JEPSEN

Redpanda 21.10.1

Kyle Kingsbury

2022-04-29

Redpanda is a distributed streaming system compatible with the Kafka wire protocol. We tested Redpanda versions 21.10.1 through 21.11.2, as well as development builds through January 30, 2022. We found three liveness and seven safety issues, ranging from crashes and aborted reads to inconsistent offsets, circular information flow, and lost/stale messages. We also discuss some potentially surprising behaviors, including an ambiguously named error and confusing documentation around write isolation in Kafka's transaction model. Redpanda has resolved seven of these issues, though some fixes aren't yet released. One crash and an issue involving lost/stale messages remain under investigation, and three more issues require only documentation. This work was funded by Redpanda Data, and conducted in accordance with the Jepsen ethics policy.

1 Background

Redpanda is a Kafka-compatible distributed streaming system based around append-only logs. Compared to Kafka, Redpanda aims to offer users lower latencies and reduced operational complexity. It uses the Raft consensus algorithm internally, rather than depending on a separate installation of Zookeeper. Redpanda speaks Kafka's wire protocol: rather than ship their own client libraries, Redpanda uses regular Kafka clients.

Like Kafka, Redpanda provides a set of named, partially ordered logs called *topics*. Each topic is sharded into one or more *partitions*,[1] each of which is a totally ordered log of messages. A message's position within a partition is identified by a unique monotonically increasing integer *offset*, which provides that total order. Offsets may be sparse: some offsets in the log are for internal messages, like transaction metadata, which are invisible to clients.

Kafka clients are split into several parts. Users write (*produce*) messages to partitions using a *producer* client, and read (*poll*) messages from partitions using a *consumer* client. Users may produce and consume to and from manually specified partitions, or allow the system to automatically select which partition a message is written to, and which partition(s) a consumer reads from.

Polling messages does not delete them. Instead, consumers emulate "consuming from a queue" by reading successive offsets from a given partition. Consumers can either *assign* themselves specific partitions and manage offsets themselves, or *subscribe* to a topic and allow Redpanda to automatically manage partitions and offsets. Offsets can be *committed* to Redpanda, which stores them durably so that crashed consumers can pick up where they (or their forebears) left off.

1.1 Safety

As of December 14th, 2021, Redpanda's home page compared Kafka to Redpanda. Where Redpanda offered "zero data loss by default", Kafka was marked "Caveat: reduces performance". Redpanda engineers explained that Kafka's default configuration may acknowledge writes before `fsync`. This might allow Kafka to lose messages when nodes fail. Redpanda nodes, by contrast, only acknowledges writes once they are `fsync`ed on a majority of replicas. Redpanda also advertised its use of the Raft consensus algorithm for safety. Other than these claims, Redpanda's documentation was relatively quiet on questions of fault-tolerance, durability, consistency, and other safety guarantees.

Like Kafka, Redpanda is intended for a variety of streaming applications. Some of these are safety-critical: messages must never be lost. Others focus on throughput or latency: skipping some messages is fine. Both Redpanda and Kafka clients expose a number of configuration settings which trade off between speed and safety. Users should be aware of these settings, and choose appropriately for their workload.

First, the producer `acks` setting controls how many nodes must acknowledge a producer's write before it is considered committed. The strongest setting is `acks = all` (also written `-1`). In Kafka this waits for all nodes in the in-sync replica set (ISR) to acknowledge the write, but does not fsync by default. In Redpanda this waits for a majority of nodes to fsync. A value of `0` allows Redpanda to confirm a message without writing it to *any* node; a single-node crash could cause data loss. Choosing `1` waits for a single node to acknowledge the message, which allows data loss in the event that node fails. The default, as of Kafka's 3.0.0 client, is `acks = all`.

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

Kafka producers can automatically retry writes, which means they may append a single message multiple times to a topic. To prevent this, clients may either set `enable.idempotence = true` or `retries = 0`. The Kafka Java client enables idempotence by default.[2]

Consumers can automatically commit their current offsets to Kafka. Since consumers advance their local offsets as soon as they see a message, this can cause messages to be considered committed even if they haven't been processed yet. If a consumer crashes after auto-commit but before processing that message, it might never be tried again. To avoid this, consumers should set `enable.auto.commit = false`. The default is true, which could effectively lead to message loss.

When a consumer begins reading from a partition, and no offset has been committed to Redpanda, that consumer has to choose an offset to start at. This behavior is controlled by `auto.offset.reset`. The default value, `latest`, starts clients near the most recent offset in the log. If clients crash without committing an offset, this could allow them to skip over some messages. To ensure clients read all messages in this scenario, use `earliest`, which rewinds fresh clients to the earliest log offset available. Of course, log expiration policies could also cause consumers to miss messages.

1.2 Transactions

Redpanda implements Kafka's transaction protocol, but this support remained behind a feature flag in version 22.1.1. Redpanda offers users no specific documentation on how to use transactions, relying on Kafka's documentation instead. Kafka's official documentation specifies that consumers can choose between two isolation levels: `read_uncomitted` and `read_committed`. `read_uncommitted` allows consumers to see "all messages, even transactional messages which have been aborted." The `read_committed` setting "will only return transactional messages which have been committed."

Readers familiar with other databases may know that these terms have existing meanings: they have been studied and formalized since at least the mid-1990s. In Adya, Liskov, & O'Neil's formalism, read uncommitted prevents phenomenon G0 (*write cycle*). Write cycle occurs when two (or more) transactions' writes interleave on one or more objects. For example, transaction T_1 writes some object x before T_2 writes x, and T_2 writes some y before T_1 writes y, given some total order of writes to x and y. Read committed proscribes three additional phenomena: G1a (*aborted read*), G1b (*intermediate read*), and G1c (*circular information flow*). Aborted read means a transaction observes a value written by a transaction which did not commit. Intermediate read involves reading a state from the middle of a transaction. Circular information flow encompasses dependency cycles between transactions where either T_1 writes some x before T_2 writes x, or T_2 reads something T_1 wrote. Of course, these anomalies pertain to histories of reads and writes over registers, not logs, but one can imagine analogous phenomena in Kafka's data model.

The `isolation_level` documentation makes it seem clear that `read_committed` proscribes G1a. What of G0, G1b, and G1c? Kafka's official documentation is somewhat vague[3] on this point, but it *does* argue two key properties. First, transactions (when combined with idempotence) ought to ensure what Kafka calls *exactly-once semantics*: one can consume messages from some topics, and send new messages to other topics, such that the new messages resulting from those particular input messages are produced only once. Second, transactions provide *atomicity*:

> Transactional delivery allows producers to send data to multiple partitions such that either all messages are successfully delivered, or none of them are.

While the Kafka documentation quoted above does not appear to discuss this, a 2017 Confluent blog post provides some caveats around atomicity: consumers may not be subscribed to all partitions involved in a transaction, so they may see only some, not all, of its effects. Another Confluent post describes Kafka's atomicity as *eventual*:

> … either all messages in the batch are eventually visible to any consumer or none are ever visible to consumers.

Jepsen is unsure whether "eventual" here is meant to imply a lack of real-time or session guarantees (e.g. transactions appear to execute in partial/total order, but there may be some time between transaction commit and visibility), or if eventual implies a lack of isolation: e.g. reads and/or writes from multiple transactions may be interleaved together. Redpanda believes both senses are true.

Confluence's Transactional Messaging wiki page provides five simple requirements that transactional applications in Kafka expect. At first, this looks like a promising summary for users trying to understand transaction semantics:

1. Atomicity: A consumer's application should not be exposed to messages from uncommitted transactions.
2. Durability: The broker cannot lose any committed transactions.
3. Ordering: A transaction-aware consumer should see transactions in the

[2] At least, it *claims* to. As we'll see, the story is slightly more complicated.

[3] Documentation and blog posts about Kafka transactions tend to explain behavior in terms of *implementation*, rather than *invariants*. Users who want to know: "can two transactions' writes interleave?" must infer the answers from descriptions of complex Kafka internals: producer transactional IDs, sequence numbers, epochs, zombie fencing, high watermarks, last stable offsets, etc. Terms like "atomic" are deployed ambiguously. Two documents will cross-reference one another while making contradictory claims. While Jepsen has read carefully and engaged Redpanda's support in writing this analysis, we frequently read between the lines to guess at what kind of invariants users might expect from Kafka & Redpanda. Any errors are, of course, Jepsen's alone.

original transaction-order within each partition.
4. Interleaving: Each partition should be able to accept messages from both transactional and non-transactional producers
5. There should be no duplicate messages within transactions.

On closer reading, this is even more confusing. “Atomicity” here means neither all-or-nothing commit nor the appearance of isolated point-in-time evaluation. Instead, atomicity is defined as the prevention of aborted read. The ordering property appears ambiguous: “transaction order” could mean an order over transactions, or an order over operations *within* each transaction. The example which follows suggests they mean both: given concurrent transactions X1 and X2…

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

From this Jepsen concludes “transaction order” means (at least) the order of commits. Although writes from X1 and X2 were interleaved in real time, consumers should process all messages from X2 strictly before all messages from X1. This suggests that G0 (write cycle) ought to be prohibited.

Redpanda believes this wiki page is wrong on both points. They point to a somewhat difficult-to-find Google Doc which serves as the design document for Kafka’s transactional protocol, which states that write atomicity refers to writes succeeding or failing as a unit, and then links to the aforementioned wiki page which says atomicity means the prevention of aborted read. The design document also provides several scenarios in which atomicity would fail to hold:

1. For compacted topics, some messages of a transaction maybe overwritten by newer versions.
2. Transactions may straddle log segments. Hence when old segments are deleted, we may lose some messages in the first part of a transaction.
3. Consumers may seek to arbitrary points within a transaction, hence missing some of the initial messages.
4. Consumer may not consume from all the partitions which participated in a transaction. Hence they will never be able to read all the messages that comprised the transaction.

These are reasonable constraints given Kafka’s data model. But what happens if we don’t use compaction, don’t seek to arbitrary offsets, and do consume from all partitions which participate in a transaction? Are writes isolated from one another?

To be specific: if transaction T_1 commits before T_2, do all offsets written by T_1 fall before those written by T_2? Many transaction systems buffer their writes and apply them more or less atomically at commit time, but a careful reading of this design document suggests that Kafka does not do this. Instead, Kafka chooses to add writes to the log immediately as each request in a transaction occurs—and to preserve performance, does not lock partitions for the duration of a transaction’s writes. This means that writes from two different transactions may interleave in offsets.

What about the order in which consumers process those writes? The Consumer section of the design document explains that messages are always delivered in offset order. This suggests that the wiki is incorrect, and transactional writes should visibly interleave.

Indeed, a subsection titled “Discussion on Transaction Ordering” explains that Kafka considered having consumers buffer and re-order these writes so that they *were* processed in transaction order. This approach would provide some additional measure of transactional isolation and improved latency. However, Kafka users expect that messages are delivered in offset order, and many parts of the Kafka API use a single high water mark to indicate which offsets have been processed. Delivering messages in transaction order would force consumers to track each offset individually, at least for a window of concurrent transactions. Perhaps the Confluence Wiki reflects earlier goals for the transaction system, and was never updated as the design evolved.

From all of these sources, a sufficiently diligent reader could conclude that Kafka (and therefore Redpanda) transactions allow G0, prohibit G1a at `read_committed`, and allow G1b. Whether G1c (cycles involving both write-write and write-read dependencies) may occur remains unclear.

1.3 Transactional IDs

Each producer performing a transaction in Kafka/Redpanda must choose a *transactional ID*: a string whose meaning is poorly defined but which, if chosen incorrectly, may cause the transaction system to exhibit undefined behavior. We must therefore discuss it in detail.

The official Kafka documentation describes the `transactional.id` producer setting like so:

> The TransactionalId to use for transactional delivery. This enables reliability semantics which span multiple producer sessions since it allows the client to guarantee that transactions using the same TransactionalId have been completed prior to starting any new transactions. If no TransactionalId is provided, then the producer is limited to idempotent delivery. If a TransactionalId is configured, `enable.idempotence` is implied. By default the TransactionId is not configured, which means transactions cannot be used. Note that, by default, transactions require a cluster of at least three brokers which is the recommended setting for production; for development you can

> change this, by adjusting broker setting `transaction.state.log.replication.factor`.

This is essentially all of the guidance which Kafka's official documentation offers regarding transactional IDs. What *are* "reliability semantics"? It's not clear: this is the only use of the word "reliability" in the documentation. The presence of a transactional ID implies idempotence, and has something to do with enforcing some kind of exclusion between multiple producer sessions sharing the same transactional ID, but how? And what transactional ID should we use?

The Java client documentation offers little more clarity:

> To use the transactional producer and the attendant APIs, you must set the transactional.id configuration property....
>
> The purpose of the `transactional.id` is to enable transaction recovery across multiple sessions of a single producer instance. It would typically be derived from the shard identifier in a partitioned, stateful, application. As such, it should be unique to each producer instance running within a partitioned application.

"Unique to each producer instance" suggests that we might want to choose a *different* transactional ID when e.g. a worker crashes and restarts. Should it be globally unique? The phrase "derived from the shard identifier" suggests that we might reuse transactional IDs across instances. And what is a session exactly? Many database clients have a first-class session API, but there appears to be no such construct in the Java Kafka client. Perhaps "session" refers to a single instance of a producer (e.g. a single client object in a single JVM on a single node), and "producer instance" refers to *multiple* such instances over time, which are intended to be logically related.

The Kafka transactions design document has a section titled "Transactional Guarantees" which elaborates on how transactional IDs ensure idempotence and transaction recovery across sessions:

> When provided with such an TransactionalId, Kafka will guarantee:
>
> 1. Idempotent production across application sessions. This is achieved by fencing off old generations when a new instance with the same TransactionalId comes online.
> 2. Transaction recovery across application sessions. If an application instance dies, the next instance can be guaranteed that any unfinished transactions have been completed (whether aborted or committed), leaving the new instance in a clean state prior to resuming work.

When no transactional ID is provided, the design document says each producer still "enjoys idempotent semantics and transactional semantics within a single session."[4]

A 2017 Confluent blog post on transactions seems to confirm this interpretation, explaining that once a producer registers its transactional ID with the cluster, it prevents any other producers ("zombies") with that same transactional ID from writing to the cluster.

> We solve the problem of zombie instances by requiring that each transactional producer be assigned a unique identifier called the `transactional.id`. This is used to identify the same producer instance across process restarts.

This hints that we should choose a unique identifier for each logical producer, and reuse it if the logical producer (e.g.) crashes and restarts, creating a new instance of the `KafkaProducer` client.[5]

The blog post also has a section helpfully titled "How to pick a transactional ID", which explains:

> The key to fencing out zombies properly is to ensure that the input topics and partitions in the read-process-write cycle is always the same for a given transactional.id. If this isn't true, then it is possible for some messages to leak through the fencing provided by transactions.
>
> For instance, in a distributed stream processing application, suppose topic-partition *tp0* was originally processed by transactional.id *T0*. If, at some point later, it could be mapped to another producer with transactional.id *T1*, there would be no fencing between *T0* and *T1*. So it is possible for messages from *tp0* to be reprocessed, violating the exactly-once processing guarantee.

The plot thickens: each transactional ID must consume from a *fixed* set of input topic-partitions. If we fail to maintain this invariant, messages might be processed multiple times—presumably, once per transactional ID.

As Kafka user Tomasz Guz explains, this critical section of Confluent's post is misleading. Two clients with different transactional IDs which each consume from a single, fixed topic-partition can both process the same message. The "key to fencing" is *not* ensuring each transactional ID has a constant set of input topic-partitions. One must instead (or, possibly, also) ensure that each input topic-partition is consumed by at most one transactional ID. Confluent's use of "for instance" is not just explaining the consequences of the previous paragraph. It appears to be introducing an *entirely different constraint*.

[4]Redpanda says this is wrong: one cannot execute transactions at all without a transactional ID.

[5]Again, Confluent's blog post uses "producer" and "producer instance" to (apparently) identify multiple instances of a `KafkaProducer` client. To be explicit, we write "producer" for a single `KafkaProducer` client object, and "producers using the same transactional ID" or "logical producer" for all producers using the same transactional ID.

Redpanda engineers make an even stronger claim: if two different transactional IDs ever interact with the same topic, guarantees within a single transactional ID, and even within a single transaction, go out the window. Users should expect duplicate delivery even within a single transaction ID. Jepsen cannot locate a source for this claim; Redpanda suspects it is implied somewhere within the sixty-seven pages of the transaction design document.

2 Test Design

We designed a test suite for Redpanda using the Jepsen testing library. We began our testing with version 21.10.1, and followed up with 21.10.2, 21.10.3, and 21.11.2, as well as various development builds through January 30, 2022. We also briefly tested Kafka 3.0.0 to compare its behavior to Redpanda. Our tests ran on both LXC containers and EC2 nodes, each running Debian Buster, with cluster sizes of 5–10 nodes.

To interact with Redpanda we used the Java Kafka client at version 3.0.0. For administrative tasks like configuring Redpanda and checking on cluster status, we used Redpanda's `rpk` command or HTTP APIs exposed by Redpanda. 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. This may have kept us from seeing safety violations.

All consumers shared a single consumer group, and when using `subscribe`, committed offsets manually after each poll operation. For transactional workloads, we gave each producer a unique transactional ID[6] and instead of committing offsets via `commitSync`, added them to each 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`, each of which has safety implications. Producers also ran with shorter timeouts, and configurable `acks`, `enable_idempotence`, and `retries`. In general we tested with the safest possible settings: default topic replications of 3, auto-commit false, acks `all`, retries 1,000, idempotence enabled, isolation level `read_committed`, `auto_offset_reset` of `earliest`, automatic creation of topics on the server disabled, and polling via `assign`.

During our tests we introduced a variety of faults, including single- and multi-node crashes and process pauses, as well as network partitions between servers. We jumped clocks forward and backwards by up to several hundred seconds, as well as strobing clocks rapidly between different times. We also performed cluster membership changes, where we politely decommissioned nodes, then assigned them new (unique) node IDs and re-added them to the cluster. At all times we preserved at least 3 active nodes (those not adding or removing), and we used a new API, added to Redpanda after 21.11.2 for this work, to determine when a node add or remove operation was complete. We did not test the impolite removal of nodes.

2.1 List-Append

The first workload we designed repurposed an existing Jepsen workload based on appends of unique values to lists. Each list is identified by a unique key. In our Redpanda list-append workload, we mapped each key to a distinct topic and partition in Redpanda. Topics were created before their first write, and our generator of operations rotated through different keys over time, limiting each key to roughly a thousand writes before creating a fresh key.[7]

Each operation in this test either appended a single unique value to the list identified by a particular key, or read all values in some key's list. We performed appends by turning the key into a Kafka `TopicPartition` and calling `producer.send` to append that value to the given topic-partition. Reads were implemented by seeking the consumer to the beginning of the given topic-partition, fetching the maximum offset of that partition via `consumer.endOffsets`, and then repeatedly calling `consumer.poll` until the maximum offset is observed.

We analyzed these histories by passing them to Elle, which inferred a dependency graph between each operation based on the values of reads and appends, plus per-process and real-time orders. Elle looked for cycles in that graph, and presented them as consistency anomalies.

2.2 Queue

As a streaming system, Redpanda's data model looks *somewhat* like a list, but our read pattern in list-append isn't how most people use Kafka and Redpanda. Instead, consumers typically assign or subscribe to a topic infrequently, and call `poll` repeatedly—making assumptions about what those polls will return. We wanted to know: are the offsets returned by `poll` contiguous? Monotonic? Can they skip over gaps? We designed a separate queue workload, which maps more closely to normal Redpanda use, to investigate these questions.

Like the list-append workload, we uniquely identified topic-partitions by integer keys, and rotated sends and polls across different keys over time.

[6]We also experimented with giving every producer the same transactional ID, but this essentially reduced the test to a single thread. Instead, we opted to use multiple transactional IDs, and not to look for exactly-once semantics across multiple transactional IDs. Redpanda maintains this is dangerous and could lead to unknown invariant violations, but Jepsen feels it is a reasonable interpretation of the (vague, somewhat contradictory) documentation.

[7]We experimented with both high and low caps on messages per key, but for most of our tests chose 1,024: high enough to require multiple calls to `poll` to fully read a partition.

In our queue workload, operations were of one of three basic classes. The first, `crash`, simulated a client failure: it terminated the logical process which executed that `crash` operation, closing its Kafka consumer and producer. Jepsen would then create a fresh process with a new producer and consumer to take its place. The second class of operations, `assign` or `subscribe`, updated the set of topics/partitions the consumer received messages from when calling `poll`: either assigning a specific set of topic-partitions (each corresponding to a single key), or subscribing to the set of topics which covered the requested key.

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

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

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

For non-transactional workloads, we constrained every `send`/`poll`/`txn` 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 message values observed at that offset, via either `send` or `poll`. We expect this mapping to be at least injective: each offset should refer to only a single message. If we observed more than one value at an offset, we recorded this as an inconsistent offset error. Since we only inserted values once, we also expected to observe no duplicate messages. If we observed a single value at multiple offsets, we identified that as a duplicate error.

When our mapping was bijective, we could construct a total order over all values with observed offsets for a given key. This order might not cover all messages in the topic-partition: calls to `send` and `poll` might not have returned offsets. Nor could we necessarily tell which offset an indeterminate-and-unobserved `send` might have produced. Moreover, not every offset contained a value: Redpanda 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 could check 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 (i.e. always increasing) and did not skip over intermediate indices. To distinguish behavior within a transaction versus between different transactions, we looked for non-monotonic or skipped offsets between two different transactions, and also within a single transaction (which we called an *internal* error).

For aborted reads, we simply searched for any poll which returned a value sent by a failed operation. For lost writes, we identified the highest read index for each key, then checked to make sure that all lower indices were also polled.

As a streaming system, Redpanda allows consumers to fall arbitrarily far behind producers: there is no expectation that consumers see up-to-date messages. Calls to `consumer.poll` tend to return successful, empty result sets regardless of whether the client is caught up on the latest messages, running behind, or talking to nodes which are completely offline, or have never run Redpanda at all. This makes it surprisingly difficult to distinguish between messages which are permanently lost versus simply delayed. Safety and liveness violations are—in this case—indistinguishable.

To address this problem, we kept track of a set of *unseen* messages: those whose `send` was successfully acknowledged, but which never appeared in any consumer's `poll` results. We plotted the number of unseen messages over time, expecting it to be nonzero for most of each test run. However, at the end of each test, we healed all network partitions, restarted and resumed any crashed or paused nodes, reset clocks, and allowed the cluster up to an hour to heal. During that healing process we performed no additional sends. Instead, we repeatedly polled every client in an attempt to catch up to any unseen messages. If we failed to observe some acknowledged messages, we reported that as an unseen error.

3 Results

Our testing identified three liveness, seven safety, and two ambiguous issues in Redpanda. We begin with duplicate writes, crashes, and inconsistent offsets, then discuss lost/stale messages and aborted reads. The second half of our results covers issues with transactions, starting with write cycles, aborted reads, and circular information flow, then moving to internal non-monotonic polls, another case of aborted read, and lost writes.

3.1 Duplicate Writes by Default (#1)

The Kafka Java client uses a default setting of `enable.idempotence = true`, which the Kafka documentation claims:

> When set to 'true', the producer will ensure that exactly one copy of each message is written in the stream. If 'false', producer retries due to broker failures, etc., may write duplicates of the retried message in the stream.

However, with Redpanda 21.10.1, any pause, crash, or network partition could cause duplicated writes with the default settings. For example, consider this test run, in which each Kafka producer logged that it was using idempotent writes:

```
ProducerConfig values:
  ...
    enable.idempotence = true
```

And yet reads of a single topic-partition returned the following messages:

```
[1 2 ... 25 26 27 28 29 30 26 27 28 29 30]
```

This workload calls `producer.send()` only once per message—but messages 26 through 30 were duplicated, thanks to the client's internal retry mechanism. This is the precise scenario which `enable.idempotence` is designed to prevent.

Stranger still, if one explicitly sets `enable.idempotence = true` in the producer config, it refuses to connect to Redpanda at all:

> UnsupportedVersionException: The broker does not support INIT_PRODUCER_ID

This is somewhat surprising, because Redpanda's FAQ claims:

> Is Redpanda Fully Kafka API Compatible?
>
> We support all parts of the Kafka API, including the transactions API that we added in release 21.8.1.

In fact Redpanda 21.10.1 required setting a configuration flag (`enable_idempotence`) to support Kafka's idempotence mechanism. Although the Java Kafka client's logging *claims* idempotence is enabled with the default settings, and although the Kafka documentation says the same, there may be a difference between "enabled by default" versus "enabled explicitly." The client may be attempting some sort of feature negotiation with the broker, and when it detects that the broker does not support idempotence, it might silently disable the feature.

We enabled idempotence in Redpanda by setting two server-side configuration variables:

```
rpk config set redpanda.id_allocator_replication 3
rpk config set redpanda.enable_idempotence true
```

The upcoming release of Redpanda 22.1.1 will enable idempotence by default, which should resolve this issue.

3.2 Duplicate Writes with Idempotence Explicitly Enabled (#3039)

With both client- and server-side idempotence explicitly enabled, we again observed duplicate writes in version 21.10.1 with single-node faults, including process crashes, pauses, or network partitions. For instance, this test with just process pauses induced five duplicate messages out of 2922 attempts:

```
{:key 7, :value 259, :count 2}
{:key 8, :value 544, :count 2}
{:key 8, :value 542, :count 2}
{:key 8, :value 543, :count 2}
{:key 8, :value 545, :count 2}
```

On key 8, polls observed messages 542, 543, 544, and 545 twice, interleaved with non-duplicated messages like 546. Take this series of messages returned from one call to `poll`:

```
... 541 542 543 545 544 543 542 544 546 545 547
```

In every case we observed, duplicates were limited to a narrow time window. The original request needed to succeed, but also have its acknowledgement message fail to arrive at the client on time, in order for a retry to occur and create a duplicate.

This behavior was caused by Redpanda failing to perform deduplication of sent messages. When the sequence numbers used to prevent duplicates arrived out of order, Redpanda returned an `OutOfOrderSequenceException` error. This seems like a reasonable choice, but it interacted poorly with clients, which interpreted that error as one they could retry. On encountering an `OutOfOrderSequenceException`, Kafka clients would increment their local epoch, reset their sequence number, and retry the request—which allowed it to take place multiple times.

In #3038 and #3039 Redpanda added support for server-side deduplication. Version 21.10.2 included deduplication support and did not exhibit duplicate messages; version 21.10.3 improved performance.

3.3 Assert Failure Deallocating Partitions (#3335)

In version 21.10.1, when Redpanda deallocated a partition from a node, it could occasionally crash. The crash handler itself then crashed due to a malformed format string, causing error messages like:

```
ERROR 2021-12-21 05:51:35,176 [shard 0] assert -
../../../src/v/cluster/scheduling/allocation_
node.cc:44@deallocate: failed to log message:
fmt='Assert failure: ({}:{})
'_allocated_partitions > allocation_capacity{0}
&& _weights[core] > 0' unable to deallocate
partition from core {} at node {}':
fmt::v7::format_error (cannot switch from
automatic to manual argument indexing)
```

This error (#3335) appeared when testing membership changes. Redpanda is still investigating.

3.4 Assert Failure in `response.partition_index` (#3336)

In rare cases involving process crashes, Redpanda 21.11.2 could occasionally encounter an assertion failure like

```
ERROR 2021-12-20 20:22:03,884 [shard 0] assert -
Assert failure: (../../../src/v/kafka/server/
handlers/fetch.cc:732) 'response.partition_index
== _it->partition_response->partition_index'
Response and current partition ids have to be
the same. Current response 0, update 1
```

We observed this error (#3336) with process crashes. It was caused by a mechanism which attempted to ensure fairness when polling multiple partitions: when a fetch request obtained messages from one partition, the server would re-order an internal cache to move that partition to the end, ensuring that the next fetch request would hit a different partition. However, this mechanism did not update a second data structure in the response message to match the new partition order. Redpanda has addressed this issue in development builds, and the fix is scheduled for version 22.1.1.

3.5 Inconsistent Offsets (#3003)

Infrequently, process kills or network partitions caused Redpanda 21.10.1 to exhibit duplicate messages which appeared at *multiple* offsets in the log—despite using `acks=all` and `retries=0`. For instance, this test run contained the following send operations on key 4:

```
{:type :ok,
 :f :send,
 :value [[:send 4 [365 381]]],
 :time 987763642489,
 :process 1737}
{:type :ok,
 :f :send,
 :value [[:send 4 [366 382]]],
 :time 988052525845,
 :process 1600}
```

Here, `[:send 4 [365 381]]` denotes a successful acknowledgement of a send to key 4: message 381 was stored at offset 365. A quarter of a second later, a call to `consumer.poll` returned message 381 at that offset:

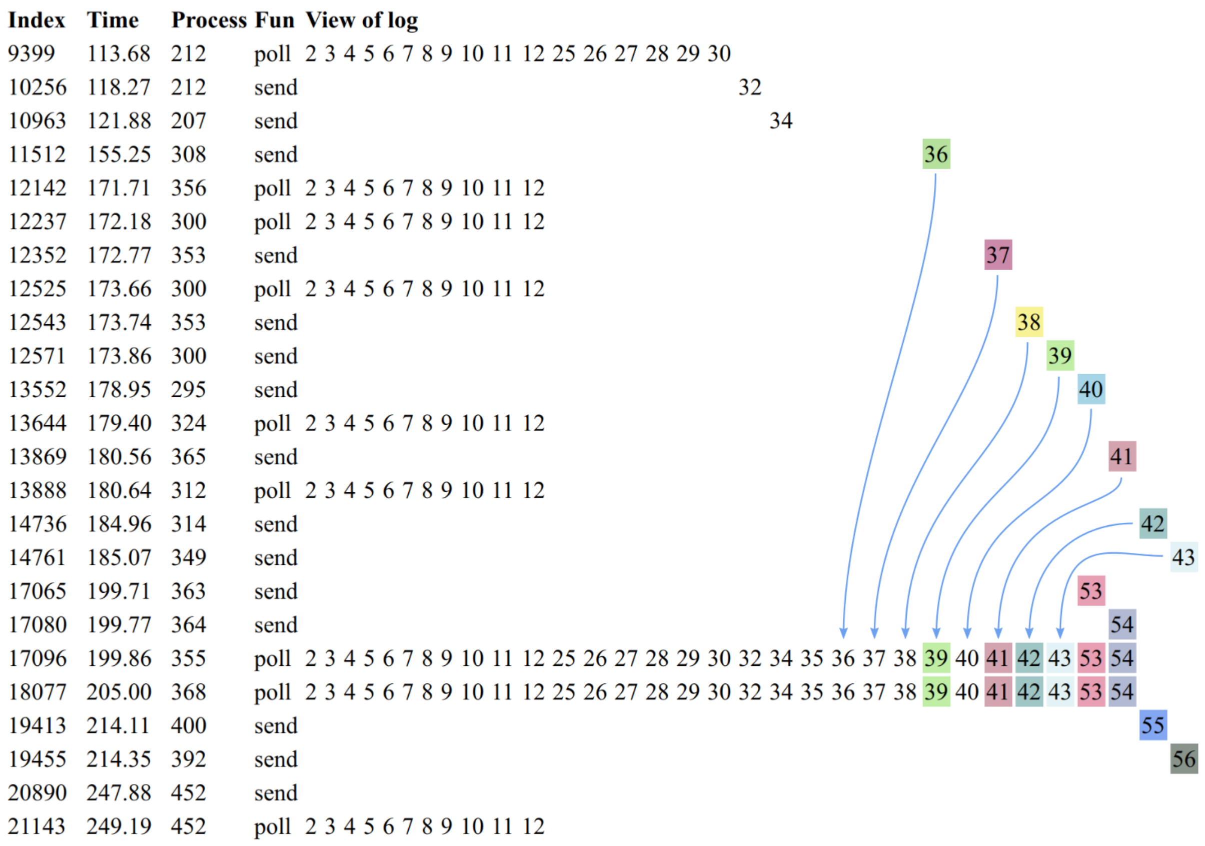
```
{:type :ok,
 :f :poll,
 :value [[:poll {4 [[0 2]
                    ...
                    [364 380]
                    [365 381]
                    [366 382]]}]],
 :time 988295099584,
 :process 1327}
```

8.7 seconds later, that same process issued another call to `consumer.poll`, which returned additional messages for key 4:

```
{:type :ok,
 :f :poll,
 :value [[:poll {4 [[367 381]
                    [368 382]
                    [369 387]
                    ...]}]],
 :time 997028597232,
 :process 1327}
```

Messages 381 and 382 were shifted two slots later in the log! Instead of occurring at offsets 365 and 366 (respectively), they now *also* occurred at offsets 367 and 378. This client processed these messages twice.

This could have been a simple duplication error—no contradictory observations of offsets 365 or 366 occurred in this test run. However, additional testing provided direct evidence of contradictory offsets in Redpanda 21.10.1. Consider this test run, where network partitions and process crashes caused 1,160 offsets to contain conflicting messages. On key 4, for example, several messages were reordered to earlier log offsets:



Index	Time	Process	Fun	View of log
9399	113.68	212	poll	2 3 4 5 6 7 8 9 10 11 12 25 26 27 28 29 30
10256	118.27	212	send	32
10963	121.88	207	send	34
11512	155.25	308	send	36
12142	171.71	356	poll	2 3 4 5 6 7 8 9 10 11 12
12237	172.18	300	poll	2 3 4 5 6 7 8 9 10 11 12
12352	172.77	353	send	37
12525	173.66	300	poll	2 3 4 5 6 7 8 9 10 11 12
12543	173.74	353	send	38
12571	173.86	300	send	39
13552	178.95	295	send	40
13644	179.40	324	poll	2 3 4 5 6 7 8 9 10 11 12
13869	180.56	365	send	41
13888	180.64	312	poll	2 3 4 5 6 7 8 9 10 11 12
14736	184.96	314	send	42
14761	185.07	349	send	43
17065	199.71	363	send	53
17080	199.77	364	send	54
17096	199.86	355	poll	2 3 4 5 6 7 8 9 10 11 12 25 26 27 28 29 30 32 34 35 36 37 38 39 40 41 42 43 53 54
18077	205.00	368	poll	2 3 4 5 6 7 8 9 10 11 12 25 26 27 28 29 30 32 34 35 36 37 38 39 40 41 42 43 53 54
19413	214.11	400	send	55
19455	214.35	392	send	56
20890	247.88	452	send	
21143	249.19	452	poll	2 3 4 5 6 7 8 9 10 11 12

This diagram shows each operation's view of the log for key 4, sorted by the time those operations completed. Time flows from top to bottom, and log offsets are arranged from left to right. When a single log offset contains conflicting values, those values are highlighted with a colored background.

Messages 36 through 43 were reordered to earlier offsets in the log. Some filled in gaps in the offsets that were initially reported to `.send()`; others overwrote messages already extant at their offsets. The resulting offsets could result in disagreement on the order of messages: senders believed 53 was inserted before 41, but pollers saw 41 before 53.

Meanwhile, on key 3, two processes both believed they were the writer of offset 78, eleven seconds apart. The first set offset 78 to 86, and the second set offset 78 to 90.

```
{:type :ok,
 :f :send,
 :value ([:send 3 [78 86]]),
 :time 490857418971,
 :process 732}
{:type :ok,
 :f :send,
 :value ([:send 3 [78 90]]),
 :time 501920547263,
 :process 806}
```

Readers of key 3 only ever observed the latter write:

```
{:type :ok,
 :f :poll,
 :value ([:poll {3 (...
                    [76 86]
                    [77 87]
                    [78 90]
                    [79 91]
                    [80 92]
                    ...)}]),
 :time 511060527454,
 :process 772}
```

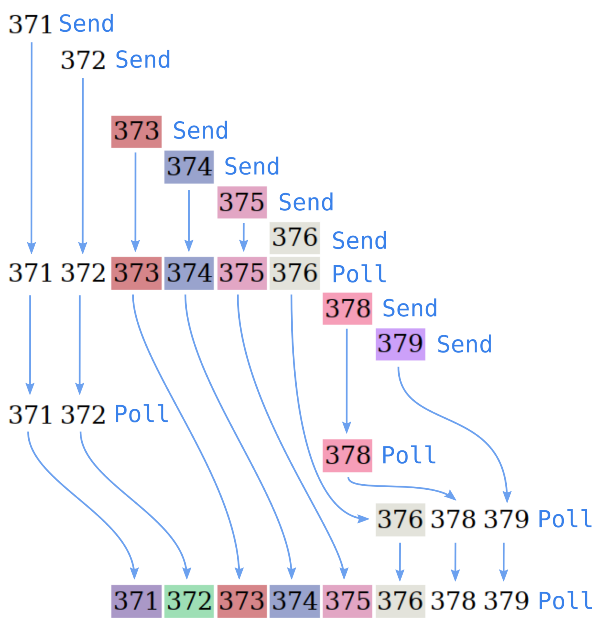
Note that this read also has message 86 at offset 76, not 78. All pollers agreed on this too!

However, on key 11, pollers did in fact disagree on which messages were at what index. Process 582 wrote message 373 at offset 242. Two hundred milliseconds later, process 438 polled key 11, and saw 373 at offset 242. 9.7 seconds later, process 486 polled key 11, and this time saw 373 reordered to offset 244: offset 242 now had value 371.

```
{:type :ok,
 :f :send,
 :value ([:send 11 [242 373]]),
 :time 258055827281,
 :process 582}
{:type :ok,
 :f :poll,
 :value ({11 (...
              [240 371]
              [241 372]
              [242 373]
              [243 374]
              [244 375]
```

```
               ...)}),
 :time 258202815546,
 :process 438}
{:type :ok,
 :f :poll,
 :value ({11 (...
              [242 371]
              [243 372]
              [244 373]
              ...)}),
 :time 267966157000,
 :process 486}
```

If we look at the surrounding neighborhood of send and poll operations (again with time flowing top to bottom, and offsets from left to right) we can see this particular disagreement between pollers was a part of a larger, more complex reordering:



Messages 371 through 379 were shifted two offsets later in the log, even after many of their original sent offsets were visible to pollers. This resulted in order inversions: both senders and some pollers thought message 376 preceded 378, but later polls showed 378 before 376—before 378 was relocated after 376 again.

In our tests Redpanda 21.10.1 and 21.10.2 would happily reorder dozens, even hundreds of messages in response to node and network faults.

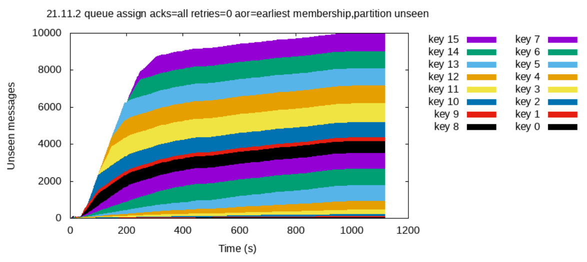
This issue was likely due to an error in Redpanda's implementation of the Raft consensus algorithm, which allowed the Redpanda state machine to apply log entries before they were known to be committed. These uncommitted entries could change if a new leader came to power, allowing a short window of split-brain. This issue (#3003) was fixed in 21.10.3 by waiting for the commit pointer to advance. We did not observe inconsistent offsets after 21.10.3.

3.6 Lost/Stale Messages (#6)

In our testing of 21.10.1, 21.10.2, 21.10.3, and 21.11.2, a variety of faults could cause successfully acknowledged messages to fail to appear in any client's polls. These missing messages were always at the end of a partition—we might observe offsets 0 through 5, but offsets 6, 7, ... would never appear in any poll.

Because `consumer.poll()` in Kafka and Redpanda is allowed to fall arbitrarily far behind producers, it was not possible to tell whether these missing messages were permanently lost or simply delayed. However, we made significant efforts to recover stale messages. At the end of each test, we ended all network partitions, unpaused any paused nodes, and restarted any crashed ones. We then tore down and recreated every client to ensure that no client was somehow "stuck." We assigned each new client the full list of topic-partitions that had been written or read during the test, and called `poll` repeatedly until it had observed every offset we'd previously seen. In case clients were stuck *again*, we repeated this teardown-and-poll process for every client every ten seconds, and extended this final polling process for up to an hour. Every node had at least one client bootstrapped from that node, to ensure that nodes with stale metadata couldn't prevent reads.

Nevertheless, we were consistently able to drive clusters into states where acknowledged messages were never seen by any client. For instance, this test run of version 21.11.2 with network partitions and membership changes caused the loss (or indefinite delay) of 9,988 out of 11,225 successfully acknowledged messages. Every single key (topic-partition) lost some messages. This stacked plot shows the number of unseen messages over time, with each key's messages in a different color.



This problem occurred despite using the strongest safety settings. Producers used `acks=all` and `retries=0`. Consumers used `auto_offset_reset=earliest` and read messages using `assign`, rather than `subscribe` (to rule out issues in the consumer group subsystem). We also increased `redpanda.default_topic_replication` to `3`, rather than the default of `1`, to make sure Redpanda's internal topics were fault-tolerant. We set `redpanda.auto_create_topics_enabled = false` to ensure Redpanda was not automatically creating under-replicated topics which would be more susceptible to data loss.

A number of faults could cause lost/stale messages. We reproduced this issue with process kills alone in

21.10.1, with membership changes and network partitions combined in 21.11.2, and with process pauses alone (as well as process crashes with membership changes) in development builds circa January 19th, 2022.

Redpanda reports that copies of missing messages could still be found in on-disk data files, but we don't know why those messages were never delivered to consumers. Redpanda is still investigating.

3.7 Aborted Read With NotLeaderOrFollower (KAFKA-13574)

In rare cases involving membership changes and process crashes, development builds of Redpanda circa December 30, 2021 exhibited what appeared to be aborted reads. For instance, this test attempted to send message 586 to key 5, which failed with a `NotLeaderOrFollowerException`. However, 586 appeared consistently in later polls of key 5:

```
{:type :ok,
 :f :poll,
 :value [[:poll {5 [...
                    [359 585]
                    [360 586]
                    [361 587]
                    ...]}]],
 :time 792033983678,
 :process 1870}
```

Jepsen and Redpanda initially suspected that `NotLeaderOrFollowerException` was a definite failure code, which would make this a case of aborted read. The Kafka documentation seemed to suggest that this error meant a request was not processed:

> Broker returns this error if a request could not be processed because the broker is not the leader or follower for a topic partition. This could be a transient exception during leader elections and reassignments. For `Produce` and other requests which are intended only for the leader, this exception indicates that the broker is not the current leader.

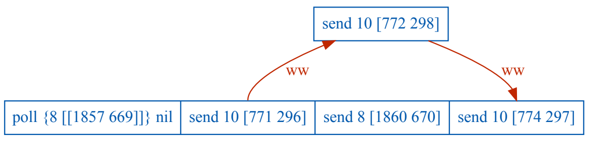
However, Kafka engineers in KAFKA-13574 informed us that `NotLeaderOrFollowerException` is in fact *indefinite*, and may signify a successful send operation. We asked the Kafka team if this was documented anywhere, but did not receive a response.

3.8 Write Cycles (#8)

The KafkaProducer documentation offers a straightforward example of transaction structure: one calls `producer.beginTransaction()`, performs one or more `producer.send()` calls, then calls `producer.commitTransaction()` to commit. Prior to commit, one can also call `producer.sendOffsetsToTransaction(...)`, which couples specific read offsets into the transaction for exactly-once semantics.

In practice, using transactions correctly is somewhat more intricate than these examples suggest. Calling `abortTransaction` appears to be mandatory in certain conditions, and illegal in others. Producers must be torn down and recreated from scratch in many (but not all!) cases. Discussions with Redpanda engineers resulted in a transaction with roughly 16 separate error-handling paths, depending on whether it was necessary to close and reopen the producer, whether an error occurred prior to or during commit, whether the abort itself was successful or crashed, and so on.

If one interprets `producer.send` as appending a record to the end of a particular topic-partition, it seems clear that transactional writes in Redpanda are not isolated from one another. Even in healthy clusters, we routinely observed transactions insert messages into the middle of other transactions' writes. This two-minute test run on a January 6th development build performed 1,594 transactions which sent a message, and contained 163 clusters of transactions whose messages interleaved with one another. For example:



The bottom transaction here sent 296 to key 10, at offset 771. However, the top transaction snuck in and wrote 298 at offset 772. The bottom transaction then wrote 297 at offset 774. This behavior occurred in every version of Redpanda we tested.

Does this constitute an isolation violation? It is certainly not equivalent to the behavior one would expect were these transactions performed sequentially. A Confluent blog post on semantics describes offsets as *monotonically increasing*, which might suggest we can infer write-write dependencies from offsets:

> An offset is a special, monotonically increasing number that denominates the position of a certain record in the partition it is in. It provides a natural ordering of records—you know that the record with offset 100 came after the record with offset 99.

And Kafka's Main Concepts and Terminology documentation specifically refers to publishing as an *append* to a topic:

> When a new event is published to a topic, it is actually appended to one of the topic's partitions.... Kafka guarantees that any consumer of a given topic-partition will always read that partition's events in exactly the same order as they were written.

This theme repeats in Kafka and Confluence documentation introducing core Kafka concepts:

> Each partition is an ordered, immutable sequence of records that is continually appended to a structured commit log. The records in the partitions are each assigned a sequential ID number called the offset, that uniquely identifies each record within the partition.

If we do interpret a Kafka partition as an "object" in the transactional sense, and calls to `send` as appends to that partition, then allowing transactions to interleave their writes is analogous to phenomenon G0: a write cycle. Per Adya, Liskov, & O'Neil, G0 ought to be disallowed under isolation level read uncommitted—not to mention read committed and higher. Moreover, the Confluence Wiki explicitly states that this behavior ought to be forbidden. Yet with Redpanda, both read uncommitted and read committed isolation levels allowed writes to interleave.

On the other hand, Redpanda argues that every individual *offset* in a partition is a separate object, and inferring write-write dependencies in this way is a category error. Under this interpretation, `producer.send` does not mean "append a message to the end of a partition". Instead, `send` means "set some unspecified offset (higher than the offset I just wrote to this partition, if any) to the given message." There are no such things as write-write dependencies in this interpretation, because every offset has only a single value; no transaction ever overwrites another. The write dependency graph is trivially empty, and no G0 anomalies exist. This also appears consistent with the behavior implied by the Exactly Once Delivery Google Doc.

Both of these views seem defensible. We do not know which interpretation Kafka & Redpanda users expect; the official Kafka documentation declines to specify, and ancillary documents contradict one another. Regardless of which interpretation one chooses, both Kafka 3.0.0 and Redpanda exhibit similar write-interleaving behavior. This behavior is a consequence of the transactional protocol itself, rather than something specific to Redpanda's implementation. We report G0 here not because it is definitively incorrect behavior, but because it could affect safety for some users, and the current behavior is underdocumented.

A bit of good news: while producer offsets frequently contained gaps due to other transactions inserting during their execution, producer offsets remained monotonic in our transactional tests.[8]

3.9 Aborted Reads & Circular Information Flow (#3036)

In 21.10.1, transactions routinely exhibited aborted reads and circular information flow under normal operation. For instance, this three-minute run without any faults, using `acks=all` and `isolation.level=read_committed`, still resulted in seven cases where a failed transaction's writes were visible to pollers. Here is a write of 567 to key 9, which failed during the `producer.sendOffsetsToTransaction` call prior to commit:

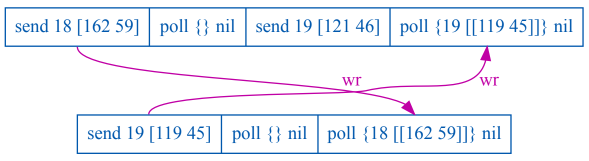
```
{:type :fail,
 :f :txn,
 :value [[:poll]
         [:poll]
         [:send 9 567]
         [:poll]],
 :time 30017247775,
 :process 4,
 :error [:add-offsets
         "Unexpected error in
         AddOffsetsToTxnResponse: The server
         experienced an unexpected error when
         processing the request."]}
```

And yet message 567 was visible to a concurrent poll operation:

```
{:type :ok,
 :f :poll,
 :value [[:poll {9 [[1477 567]]}]],
 :time 29973974218,
 :process 6}
```

567 was not visible to later readers. We frequently observed messages which were visible to some pollers and then later disappeared, both from known failed and indefinite transactions. This anomaly is G1a (aborted read), and is expressly prohibited under both ANSI and Kafka descriptions of read committed.

Moreover, 21.10.1 frequently exhibited G1c (circular information flow), even when we took into account only write-read, rather than write-write, dependencies. In that same test run we found 35 clusters of transactions where each transaction's writes were visible to every other transaction. For instance, consider this pair of transactions:



The top transaction sent message 59 to key 18, at offset 162. That same offset and value were returned to a poll by the bottom transaction. However, the bottom transaction sent message 45 at offset 119 to key 19—and that message was polled by the top transaction. At least one of these transactions must have read a value from the other *before* it was committed. Since both transactions committed, this does not constitute G1a (aborted read). This type of G1c cycle shows why

[8]This is *not* true for non-transactional workloads, where the documentation is clear: internal retries allow two `send` calls on a single producer to obtain offsets in either increasing or decreasing order.

[9]What of G1b, intermediate read? There appears to be no guarantee that consumers in Kafka & Redpanda will observe all writes from a transaction together. One could have a transaction write to partitions a and b, and a consumer assigned to both partitions could observe the write to a but not b or vice versa. We suspect G1b occurs normally, but have not experimentally confirmed this.

Adya's formalization of read committed prohibits G1a *and* G1c.[9]

Redpanda had already identified this problem before Jepsen began testing transactions, and addressed it in #3036. In short, an off-by-one error allowed the last stable offset visible to pollers to advance just past committed messages. In addition, #3232 allowed Redpanda to accidentally abort more transactions than necessary. These fixes were released in version 21.10.2, and we did not observe aborted reads or *wr*-only circular information flow in higher versions.[10]

3.10 Internal Non-Monotonic Polls (#10)

Reads introduce additional complexity into the Kafka/Redpanda transactional model. Kafka has separate clients for reading and writing data: consumers and producers are separate objects. There is no supervening client which provides transactions across both.[11] Instead, transactional methods are only available on the producer, and callers couple the highest offsets read from their consumer into the transaction by calling `producer.sendOffsetsToTransaction`.[12] For example:

```
consumer.subscribe(["some-topic"]);
producer.beginTransaction();
records1 = consumer.poll();
records2 = consumer.poll();
producer.send(record);
offsets = <highest offsets observed in records1
           and records2, each plus one>;
producer.sendOffsetsToTransaction(
  offsets,
  "some-consumer-group"
);
producer.commitTransaction();
```

What exactly are the ordering semantics of these reads?

One might expect that consumers poll offsets contiguously within a transaction, and do not skip over any messages. A skip within a transaction would be problematic because Kafka transactions do not commit each individual offset consumed, but rather the *maximum* offset consumed. Happily, in 21.11.2 and higher, we never observed skips within transactions.[13]

One might also expect that consumers poll strictly monotonic increasing offsets, so that they process messages both in order and at most once. Unfortunately, this does not appear to be true. Within a single transaction, a single consumer using `subscribe` could quietly rewind its position and poll records whose offsets fell *prior* to those they had just polled—in most cases consuming the same records multiple times. This happened regularly in healthy clusters in all versions of Redpanda we tested. Take this run of a development build from January 6, 2022, with no faults. It contained the following transaction:

```
[[:poll {25 [[924 359]
             [925 360]
             [928 361]
             [931 364]
             [935 365]
             [937 362]
             [938 363]
             [941 370]
             [944 366]
             [945 367]
             [947 372]
             [949 373]
             [952 374]
             [957 368]
             [958 375]
             [959 369]
             [963 376]]}]
 [:poll {}]
 [:poll {25 [[935 365]
             [937 362]
             [938 363]
             [941 370]
             [944 366]
             [945 367]
             [947 372]
             [949 373]
             [952 374]
             [957 368]
             [958 375]
             [959 369]
             [963 376]
             [964 371]
             [968 378]]}]
 [:send 25 [973 377]]]
```

This transaction executed three consecutive calls to `consumer.poll` with a single consumer, which had earlier been subscribed to the topic containing key 25. The first poll started with message 359 at offset 924, and continued through to offset 963. The second returned nothing. The third jumped backwards twelve messages, and returned offsets 935 to 968. Other transactions in this history jumped back to offsets completely before their previous call to `poll`.

This behavior seems dangerous. If we were trying to achieve "exactly-once" semantics, this single transaction would actually process some (but not all!) of its records twice. It might also violate ordering relationships between messages—if we processed the records in `records1`, then `records2`, we would process offset 935 immediately after 963. If message processing

[10]As previously noted, there is arguably no write isolation in Kafka transactions, which means that transactions routinely exhibit G1c cycles involving *ww* and *wr* edges: for instance, where T_1 sends a message before T_2, and goes on to read a message sent by T_2. Like G0, this behavior is also prohibited under Adya et al's formalization of read committed.

[11]The Kafka Streams library might provide this functionality, but we have not explored it.

[12]The Kafka client actually offers two flavors of `sendOffsetsToTransaction`. The one which Redpanda supports is deprecated; the one which Kafka recommends as of their 3.0.0 release throws `UnsupportedVersionException` on every invocation.

[13]We did routinely observe skips *between* two successive transactions executed by the same poller when using `subscribe`. This makes sense—consumers in a consumer group trade off consuming offsets, so no single consumer should expect to see every record.

were non-commutative, this could lead to unexpected results.

Is this behavior expected? The Kafka consumer documentation seems to suggest that consumers should return sequentially increasing offsets:

> The `position` of the consumer gives the offset of the next record that will be given out. It will be one larger than the highest offset the consumer has seen in that partition. It automatically advances every time the consumer receives messages in a call to `poll(Duration)`.

And yet this is clearly not what happens! This is especially vexing because all the state required to prevent this kind of non-monotonic behavior is already present in the consumer. The consumer *knows* that its previous `poll` returned offset 963. It can therefore enforce that its next `poll` returns offset 964 or higher!

As it turns out internal non-monotonic polls are a consequence of of *consumer group rebalance events*, where the Kafka client and Redpanda coordinate to automatically reassign partitions among subscribers. A more thorough investigation revealed that consumers were having their partitions automatically reassigned within the scope of a transaction, causing those consumers to return to earlier positions in the log.

To avoid this hazard users must provide a `ConsumerRebalanceListener`, which receives callbacks indicating changes in partition assignment. When a rebalance event occurs, the client can abort the current transaction—preventing it from observing messages out-of-order.

3.11 Aborted Read With `InvalidTxnState` (#3616-a)

We frequently observed transactions which appeared to fail, but whose writes were visible to later reads. With process pauses, crashes, or network partitions, versions 21.10.1, 21.11.2, and development builds in early January 2022 would reliably throw `InvalidTxnStateException` when committing transactions, but the writes performed by those transactions might later be visible. In this test run, for instance, we encountered 66 transactions like so:

```
{:type :fail,
 :f :txn,
 :value [[:send 7 [97 32]]
         [:send 6 [33 11]]
         [:poll {9 [[297 120]]}]],
 :time 68499617337,
 :process 9,
 :end-process? true,
 :error
 {:type :abort,
  :abort-ok? false,
  :tried-commit? true,
  :definite? true,
  :body-error "org.apache.kafka.common.errors.
               InvalidTxnStateException: The
               producer attempted a
               transactional operation in an
               invalid state.",
  :abort-error "org.apache.kafka.common.
                KafkaException: Cannot
                execute transactional method
                because we are in an error
                state"}}
```

This transaction attempted to send message 32 to key 7, but when it called `producer.commitTransaction()`, received an `InvalidTxnStateException`. It then attempted to abort the transaction, but the abort call failed because the producer was in an error state. Message 32 then appeared in later reads:

```
{:type :ok,
 :f :txn,
 :value [...
         [:poll
          {7 [[85 28]
              [87 29]
              [91 30]
              [94 31]
              [97 32]],
           9 [[297 120]
              [301 121]
              [302 122]]}]],
 :time 69250120511,
 :process 10}
```

"Attempted in an invalid state" sounds like a definite failure: if the state were invalid before the attempt to commit, how could it possibly have succeeded? The documentation for `InvalidTxnStateException` says nothing about its meaning, and the `commitTransaction` documentation does not mention it in their list of possible exceptions. We asked the Kafka team about this error code, but did not receive a response.

This turned out to be a bug in Redpanda's transaction path. A client would commit a transaction, but the transaction coordinator crashed just prior to acknowledging that transaction to the client. The client would time out and attempt to retry the commit. On receiving that second commit request for an already committed transaction, the server would respond with `invalid_txn_state`—which the client might interpret as a failure. Unfortunately, the Kafka transactional protocol does not include a unique identifier for transactions, which makes it difficult to tell *which* transaction is being committed when a coordinator fails. Redpanda cannot tell, in general, whether a retried commit request should succeed or fail.

This issue was addressed via a suite of transaction improvements in #3616. Redpanda now returns an `unknown_server_error`, which more clearly signals the transaction's indeterminate state. In development builds after January 21, 2022, we no longer observed aborted reads with transactions. This issue was fixed in 21.11.15.

3.12 Lost Transactional Writes (#3616-b)

In 21.11.2, as well as development builds circa January 6th and 19th, 2022, we observed occasional cases in healthy clusters where writes performed by a successfully committed transaction would vanish, never to be seen again. In this test run, the following single-write transaction successfully committed:

```
{:type :ok,
 :f :send,
 :value [[:send 22 [1903 689]]],
 :time 823319861300,
 :process 215}
```

And yet offset 1903 and value 689 never appeared in any poll. The immediately following poll skipped right over it. So too did every final poll, which used `assign` and `seekToBeginning` to attempt to read the entire partition in order. All observed something like:

```
{:type :ok,
 :f :txn,
 :value ([:poll {22 [...
                     [1895 682]
                     [1898 683]
                     ; No 1903!
                     [1908 688]
                     [1911 690]
                     [1912 691]
                     [1913 692]
                     ...]}]),
 :time 824502875814,
 :process 201}
```

Like the transactional aborted reads discussed just prior, this issue was addressed as a part of a package of transaction protocol improvements in #3616. When applying log operations to the local state, a Redpanda leader would check to make sure that they were still the most current leader, and if that check succeeded, go on to perform multiple state transitions. However, the node could apply an action, lose leadership, regain leadership, then go on to apply another action—falsely assuming that it had been the sole leader the entire time. This could allow state machine operations to interleave incorrectly. Redpanda suspects that this caused successfully committed transactions to be lost.

The problem was resolved in development builds circa January 21, 2022. Redpanda now reads the current term prior to performing multiple state machine actions, and ensures that term is still current when applying each action. The fix was released in 21.11.15.

№	Summary	Event Required	Fixed In
1	Duplicate writes by default	Pause, crash, or partition	22.1.1*
3039	Duplicate writes with idempotence	Pause, crash, or partition	21.10.3
3335	Assert failure deallocating partitions	Membership change	Unresolved
3336	Assert failure involving partition IDs	Crash	22.1.1*
3003	Inconsistent offsets	Crash or partition	21.10.3
6	Lost/stale messages	Pause, crash, or partition	Unresolved
KAFKA-13574	Aborted read with NotLeaderOrFollower	Membership change & pause	Unresolved
8	Write cycles	None	Unresolved
3036	Aborted read & circular information flow	None	21.10.2
10	Internal non-monotonic polls	None	Unresolved
3616-a	Aborted read with InvalidTxnState	Pause or crash	21.11.15
3616-b	Lost transactional writes	None	21.11.15

** 22.1.1 is an upcoming release*

4 Discussion

We identified ten issues (seven safety, three liveness) in Redpanda 21.10.1 through 21.11.2, including crashes, duplicated messages, non-monotonic polls, aborted reads, circular information flow, inconsistent offsets, and lost or delayed writes. Some of these issues, like aborted reads and lost writes, occurred in healthy clusters. Others required only minor faults: a process pausing for a few seconds could cause data loss. Both non-transactional and transactional workloads exhibited safety violations.

We also identified two safety behaviors which might be surprising, but are not necessarily incorrect. The first is an ambiguous error, `NotLeaderOrFollowerException`, which might suggest to users that a given write did not succeed when, in fact, it did. The second is that transactions provide no write isolation, allowing G0 (write cycle), G1b (intermediate read), and G1c (circular information flow) with write-write edges. G1c with only write-read edges is prevented in 21.10.2 and higher. Documentation disagrees as to whether this should be prohibited or allowed.

The Redpanda team already had an extensive test suite— including fault injection—prior to our collaboration. Their work found several serious issues including duplicate writes (#3039), inconsistent offsets (#3003), and aborted reads/circular information flow (#3036) before Jepsen encountered them. Redpanda has also extended their test suite to reproduce new issues Jepsen identified.

The most frequent safety issues we found were resolved in 21.10.3, but some problems, including aborted reads and lost writes, remained extant in 21.11.2. Aborted reads (#3616-a) and lost transactional writes (#3616-b) were fixed in the just-released 21.11.15. Fixes for duplicate writes by default (#1) and assert failures (#3336) are scheduled for version 22.1.1. We recommend users upgrade to 21.11.15 as soon as feasible, and 22.1.1 once available, to reduce the probability of safety errors.

Two issues remain to be investigated and patched: a crash when deallocating partitions (#3335) and lost/stale messages (#6). Aborted reads with NotLeaderOrFollower (#KAFKA-13574), write cycles (#8), and internal non-monotonic polls (#10) can all be addressed through documentation, but these docs have not yet been written.

We did not observe any safety issues related to clock skew, which makes sense: Redpanda uses Raft extensively for state machine replication, and does not rely on wall-clock timestamps for correctness. While the Kafka protocol does involve heartbeats and timeouts for consumer liveness, it uses logical epochs and sequence numbers for safety.

Jepsen found Redpanda straightforward to install and operate during testing. The `rpk` administrative tool made configuration simple, although we did have a minor issue with `rpk` altering config file ownership and making the config file unreadable to Redpanda itself. Node startup and cluster join were fast and robust. Clusters recovered in a few seconds from crashes and partitions. With minor surprises (namely, idempotence and transactions), the Java Kafka client worked seamlessly out of the box. Membership changes involved undocumented HTTP APIs (and new ones had to be built in order to perform membership changes safely) but Jepsen is confident that this process can be streamlined and integrated into `rpk`.

As always, we note that Jepsen takes an experimental approach to safety verification: we can prove the presence of bugs, but not their absence. While we try hard to find problems, we cannot prove the correctness of any distributed system.

4.1 Mild Surprises

During our testing we encountered a number of minor surprises in the Kafka Java client and Redpanda proper. We'd like to discuss these briefly, in the hopes that they might help other users.

First, network clients typically throw either when connecting to or reading from nodes which are unavailable. A `KafkaConsumer`, by contrast, will happily connect to a jar of applesauce[14] and return successful, empty result sets for every call to `consumer.poll`. This makes it surprisingly difficult to tell the difference between "everything is fine and I'm up to date" versus "the cluster is on fire", and led to significant confusion in our tests.

[14]Assuming the applesauce jar speaks TCP/IP.

Consumers can use `commitSync` to inform Redpanda (or Kafka, as the case may be) that they have processed messages up to some offset. When producers fail and restart, they can pick up at the committed offset to avoid re-processing too many records. One might assume that the committed offset for a given topic-partition is monotonic, but this is not true. Committed offsets are allowed to go backwards, effectively un-committing committed messages. This could occur if a commit network message is delayed due to a process pause or network hiccup. In practice this seems unlikely to cause safety issues: users should already assume that in most cases, messages in Kafka/Redpanda will be processed multiple times.

Kafka and Redpanda can automatically create topics when producers send or consumers poll. By default, the replication factor for these topics in Redpanda is `1`, which means that if a client happens to interact with a topic prior to its official creation, and auto-creation is enabled on the client and server, one might end up with a topic with essentially no fault-tolerance. Users can increase the replication factor for these newly created topics by setting `redpanda.default_topic_replication` to three or higher. You can also disable topic autocreation by setting `redpanda.auto_create_topics_enabled = false`.

4.2 Non Fault-Tolerant Defaults

Even in production mode, Redpanda did not provide fault tolerance by default for its transaction coordinator, ID allocator, and internal metadata. This allowed Redpanda to run on a single-node installation out of the box, but could lead to safety issues if a single node fails. We recommend users with multi-node clusters set `redpanda.default_topic_replications`, `redpanda.id_allocator_replication`, and `redpanda.transaction_coordinator_replication` to at least 3, in order to survive the failure of any individual node. Redpanda plans to address this in future releases. This requirement remains undocumented.

Redpanda still does not enable idempotence support by default. This could cause clients which default to idempotence to silently duplicate messages. We recommend Redpanda users set `redpanda.enable_idempotence = true` to avoid this problem. Redpanda plans to enable idempotence by default in 22.1.1.

4.3 Transaction Isolation

Users of Kafka & Redpanda transactions should be aware that transactions may, depending on one's interpretation of write conflicts for `send` operations, allow G0 (write cycle) and G1c (circular information flow) by design. Two transactions may interleave their writes together. Transaction T_1 can write a message prior to a write by T_2, then go on to read T_2's writes before committing. We suspect, but have not demonstrated, that

G1b (intermediate read) is also allowed by the transaction protocol which Redpanda and Kafka share. Users should expect these behaviors in their transactions so long as the Kafka transaction protocol remains unchanged. Since the official Kafka documentation declines to specify transactional write isolation, and ancillary documentation makes contradictory claims, it's hard to say what the "correct" behavior ought to be.

In 21.10.1, transactions allowed G1c cycles comprised entirely of write-read edges: T_1 could write something which T_2 read, and T_2 could write something which T_1 read. This behavior was a bug, and was fixed in 21.10.2.

We identified multiple cases of G1a (aborted read) in Redpanda transactions, up to and including version 21.11.2. These too were bugs, but all instances of G1a we identified should be fixed as of Redpanda 21.11.15. We believe users of Redpanda 21.11.15 who use transactions with `isolation_level = read_committed` should not observe G1a or G1c cycles comprised entirely of write-read dependencies. Running with `isolation_level = read_uncommitted` will likely still allow both phenomena.

4.4 Exactly-Once Semantics

We never attempted to check exactly-once processing across multiple clients, because *individual* clients would routinely double-process messages in our tests—even within a single transaction. This could be caused by the fact that calls to `consumer.poll` were often non-monotonic, going back to re-read some or all messages previously observed in a single transaction. It could also be because our test broke some poorly documented rules of Kafka transactions.

We initially patterned our queue transactional workload after Kafka's example code for exactly-once transactional processing, but intended to measure only atomicity and isolation, rather than exactly-once processing. Consequently we made two changes which might invalidate global exactly-once semantics: we performed multiple calls to `poll` in a single transaction, and we allowed multiple transactional IDs to consume from a single partition.

Since the Kafka transactions design document specifies that a single transaction may perform multiple requests over a long phase, we replaced the example's single call to `consumer.poll` with a short, random series of polls and sends. Performing multiple reads is a typical pattern in other transaction systems, and we found nothing in Kafka nor Redpanda's documentation which says this is illegal.

Where the Kafka demo used `GROUP_INSTANCE_ID_CONFIG` to establish a static mapping of consumers to partitions, we used a single consumer group with automatic rebalancing. This allowed two clients with different transactional IDs to consume from the same partition. Neither the official documentation nor the Javadocs discuss the semantics of multiple transactional IDs—though they do suggest IDs be "unique to each producer instance." A careful reading of a Confluent blog post suggests this should have allowed duplicate processing of messages across different transactional IDs, but says nothing about behavior within a single transaction or transactional ID. The Kafka transactions design doc indicates that in the *absence* of transactional IDs, each producer still "enjoys idempotent semantics and transactional semantics within a single session," but declines to say whether those guarantees hold when transactional IDs are provided but differ.

Redpanda believes these choices essentially invalidate our transactional results: performing multiple polls per transaction and allowing multiple transactional IDs to consume from the same partition means that we cannot expect safety even within a single transaction—let alone two transactions performed by clients with the same transactional ID. We cannot locate a source for this claim in the Kafka or Redpanda literature; if true, it may be undocumented.

Instead, Redpanda suggested that obtaining safe transactional semantics requires a more complex dance in which one generates producers with specifically chosen transactional IDs based on source partitions. Each producer reads saved offsets for the partitions they intend to interact with, writes them back in an otherwise empty transaction to ensure concurrent producers haven't overwritten those offsets, seeks to those offsets, and then performs transactions as normal. Neither this approach nor this particular code were referenced in any public documentation, and Jepsen is unsure how users would have learned to do this on their own. We did not have time to implement this pattern in our test suite, but other Kafka and Redpanda users might find it helpful.

All of this is complicated by official documentation which is absent, incomplete, vague, byzantine, or simply wrong. Vendor and third-party blog posts can be downright confusing. Users must piece together all of these resources, which often phrase behavior not in terms of application-level invariants but via the implementation details of an interlocking collection of distributed locking, idempotence, atomicity, retry, and crash-recovery mechanisms split across readers and writers. In short, Kafka and Redpanda offer less of a transaction system in the sense that database users are accustomed to, and more of a choose-your-own-adventure book in which half the pages are missing, critical plot points are scrawled in the margins by other readers, and many paths lead to silent invariant violations.

We stress that this is not Redpanda's fault: conformance with the Kafka wire protocol significantly constrains what Redpanda can offer users. Nevertheless, better documentation would be a blessing.

4.5 Membership Changes

As of February 1, 2022, Redpanda documentation did not mention how to remove nodes from the cluster. Instead Redpanda clients learned how to remove and add nodes in support channels on Slack. Redpanda has since added some documentation for the `decommission` command.

These membership changes were difficult to execute correctly in unhealthy networks. Both `rpk cluster info` and the `/v1/brokers` API could return arbitrarily stale views of the cluster, which could lead to operators concluding that a node had completed its decommissioning process when it was, in fact, still handing off data to other members. We suspect that in practice this should be a rare occurrence: most network partitions last a few days at most, and nodes are usually not removed immediately after being added.

Still, operators who wish to perform node changes more safely (or rapidly!) can use a new API: `/v1/cluster_view`. This view of the cluster includes a monotonic `version` field which can be used to ensure one sees only successive views of the cluster state. To safely remove a node, one should:

1. Read a cluster view containing the node ID one intends to remove.
2. Issue a decommission request for that node.
3. Wait until one has read a cluster view which is of a higher version than the initial view, and does not contain the given node ID.

After this, it should be safe to tear down the decommissioned node. This API was just released in version 21.11.15.

Another potential pitfall: one should always generate new, unique node IDs when adding nodes to a Redpanda cluster. Reusing previously generated IDs—even if those nodes are no longer a part of the cluster—could lead to data loss. This too was undocumented.

Redpanda is expanding their documentation to explain membership change operations in detail.

4.6 Future Work

Normally, Jepsen tests bind each logical process in the test to a single node in the cluster, and stripe processes across nodes. This allows Jepsen to observe the state of different nodes simultaneously, which is key to finding concurrency bugs. The Java Kafka client, by contrast, fetches the list of all nodes in the cluster from whichever node it initially connects to, and from there establishes independent connections to different servers on its own. We were unable to easily interfere with this process, which means that our tests may have failed to identify split-brain or other anomalies between nodes. Every client may, assuming a stable topology, have routed all their requests for a given partition to a single node, rather than multiple nodes. We would like to revisit this problem and devise a way to constrain Kafka clients to talk *only* to specific nodes.

Our membership tests only examined polite node removal, where we issued a decommission request and allowed the node to hand off its data before wiping the node clean. Future testing might explore limited numbers of unplanned node removals, to explore what happens when (e.g.) a hard disk fails and cannot be recovered. We also discussed the possibility of injecting filesystem-level faults, but have not implemented them yet.

Our queue and list-append workloads used a fixed number of partitions per topic. Future work could expand the number of partitions dynamically.

Finally, we were unable to obtain exactly-once semantics during our transaction testing—possibly because of the structure of our transactions. Redpanda has suggested a more complex way of using the transaction API which might provide stronger guarantees. We'd like to integrate that into the Jepsen tests someday.

Jepsen carried out an informal poll of a handful of Kafka users to better understand their use of transactions. Several reported they had adopted the Kafka Streams API rather than using the producer transaction API directly. Perhaps Kafka Streams offers stronger guarantees! Future tests might explore Kafka Streams behavior, and see whether it prevents some of the transactional anomalies we observed.

This work was funded by Redpanda Data, and conducted in accordance with the Jepsen ethics policy. Jepsen wishes to thank the Redpanda team for their assistance—especially Camilo Aguilar, Travis Bischel, Bob Dever, Juan Castillo, Alexander Gallego, Dhruv Gupta, Michal Maslanka, Denis Rystsov, John Spray, Coral Waters, David Wang, and Noah Watkins. We would also like to thank Irene Kannyo for her editorial support during preparation of this manuscript.