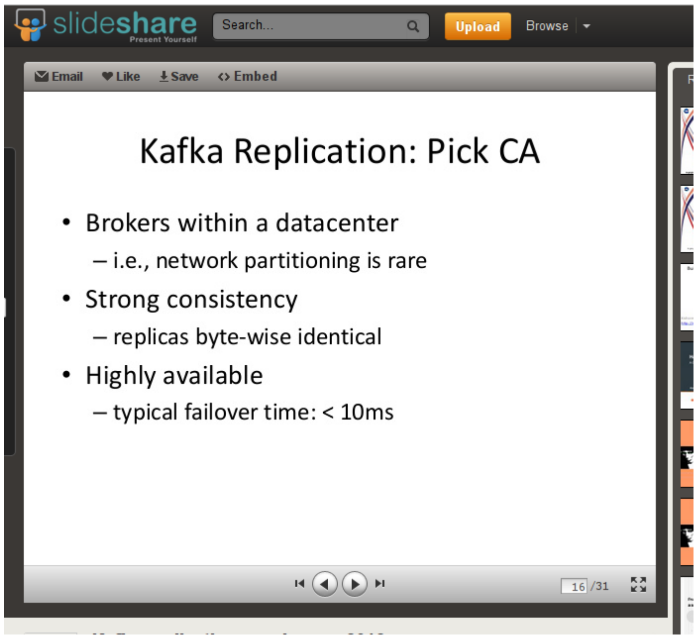
Aphyr

Jepsen: Kafka

Software Network Distributed Systems Jepsen Kafka
2013-09-24

In the last Jepsen post, we learned about NuoDB. Now it's time to switch gears and discuss Kafka. Up next: Cassandra.

Kafka is a messaging system which provides an immutable, linearizable, sharded log of messages. Throughput and storage capacity scale linearly with nodes, and thanks to some impressive engineering tricks, Kafka can push astonishingly high volume through each node; often saturating disk, network, or both. Consumers use Zookeeper to coordinate their reads over the message log, providing efficient at-least-once delivery–and some other nice properties, like replayability.



Kafka Replication: Pick CA

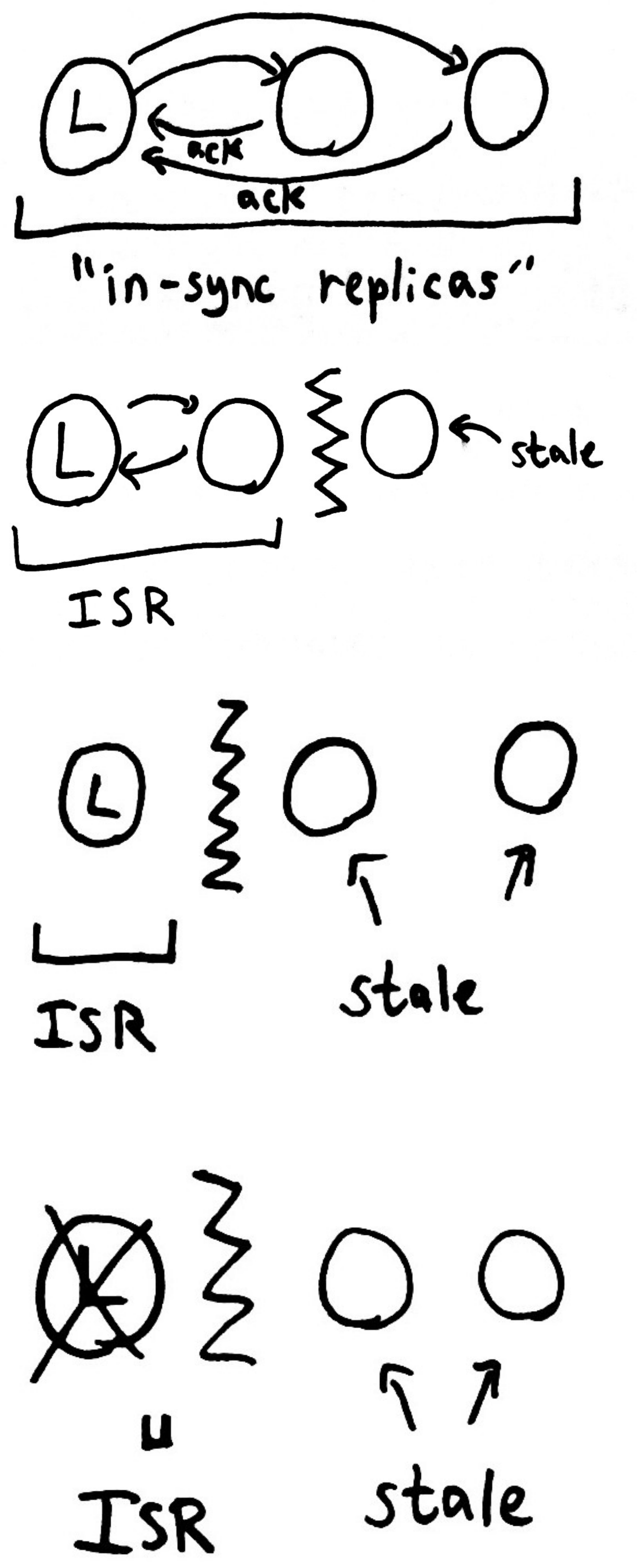
- Brokers within a datacenter
  - i.e., network partitioning is rare
- Strong consistency
  - replicas byte-wise identical
- Highly available
  - typical failover time: < 10ms

In the upcoming 0.8 release, Kafka is introducing a new feature: replication. Replication enhances the durability and availability of Kafka by duplicating each shard's data across multiple nodes. In this post, we'll explore how Kafka's proposed replication system works, and see a new type of failure.

Here's a slide from Jun Rao's overview of the replication architecture. In the context of the CAP theorem, Kafka claims to provide both serializability and availability by sacrificing partition tolerance. Kafka can do this because LinkedIn's brokers run in a datacenter, where partitions are rare.

Note that the claimed behavior isn't impossible: Kafka could be a CP system, providing "bytewise identical replicas" and remaining available whenever, say, a majority of nodes are connected. It just can't be fully available if a partition occurs. On the other hand, we saw that NuoDB, in purporting to refute the CAP theorem, actually sacrificed availability. What happens to Kafka during a network partition?

Design

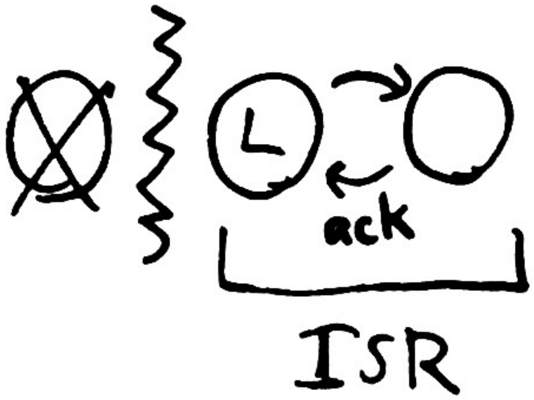


Kafka's replication design uses leaders, elected via Zookeeper. Each shard has a single leader. The leader maintains a set of in-sync-replicas: all the nodes which are up-to-date with the leader's log, and actively acknowledging new writes. Every write goes through the leader and is propagated to every node in the In Sync Replica set, or ISR. Once all nodes in the ISR have acknowledged the request, the leader considers it committed, and can ack to the client.

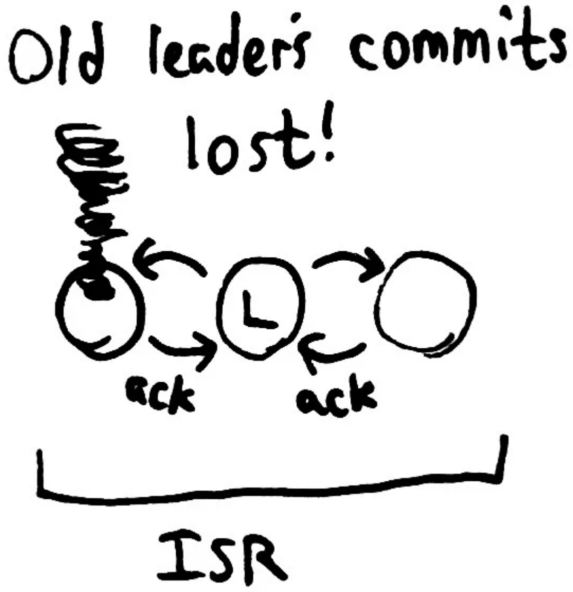
When a node fails, the leader detects that writes have timed out, and removes that node from the ISR in Zookeeper. Remaining writes only have to be acknowledged by the healthy nodes still in the ISR, so we can tolerate a few failing or inaccessible nodes safely.

So far, so good; this is about what you'd expect from a synchronous replication design. But then there's this claim from the replication blog posts and wiki: "with f nodes, Kafka can tolerate f-1 failures".

This is of note because most CP systems only claim tolerance to n/2-1 failures; e.g. a majority of nodes must be connected and healthy in order to continue. Linkedin says that majority quorums are not reliable enough, in their operational experience, and that tolerating the loss of all but one node is an important aspect of the design.



Kafka attains this goal by allowing the ISR to shrink to just *one* node: the leader itself. In this state, the leader is acknowledging writes which have been only been persisted locally. What happens if the leader then loses its Zookeeper claim?



The system *cannot* safely continue–but the show must go on. In this case, Kafka holds a new election and promotes any remaining node–which could be arbitrarily far behind the original leader. That node begins accepting requests and replicating them to the new ISR.

When the original leader comes back online, we have a conflict. The old leader is identical with the new up until some point, after which they diverge. Two possibilities come to mind: we could preserve *both* writes, perhaps appending the old leader's writes to the new–but this would violate the linear ordering property Kafka aims to preserve. Another option is to drop the old leader's conflicting writes altogether. This means destroying committed data.

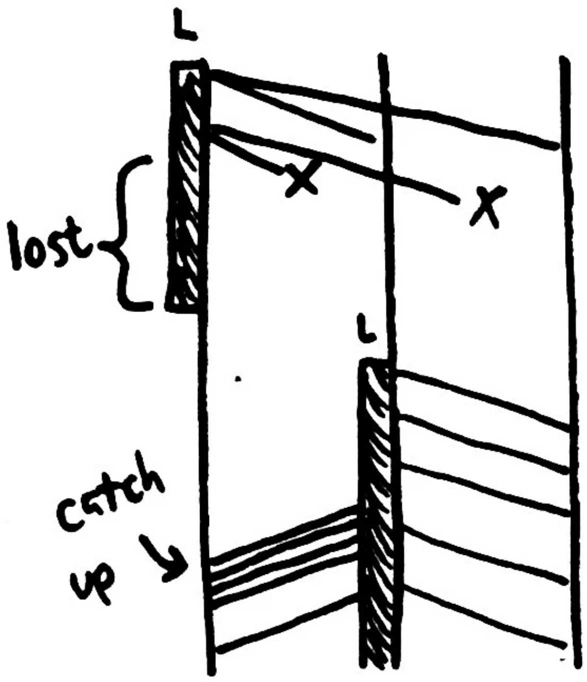
In order to see this failure mode, two things have to happen:

1. The ISR must shrink such that some node (the new leader) is no longer in the ISR.
2. All nodes in the ISR must lose their Zookeeper connection.

For instance, a lossy NIC which drops some packets but not others might isolate a leader from its Kafka followers, but break the Zookeeper connection slightly later. Or the leader could be partitioned from the other kafka nodes by a network failure, and then crash, lose power, or be restarted by an administrator. Or there could be correlated failures across multiple nodes, though this is less likely.

In short, two well-timed failures (or, depending on how you look at it, one complex failure) on a single node can cause the loss of arbitrary writes in the proposed replication system.

I want to rephrase this, because it's a bit tricky to understand. In the causality diagram to the right, the three vertical lines represent three distinct nodes, and time flows downwards. Initially, the Leader (L) can replicate requests to its followers in the ISR. Then a partition occurs, and writes time out. The leader detects the failure and removes nodes 2 and 3 from the ISR, then acknowledges some log entries written only to itself.



When the leader loses its Zookeeper connection, the middle node becomes the new leader. What data does it have? We can trace its line upwards in time to see that it only knows about the very first write made. All other writes on the original leader are *causally disconnected* from the new leader. This is the reason data is lost: the causal invariant between leaders is violated by electing a new node once the ISR is empty.

I suspected this problem existed from reading the JIRA ticket, but after talking it through with Jay Kreps I wasn't convinced I understood the system correctly. Time for an experiment!

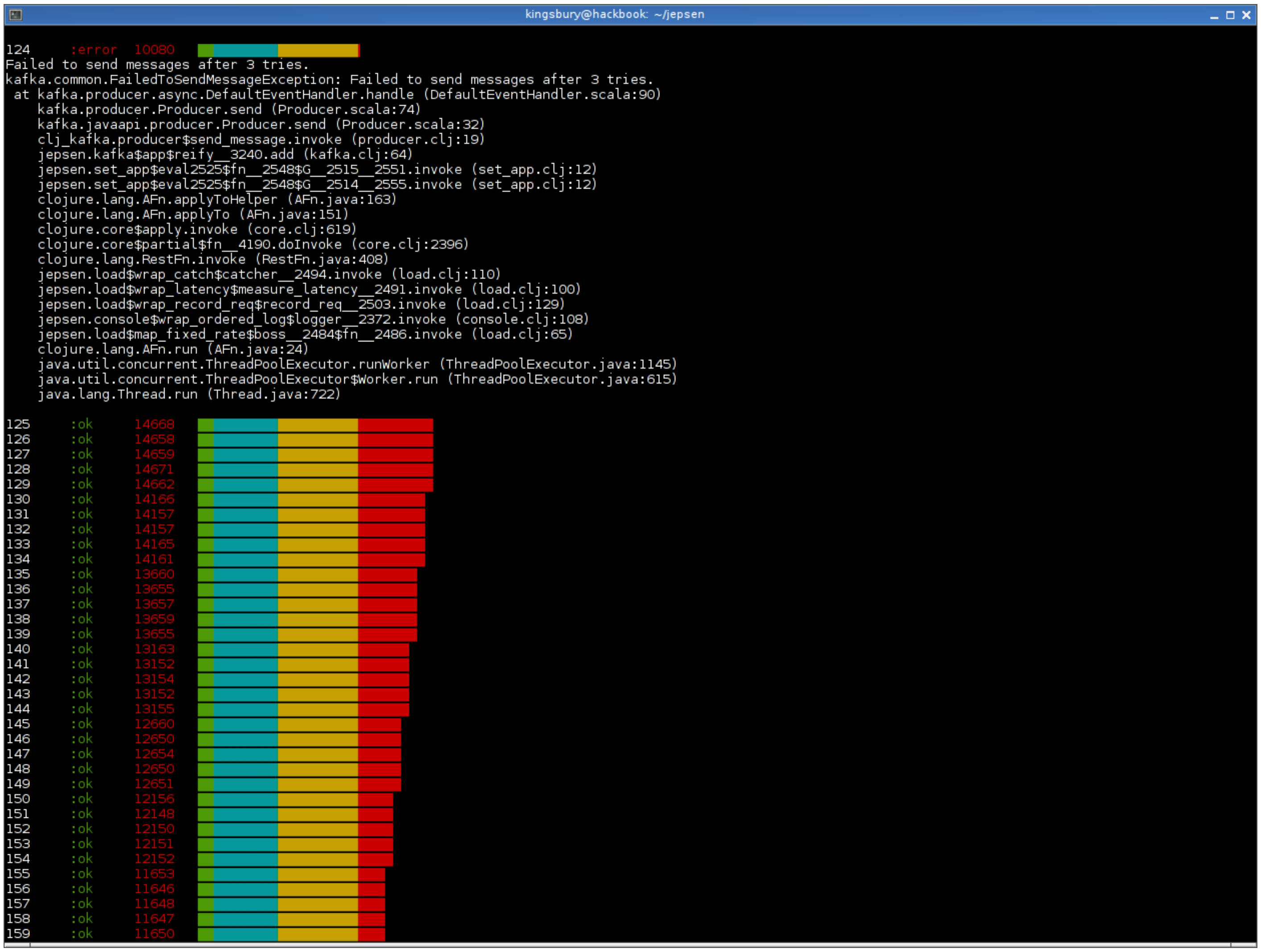
Results

First, I should mention that Kafka has some parameters that control write consistency. The default behaves like MongoDB: writes are not replicated prior to acknowledgement, which allows for higher throughput at the cost of safety. In this test, we'll be running in synchronous mode:

```
(producer/producer
  {"metadata.broker.list" (str (:host opts) ":9092")
   "request.required.acks" "-1" ; all in-sync brokers
   "producer.type"         "sync"
   "message.send.max_retries" "1"
   "connect.timeout.ms"    "1000"
   "retry.backoff.ms"       "1000"
   "serializer.class"     "kafka.serializer.DefaultEncoder"
   "partitioner.class"    "kafka.producer.DefaultPartitioner"})
```

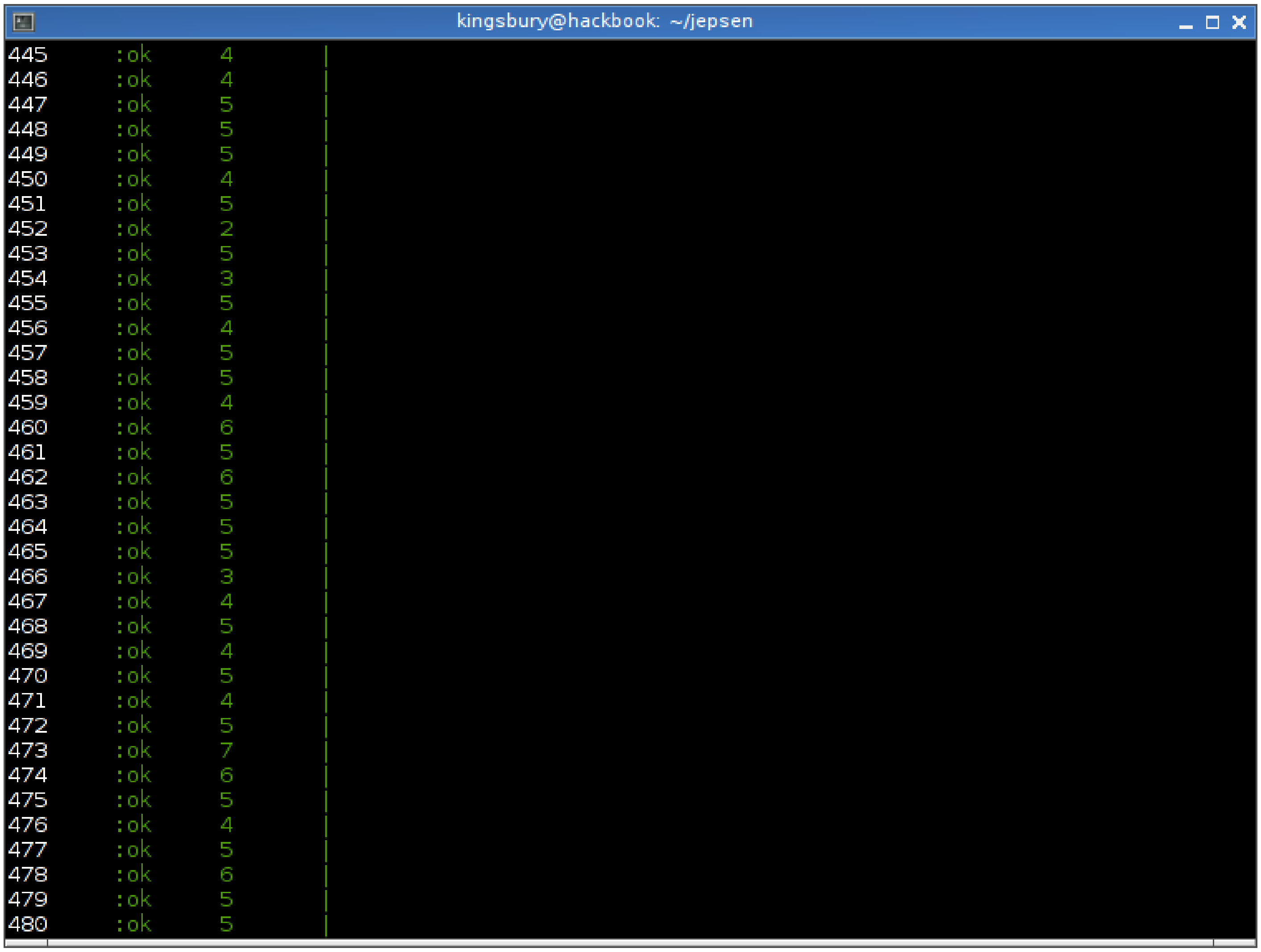
With that out of the way, our writes should be fully acknowledged by the ISR once the client returns from a write operation successfully. We'll enqueue a series of integers into the Kafka cluster, then isolate a leader using iptables from the other Kafka nodes. Latencies spike initially, while the leader waits for the missing nodes to respond.

A few requests may fail, but the ISR shrinks in a few seconds and writes begin to succeed again.



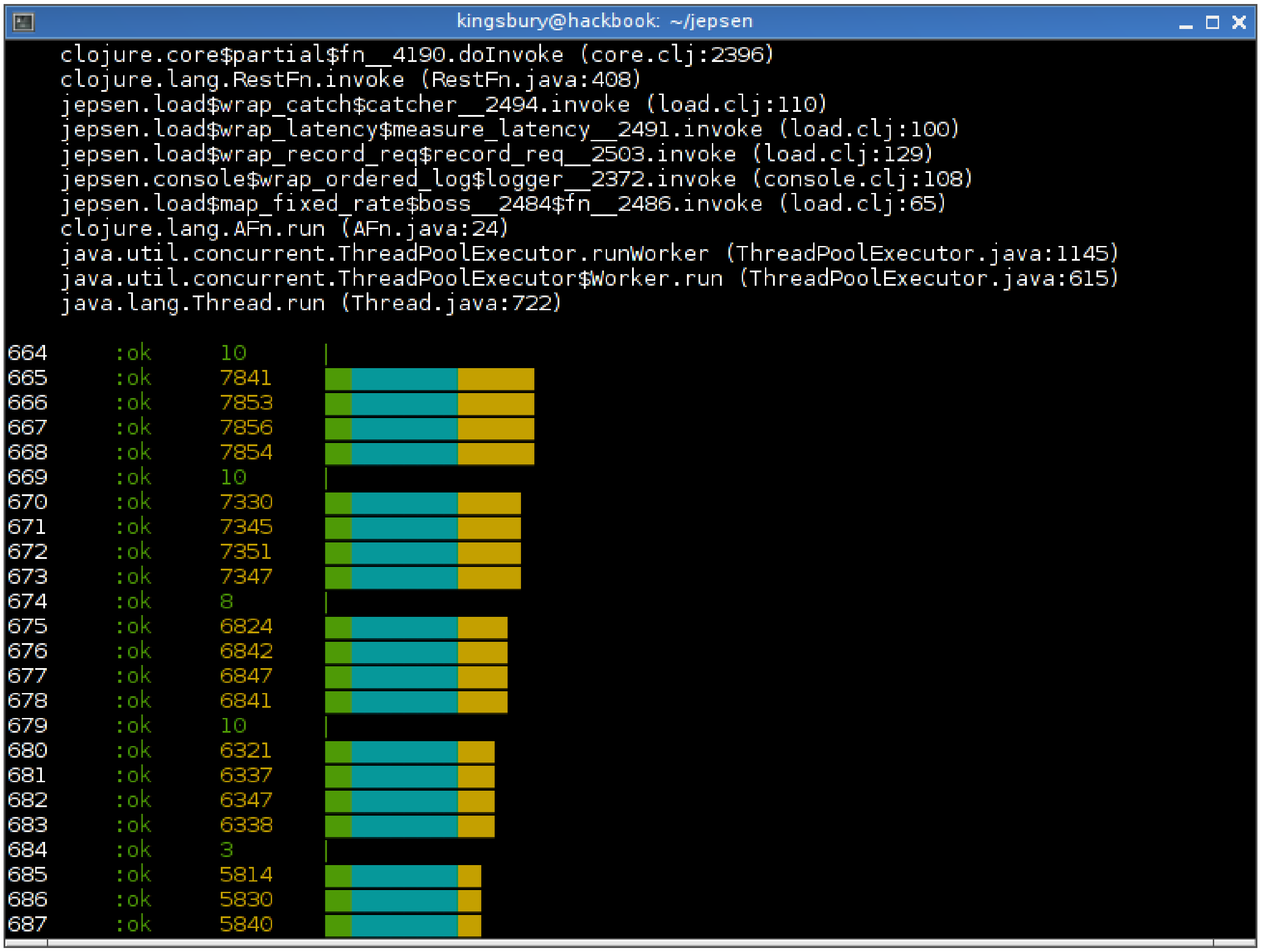
```
124     :error  10080
Failed to send messages after 3 tries.
kafka.common.FailedToSendMessageException: Failed to send messages after 3 tries.
 at kafka.producer.async.DefaultEventHandler.handle (DefaultEventHandler.scala:90)
    kafka.producer.Producer.send (Producer.scala:74)
    kafka.javaapi.producer.Producer.send (Producer.scala:32)
    clj_kafka.producer$send_message.invoke (producer.clj:19)
    jepsen.kafka$app$reify__3240.add (kafka.clj:64)
    jepsen.set_app$eval2525$fn__2548$G__2515__2551.invoke (set_app.clj:12)
    jepsen.set_app$eval2525$fn__2548$G__2514__2555.invoke (set_app.clj:12)
    clojure.lang.AFn.applyToHelper (AFn.java:163)
    clojure.lang.AFn.applyTo (AFn.java:151)
    clojure.core$apply.invoke (core.clj:619)
    clojure.core$partial$fn__4190.doInvoke (core.clj:2396)
    clojure.lang.RestFn.invoke (RestFn.java:408)
    jepsen.load$wrap_catch$catcher__2494.invoke (load.clj:110)
    jepsen.load$wrap_latency$measure_latency__2491.invoke (load.clj:100)
    jepsen.load$wrap_record_req$record_req__2503.invoke (load.clj:129)
    jepsen.console$wrap_ordered_log$logger__2372.invoke (console.clj:108)
    jepsen.load$map_fixed_rate$boss__2484$fn__2486.invoke (load.clj:65)
    clojure.lang.AFn.run (AFn.java:24)
    java.util.concurrent.ThreadPoolExecutor.runWorker (ThreadPoolExecutor.java:1145)
    java.util.concurrent.ThreadPoolExecutor$Worker.run (ThreadPoolExecutor.java:615)
    java.lang.Thread.run (Thread.java:722)

125     :ok     14668
126     :ok     14658
127     :ok     14659
128     :ok     14671
129     :ok     14662
130     :ok     14166
131     :ok     14157
132     :ok     14157
133     :ok     14165
134     :ok     14161
135     :ok     13660
136     :ok     13655
137     :ok     13657
138     :ok     13659
139     :ok     13655
140     :ok     13163
141     :ok     13152
142     :ok     13154
143     :ok     13152
144     :ok     13155
145     :ok     12660
146     :ok     12650
147     :ok     12654
148     :ok     12650
149     :ok     12651
150     :ok     12156
151     :ok     12148
152     :ok     12150
153     :ok     12151
154     :ok     12152
155     :ok     11653
156     :ok     11646
157     :ok     11648
158     :ok     11647
159     :ok     11650
```

We'll allow that leader to acknowledge writes independently, for a time. While these writes *look* fine, they're actually only durable on a single node–and could be lost if a leader election occurs.



```
445     :ok     4       |
446     :ok     4       |
447     :ok     5       |
448     :ok     5       |
449     :ok     5       |
450     :ok     4       |
451     :ok     5       |
452     :ok     2       |
453     :ok     5       |
454     :ok     3       |
455     :ok     5       |
456     :ok     4       |
457     :ok     5       |
458     :ok     5       |
459     :ok     4       |
460     :ok     6       |
461     :ok     5       |
462     :ok     6       |
463     :ok     5       |
464     :ok     5       |
465     :ok     5       |
466     :ok     3       |
467     :ok     4       |
468     :ok     5       |
469     :ok     4       |
470     :ok     5       |
471     :ok     4       |
472     :ok     5       |
473     :ok     7       |
474     :ok     6       |
475     :ok     5       |
476     :ok     4       |
477     :ok     5       |
478     :ok     6       |
479     :ok     5       |
480     :ok     5       |
```

Then we totally partition the leader. ZK detects the leader’s disconnection and the remaining nodes will promote a new leader, causing data loss. Again, a brief latency spike:



```
kingsbury@hackbook: ~/jepsen
    clojure.core$partial$fn__4190.doInvoke (core.clj:2396)
    clojure.lang.RestFn.invoke (RestFn.java:408)
    jepsen.load$wrap_catch$catcher__2494.invoke (load.clj:110)
    jepsen.load$wrap_latency$measure_latency__2491.invoke (load.clj:100)
    jepsen.load$wrap_record_req$record_req__2503.invoke (load.clj:129)
    jepsen.console$wrap_ordered_log$logger__2372.invoke (console.clj:108)
    jepsen.load$map_fixed_rate$boss__2484$fn__2486.invoke (load.clj:65)
    clojure.lang.AFn.run (AFn.java:24)
    java.util.concurrent.ThreadPoolExecutor.runWorker (ThreadPoolExecutor.java:1145)
    java.util.concurrent.ThreadPoolExecutor$Worker.run (ThreadPoolExecutor.java:615)
    java.lang.Thread.run (Thread.java:722)

664     :ok     10      |
665     :ok     7841
666     :ok     7853
667     :ok     7856
668     :ok     7854
669     :ok     10      |
670     :ok     7330
671     :ok     7345
672     :ok     7351
673     :ok     7347
674     :ok     8       |
675     :ok     6824
676     :ok     6842
677     :ok     6847
678     :ok     6841
679     :ok     10      |
680     :ok     6321
681     :ok     6337
682     :ok     6347
683     :ok     6338
684     :ok     3       |
685     :ok     5814
686     :ok     5830
687     :ok     5840
```

At the end of the run, Kafka typically acknowledges 98–100% of writes. However, *half* of those writes (all those made during the partition) are lost.

```
Writes completed in 100.023 seconds

1000 total
987 acknowledged
468 survivors
520 acknowledged writes lost! (╯°□°)╯︵ ┻━┻
130 131 132 133 134 135 ... 644 645 646 647 648 649
1 unacknowledged writes found! ヽ(´ー｀)ノ
(126)
0.987 ack rate
0.52684903 loss rate
0.0010131713 unacknowledged but successful rate
```

Discussion

Kafka's replication claimed to be CA, but in the presence of a partition, threw away an arbitrarily large volume of committed writes. It claimed tolerance to F-1 failures, but a single node could cause catastrophe. How could we improve the algorithm?

All redundant systems have a breaking point. If you lose all N nodes in a system which writes to N nodes synchronously, it'll lose data. If you lose 1 node in a system which writes to 1 node synchronously, that'll lose data too. There's a tradeoff to be made between how many nodes are required for a write, and the number of faults which cause data loss. That's why many systems offer per-request settings for durability. But what choice is *optimal*, in general? If we wanted to preserve the all-nodes-in-the-ISR model, could we constrain the ISR in a way which is most highly available?

It turns out there *is* a maximally available number. From Peleg and Wool's overview paper on quorum consensus:

> It is shown that in a complete network the optimal availability quorum system is the majority (Maj) coterie if p < 1/2.

In particular, given uniformly distributed element failure probabilities smaller than 1/2 (which realistically describes most homogenous clusters), the *worst* quorum systems are the Single coterie (one failure causes unavailability), and the *best* quorum system is the simple Majority (provided the cohort size is small). Because Kafka keeps only a small number (on the order of 1-10) replicas, Majority quorums are provably optimal in their availability characteristics.

You can reason about this from extreme cases: if we allow the ISR to shrink to 1 node, the probability of a single additional failure causing data loss is high. If we require the ISR include *all* nodes, any node failure will make the system unavailable for writes. If we assume failures are partially independent, the probability of two failures goes like 1 - (1-p)^2, which is *much* smaller than p. This superlinear failure probability at both ends is why bounding the ISR size in the *middle* has the lowest probability of failure.

I made two recommendations to the Kafka team:

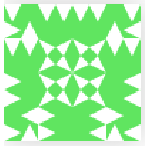
1. Ensure that the ISR never goes below N/2 nodes. This reduces the probability of a single node failure causing the loss of commited writes.
2. In the event that the ISR becomes empty, *block and sound an alarm* instead of silently dropping data. It's OK to make this configurable, but as an administrator, you probably want to be aware when a datastore is about to violate one of its constraints–and make the decision yourself. It might be better to wait until an old

leader can be recovered. Or perhaps the administrator would like a dump of the to-be-dropped writes which could be merged back into the new state of the cluster.

Finally, remember that this is pre-release software; we're discussing a *candidate* design, not a finished product. Jay Kreps and I discussed the possibility of a "stronger safety" mode which does bound the ISR and halts when it becomes empty–if that mode makes it into the next release, and strong safety is important for your use case, check that it is enabled.

Remember, Jun Rao, Jay Kreps, Neha Narkhede, and the rest of the Kafka team are seasoned distributed systems experts–they're much better at this sort of thing than I am. They're also contending with nontrivial performance and fault-tolerance constraints at LinkedIn–and those constraints shape the design space of Kafka in ways I can't fully understand. I trust that they've thought about this problem extensively, and will make the right tradeoffs for their (and hopefully everyone's) use case. Kafka is still a phenomenal persistent messaging system, and I expect it will only get better.

The next post in the Jepsen series explores Cassandra, an AP datastore based on the Dynamo model.



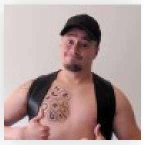
Andrew on 2013-09-25

> Kafka holds a new election and promotes any remaining node–which could be arbitrarily far behind the original leader. That node begins accepting requests and replicating them to the new ISR.

This seems to violate the whole concept of the ISR set, no? It'd be great if your recommendation #2. makes it in to 0.8. I'd much rather get pages about produce errors rather than have to figure out how to clean up inconsistent partitions.

Aphyr on 2013-09-26

Jay Kreps has written a great follow-up post with more details.



Alex Faraone on 2015-01-21

Just idly wondering if there has been any followup on the two recommendations that you put forward in this article since it was written. I'm considering using kafka, and weighing the pros and cons of trying to do so in a cloud scenario (which makes “P” very important in the CAP equation).

Just curious if you have revisited this in the last year or so.

Omid Aladini on 2015-02-04

Since 0.8.2 it's possible to disable unclean leader election: https://issues.apache.org/jira/browse/KAFKA-1028

Vlad Dragos on 2015-05-14

Hello,

Can you please do a follow-up and redo the tests using Kafka 0.8.X.?

Best regards.

James Baiera on 2015-09-06

Great article, as usual. Just finished up Jay's response article that he published. He does make an interesting point here: There's both a Zookeeper quorum and Kafka quorum in play.

It seems that the nature of the failure mode you're demonstrating here is that the ISR's replicated nodes are unreachable from the leader, but those replicas are still part of some group in Zookeeper that are allowed to partake in leader elections. Perhaps it makes more sense that as soon as the ISR leader decides that some following node is unreachable it should notify ZK that the lost follower should not be allowed to run for election.

This could also prove to notify the existing leader if itself is the one that's been ‘lost’. If it's capable of requesting the removal of that node from the group in ZK, even if ZK is partitioned, it can only succeed in that request if it's on the majority side of ZK's partition. If ZK responds that it's in read only mode from partition, then the leader knows up front that it is the odd man out (even if it can still call most of it's ISRs) and yields accordingly.

At that point, if the leader is value and truly alone and on the majority side of ZK's quorum, leadership becomes entirely dependent on whether or not the link with Zookeeper remains open (should ZK partition further or the partition shifts the majority), but should those quorums be split across multiple partitions, well, then it comes back to Jay's point about the difference of being incorrect and alive vs correct and dead.

Ultimately that question being asked just further proves the confusion for classifying it as a "CA" system in the first place…

anonymous on 2015-12-22

The link to the test source is broken. I also can't find any history for it in the parent directory at the github.

Soumen Sarkar on 2016-02-23

Hi Kyle,

Kafka 0.9 has improved reliability by getting away from ZooKeeper as a data store (anti-pattern) and using ZooKeeper for coordination. I am interested to know your plans to retest Kafka.

Thanks and regards, Soumen

Eberhard Wolff on 2016-10-23

The problem seems to be solved. http://kafka.apache.org/documentation.html#design_ha says you can choose a minimum size of an ISR. If it shrinks below that threshold, the system becomes unavailable for writes. Or you can disable unclean leader election. If there is one node left, the system becomes unavailable until the former leader is available again.

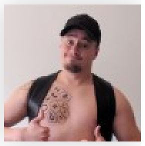
Philipp Krüger on 2017-02-21

Thanks for the educating article and the insights. Minor detail: your computations are off, if I'm not mistaken:

1000 total and 520 lost should be a loss rate of 0,52 right? Accordingly, 1 unacknowledged write should mean 0,001 unacknowledged but successful rate.

I tried calculating your results using various off by 1 errors but I failed. No idea how you get those numbers. Am I missing something?

Aphyr on 2017-02-21



> 1000 total and 520 lost should be a loss rate of 0,52 right?

We only consider records "lost" if they were acknowledged as successful, and not all 1000 attempted writes were acknowledged.

Kevin DeGraaf on 2017-04-24

I have a working docker setup and jepsen project at https://github.com/gator1/jepsen/tree/master/kafka that tests Kafka 0.10.2.0. The problems identified in Kyle's original posts still hold true. I tested using a replication-factor of 3 and 5 partitions. There were a couple of new configuration settings that were added to address those original issues. In particular setting unclean.leader.election.enable=false, and setting min.insync.replicas (I tried 3). Although setting those properties significantly reduced the chance of failures, they still can occur. In particular checker/total-queue would fail occasionally due to successfully acked enqueues getting lost and not finding that value in subsequent history in any dequeues. If anyone has ideas on why this is still a problem, I would be interested in hearing from them. Thanks,

Kevin.

Srdan Srepfler on 2017-07-03

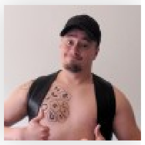
With articles like “Confluent achieves Holy Grail of ‘exactly once’ delivery on Kafka messaging service” and the annoucement of Kafka 0.11.0 https://www.confluent.io/blog/exactly-once-semantics-are-possible-heres-how-apache-kafka-does-it/ I wonder if there’s space to update this article with Part deux to see if Jepsen can debunk Confluent’s claims?

Steven on 2023-02-22

Is it possible that we can get a new review of kafka? They’ve removed Zookeeper from the mix.

Aphyr on 2023-02-28

I’d love to! Ask the Confluent folks about it. :-)



Post a Comment

Comments are moderated. Links have `nofollow`. Seriously, spammers, give it a rest.

Name

E-Mail (for Gravatar, not published)

Personal URL

Comment

Supports Github-flavored Markdown, including `[links](http://foo.com/)`, `*emphasis*`, `_underline_`, `` `code` ``, and `> blockquotes`. Use ```` ```clj ```` on its own line to start an (e.g.) Clojure code block, and ```` ``` ```` to end the block.

Post Comment