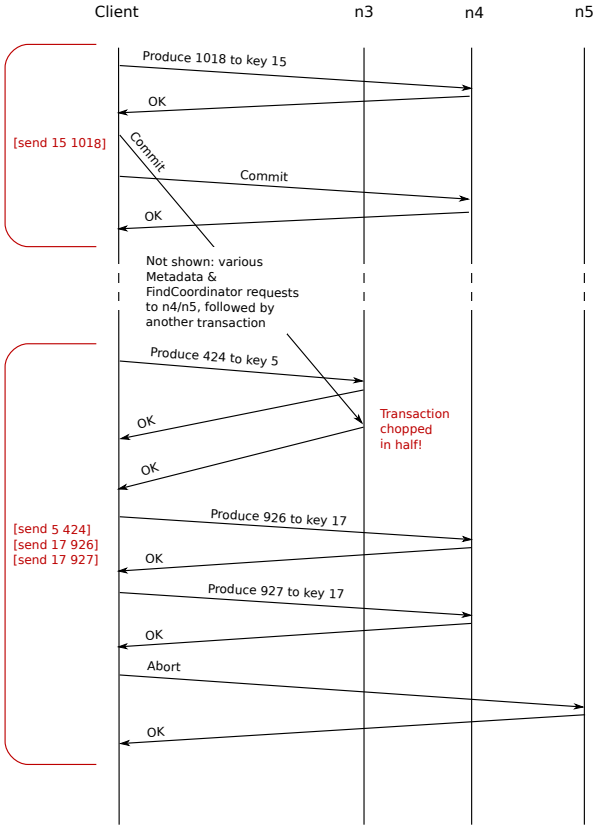
This directly contradicts Confluent's documentation. It also runs contrary to KIP-98's statement that "a set of messages may be considered consumed only if the entire 'consume-transform-produce' executed in its entirety." KIP-98's example code contains no rewind, and we could not locate any documentation guiding users to rewind by hand. We suggested that Kafka document this behavior, and consider changing consumers to rewind by default on transaction abort. We also updated our queue workload to explicitly rewind consumers.

5.4 Write Loss, Aborted Reads, Torn Transactions (KAFKA-17754)

In Bufstream 0.1.0 through 0.1.3 we observed aborted read, lost writes, and atomicity violations with as little as pausing or crashing the Bufstream process, or the coordinator, or a network partition. These behaviors led us to a fundamental flaw in the Kafka transaction protocol. For example, take this queue test of version 0.1.3-rc.9. The test harness executed the following transaction, but aborted it intentionally:

```
{:type    :fail,
 :process 76,
 :f       :txn,
 :value   [[:send 5 [653 424]]
           [:send 17 [1360 926]]
           [:poll {}]
           [:send 17 [1382 927]]],
 :error
 {:type          :abort,
  :abort-ok?     true,
  :tried-commit? false,
  :definite?     true,
  :body-error    {:type :intentional-abort}}}
```

Process 76 selected the unique transactional ID `jt1234` on initialization. From packet captures and Bufstream debug logs, we see `jt1234` used producer ID 233, submitted all four operations, then sent an `EndTxn` request with `committed = false`, which denotes a transaction abort. However, fifteen separate calls to `poll()` observed this transaction's write of 424 to key 5—a clear case of aborted read. Even stranger, *no* poller observed the other writes from this transaction: key 17 apparently never received values 926 or 927. Why?



Close inspection of the packet capture, combined with Bufstream's logs, allowed us to reconstruct what happened. A few transactions prior, process 76 began a transaction which sent `1018` to key `15`. It sent an `EndTxn` message to commit that transaction to node `n3`. However, it did not receive a prompt response. The client then quietly sent a *second* commit message to `n4`, which returned OK, and the test harness's call to

`commitTransaction` completed successfully. The process then began and intentionally aborted a second transaction, which completed OK. So far, so good.

Then process 76 began a third, problematic transaction. It sent `424` to key `5` and added new partitions to the transaction. Just after accepting record `424`, node `n3` received the delayed commit message from two transactions ago. This committed the current transaction, effectively tearing it in half. The first half (sending `424` to key `5`) was committed and visible to pollers. The second half (sending `926` and `927` to key `17`) implicitly began a second transaction, which was then aborted by the client.

This suggests a fundamental problem in the Kafka transaction protocol. The protocol is designed to allow clients to submit requests over multiple TCP connections and to distribute them across multiple nodes. There is no sequence number to order requests from the same client. There is no concept of a transaction number.[12] When a server receives a commit (or abort) message, it has no way to know what transaction the client intended to commit. It simply commits (or aborts) whatever transaction happens to be in progress.

This means transactions which appeared to commit could actually abort, and vice versa: we observed both aborted reads and lost writes. It also means transactions could be cut in half: a single transaction could have some of its writes lost, and others preserved. We don't know a name for this anomaly. It's clearly a violation of atomicity, but "atomic" is a somewhat vague term. If a reader observed some but not all of a different transaction's writes we would call it a *fractured read*, but this anomaly occurs on the write path: no (`read_committed`) poller will ever observe the lost writes. We call this behavior a *torn transaction*.[13]

What does it take to get this behavior? First, an `EndTxn` message must be delayed, for instance due to network latency, packet loss, a slow computer, garbage collection, etc. Second, while that `EndTxn` arrow is hovering in the air, the client needs to move on to perform a second transaction using the same producer ID and epoch. There are several ways this could happen.

First, users could explicitly retry committing or aborting a transaction. The docs say they can, and the client won't stop them. Second, the official Kafka Java client docs repeatedly instruct users to call `abortTransaction` if an error occurs during `commitTransaction`.[14] Following the documentation's example leads directly to this behavior: if `commitTransaction` times out, one calls `abortTransaction`, and there are now multiple `EndTxn` messages in flight. Third, even if users try to avoid this by only calling `commitTransaction` *or* `abortTransaction`, the client's internal retry mechanism treats timeouts as retryable and sends multiple `EndTxn` messages automatically. In the above example, process 76 called commit or abort exactly once for every transaction it ever performed, and it still hit data loss.

We observed aborted reads and torn transactions due to process pauses in Kafka as well, and opened KAFKA-17754 to track the issue. Kafka's engineers believe KIP-890, which was motivated by hanging transactions in Kafka, will likely fix the problem. In brief, KIP-890 revises the transaction protocol to bump the producer's epoch on every transaction. Since servers reject messages from older epochs, this should prevent commit messages from prior transactions leaking into later ones. KIP-890 was opened in November 2022, and work is ongoing.

Bufstream has added a mechanism in 0.1.3 to reduce the frequency of these issues. Bufstream nodes now use the most recently observed etcd revision—a global, monotonically increasing integer coupled to the system state as a whole—as a logical clock on `AddPartitionsToTxn` and `EndTxn` messages. If an `EndTxn` message attempts to commit or abort a transaction with a newer `AddPartitionsToTxn`, Bufstream returns a non-retryable error. Of course this order is only known once RPC messages arrive on Bufstream nodes; it does not prevent reorderings that occur between the client and Bufstream itself. Indeed, we continue to observe aborted read, lost writes, and torn transactions in 0.1.3. We must wait for clients to resolve this issue.

№	Summary	Event Required	Fixed in
1	Stuck consumers due to lagging highest stable offset	None	0.1.3-rc.6
2	Stuck producers/consumers due to etcd lease expiry	Pause	0.1.3-rc.8
3	Spurious zero offsets	Pause	0.1.3-rc.6
4	Lost transaction writes	None	0.1.3-rc.2
5	Lost writes due to server-side filtering	Pause	0.1.3-rc.12
KIP-588	Wrong error message on transaction timeout	None	Unresolved
KAFKA-17734	`ConsumerClient.close()` can block indefinitely	Pause	Unresolved
KAFKA-17582	Unpredictable consumer offsets after transaction failure	None	Unresolved
KAFKA-17754	Write loss, aborted read, torn transactions	Pause	Unresolved

[12] The Kafka protocol documentation says that `EndTxn`, the message which commits or aborts a transaction, has a `transactional_id` field which specifies "The ID of the transaction to end." As previously mentioned, a transactional ID identifies a set of producers, not a transaction.

[13] Cut my writes into pieces / This is commit-abort.

[14] The only exceptions in the docs are `ProducerFencedException`, `OutOfOrderSequenceException`, and `AuthorizationException`, none of which apply here.

6 Discussion

We found two liveness and three safety issues in Bufstream proper. Two issues (#1 and #2) involved consumers or producers getting stuck, sometimes indefinitely. One (#3) involved Bufstream returning `0` rather than `-1` for a sent record which failed indefinitely. The official Java client interpreted this response as a success, rather than an error. Two issues allowed the loss of committed writes due to a bug in a concurrency control mechanism (#4) and a bug in a filter to limit response sizes (#5). We identified both #4 and #5 before production users were affected. As of version 0.1.3, all five issues are resolved.

However, Bufstream continues to exhibit aborted reads, lost writes, and torn transactions due to the design of the Kafka transaction protocol (KAFKA-17754). These issues cannot be resolved without the help of the Kafka team and client implementers. We also note three other Kafka issues, including an incorrect error message (KIP-588), a deadlock in `ConsumerClient.close()` (KAFKA-17734), and the lack of any authoritative documentation for transaction semantics (KAFKA-17671).

As always, we caution that Jepsen takes an experimental approach to safety verification: we can prove the presence of bugs, but not their absence. While we make extensive efforts to find problems, we cannot prove correctness. In particular, KAFKA-17754 makes it difficult to determine if there are *other* cases of (e.g.) write loss in Bufstream.

6.1 Bufstream Recommendations

Bufstream users who use the official Java Kafka client should be aware that transactions are, at present, unsafe. Aborted transactions could actually commit, committed transactions could actually abort, and transactions could be torn in half, preserving some but not all of their effects. Bufstream believes the Franz-go client is less susceptible to this problem, but we have not tested it using the same techniques as the present work. Other clients may or may not be susceptible. While Bufstream is trying to reduce the frequency of these issues, they cannot prevent them entirely. That power lies with the Kafka team and client implementers.

Bufstream users prior to 0.1.3 should be aware that calls to `producer.send()` might incorrectly return a zero offset for a record, rather than the actual offset. They may also encounter metastable availability issues where clients get "stuck." We recommend upgrading to 0.1.3.

Bufstream's overall architecture appears sound: relying on a coordination service like etcd to establish the order of immutable chunks of data is a relatively straightforward approach with years of prior art in both OLTP and streaming systems. Kafka's attention to KIP-588, additional deployment experience, and further testing should help identify and resolve any remaining safety bugs. In the meantime, we made two small operational recommendations for Bufstream.

First, we suggested Bufstream retry a network operation that often caused clusters to crash. Bufstream requests a shared file in storage on startup, as a safety check designed to prevent users from accidentally running two different Bufstream clusters on top of the same storage bucket. Bufstream exits if it can't complete this request. This caused Bufstream to crash roughly one in thirty tests, due to `404 not found` responses from storage. Bufstream has added a layer of retries, and intends to further ameliorate this issue by retrying indefinitely while refusing client connections. Version 0.1.3 appears significantly more robust on startup.

Second, Jepsen suggested that Bufstream processes try to keep running when their dependencies are unavailable. Presently, agents kill themselves when they lose access to storage or the coordinator, and rely on a supervisor system like Kubernetes to restart them. This works, but it requires operators to run their own supervisor. It may also impact performance and availability. Clients must tear down and re-open TCP connections, which could create thundering-herd issues. Bufstream processes may also need to do special work on restarting: fetching resources from storage, warming caches, re-connecting to etcd, and so on. Generally speaking, services should try to keep running when their dependencies become unavailable—the service can offer backpressure, advise clients of system status, and generally recover more gracefully. Bufstream has added additional retry logic for etcd, but as of 0.1.3, still requires constant supervision to stay online. We also recommend users verify that they have a process supervisor in place, and test that it works correctly (rather than giving up) during prolonged outages.

6.2 Kafka Needs Transaction Docs

Kafka's official documentation says almost nothing about transactions, leaving users to piece together behavior from a maze of vague, confusing, and contradictory sources. We encouraged the Kafka team to write an official, centralized, easy-to-find document which lays out the intended semantics of transactions: KAFKA-17671. This document should specify exactly what users must do to use transactions safely, and what invariants they can expect in return. For instance, it should describe the rules for offsets visible to a single producer or consumer:

- When will a consumer observe monotonically increasing offsets?
- When will a consumer skip over acknowledged records?
- Is this behavior different within a single call to `poll`, versus between two subsequent calls?
- Can a rebalance take effect in the middle of a transaction? How do rebalances affect transaction semantics?

- When will records sent by a producer to a single topic-partition have monotonically increasing offsets?
- When will those offsets be interleaved with writes from other producers?
- For both consumers and producers, how do these behaviors differ within a transaction vs. between two subsequent transactions on the same client?

This document should also explain the isolation properties of transactions:

- When is G0 (write cycle) legal?
- When is G1a (aborted read) legal?
- When is G1b (intermediate read) legal?
- When is G1c (circular information flow) legal?
- What particular cycles are prohibited, if any? For example, are cycles composed entirely of write-read edges proscribed?
- When is fractured read legal? That is, when can a transaction observe some, but not all, of another transaction's effects?
- Is it legal to read values written inside the current transaction?

It should describe the semantics of aborted transactions:

- How are explicitly aborted transactions different from those which (e.g.) crash before committing?
- Are the values and offsets returned by `poll()` correct even if the transaction aborts?
- What offsets should a consumer poll after a transaction crashes?
- How should users navigate Kafka's maze of transaction errors?
- How should users handle errors that occur during the transaction abort process, or during rewind?

… and clarify other ambiguities in the existing documentation:

- Is the "committed position" the offset of the highest committed record, or the uncommitted offset one higher?
- Is it safe to let multiple transactional IDs process records from the same topic-partitions?
- Which offsets can auto-commit commit? When is data loss possible?
- How does auto-commit interact with transactions?

We also recommend that Confluent align their safety claims with Kafka's behavior. Confluent repeatedly claims that Kafka offers at-least-once delivery by default. This appears untrue: `auto.offset.reset = latest` allow records to be marked "committed" despite never being processed, and Confluent's offset management docs say that with the default auto-commit behavior, "there's a risk of losing the message's progress if the consumer crashes". Those same docs incorrectly claim that consumers rewind offsets on transaction abort. They do not; this too could lead to data loss. Either the default behaviors should be changed, or the documentation updated.

6.3 Kafka Transactions are Broken

The Kafka transaction protocol is fundamentally broken and must be revised. As we showed in KAFKA-17754, anyone who uses the official Java Kafka client with Bufstream, Kafka, or presumably any Kafka-compatible system may observe aborted reads, lost writes, and torn transactions. These break the most basic safety guarantees Kafka transactions are supposed to provide.

The crux of the problem is that Kafka's transaction system implicitly assumes ordered, reliable delivery where none exists. Processes pause, networks are not reliable, latency is non-zero, and delivery across different TCP sockets is fundamentally unordered. Kafka's protocol distributes messages across different nodes and TCP sockets by design. Clients automatically retry messages, leading to duplicates. The protocol includes neither a sequence number to reconstruct the order of messages sent by a single client,[15] nor a transaction number to ensure messages affect the right transaction.

On top of this unreliable foundation, the Kafka transaction protocol is an ordered state machine. `Produce` and `EndTxn` messages add records to, commit, or abort, whatever transaction happens to be ongoing at the time. Producer epochs provide a logical clock, but nothing ensures order *within* an epoch, and epochs are incremented infrequently. This demonstrates the importance of the end-to-end principle in protocol design: the client and transaction state machine must explicitly encode and enforce ordering "at the edges," rather than relying on the unreliable network between them.

KIP-890 intends to ensure a stricter order by incrementing the epoch on every transaction commit. Client libraries could also help by re-initializing producers (which bumps the epoch) when a message is not acknowledged.

We know the official Java Kafka client is vulnerable to this problem as of version 3.8.0. We believe Franz-go does re-initialize on timeouts, which could mitigate or prevent these issues. We haven't investigated other client libraries.

6.4 Future Work

Many users rely on the Kafka Streams API for "exactly-once semantics," rather than performing transactions themselves. Future work could explore the correctness of Streams applications.

While investigating issues like KAFKA-17754, we also encountered unseen writes in Kafka. Owing to time constraints we have not investigated this behavior,

[15] Kafka messages do include a unique *correlation id*, but this ID is not used for ordering. There are also sequence numbers on `Produce` messages, but they do not extend to (e.g.) `EndTxn`.

but unseen writes could be a sign of hanging transactions, stuck consumers, or even data loss. We are curious whether a delayed `Produce` message could slide into a future transaction, violating transactional guarantees. We also suspect that the Kafka Java Client may reuse a sequence number when a request times out, causing writes to be acknowledged but silently discarded. More Kafka testing is warranted.

When a rebalance event occurs, the positions of consumers may shift forward or back. It is unclear what the rules are for these rebalances. Our test suite detected various cases of consumers or producers encountering internal (or external) non-monotonic (or skipped) sends (or polls). However, we are unsure when these behaviors are legal, and our checker does not report them as outright failures. Once Kafka documents intended behavior, we would like to verify it.

Jepsen is a random process. The generators of operations, thread scheduler, network and disk IO, services under test, and operating systems involved are all nondeterministic. We rely on anomalies being probable enough that they occur regularly across different, random test runs. This is an effective way to identify and resolve bugs that are likely to occur in real-world systems. However, it is a terrible way to explore rare behaviors: if an anomaly occurs once in a hundred hours of testing, debugging and reproducing it becomes arduous.

For example, we encountered a single case of *non-zero* duplicate offsets on 0.1.3-rc.2. In this two-minute test run with producer pauses, a process sent record `743` to key `9`, was assigned offset `1225`, and was able to poll the record at that offset. Other pollers begged to differ—every other process observed value `743` at offset `1219`. We were unable to reproduce this behavior after weeks of testing, and it could have been an error in our test harness. Lacking confidence, we opted not to include this in our findings. A reproducible test would have made discharging this issue significantly easier.

Bufstream also uses Antithesis, a testing platform which runs an entire distributed system in a deterministic hypervisor and simulated network. This allows perfectly reproducible tests, and also lets testers rewind time to inspect the state of a system just before and after a bug occurred. We would like to combine Jepsen's workload generation and history checking with Antithesis' deterministic and replayable environment to make our tests more reproducible.

This work would not have been possible without the help of the Buf team, including Jacob Butcher, Mary Cutrali, Peter Edge, Rubens Farias, Alfred Fuller, Artūras Lapienė, Connor Mahony, David Marby, Chris Pine, Derek Perez, Luke Rewega, Chris Roche, Akshay Shah, Nick Snyder, and Philip Warren. Our thanks also to Artem Livshits and Justine Olshan for their support in investigating Kafka behavior. Our sincere appreciation to Irene Kannyo for her editorial support. This report was funded by Buf Technologies, Inc. and conducted in accordance with the Jepsen ethics policy.