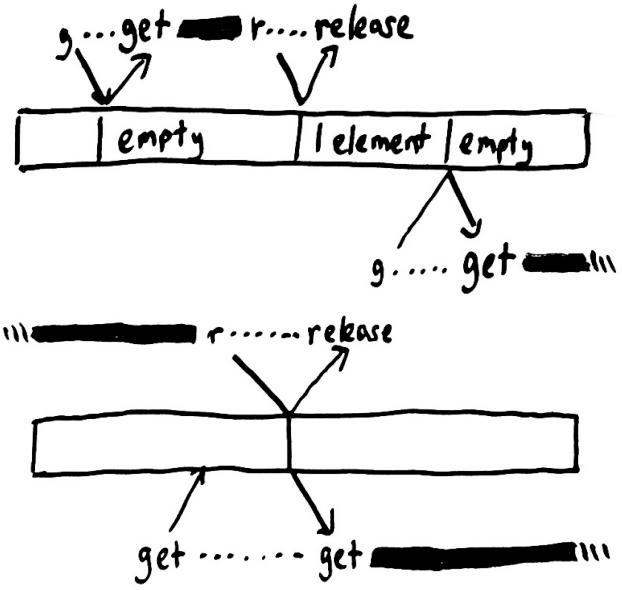
Call me maybe: RabbitMQ

2014/06/06
Distributed Systems Jepsen Networks RabbitMQ

RabbitMQ is a distributed message queue, and is probably the most popular open-source implementation of the AMQP messaging protocol. It supports a wealth of durability, routing, and fanout strategies, and combines excellent documentation with well-designed protocol extensions. I'd like to set all these wonderful properties aside for a few minutes, however, to talk about using your queue as a lock service. After that, we'll explore RabbitMQ's use as a distributed fault-tolerant queue.

Rabbit as a lock service

While I was working on building Knossos–Jepsen's linearizability checker–a RabbitMQ blog post made the rounds of various news aggregators. In this post, the RabbitMQ team showed how one could turn RabbitMQ into a distributed mutex or semaphore service. I thought this was a little bit suspicious, because the RabbitMQ documentation is very clear that partitions invalidate essentially all Rabbit guarantees, but let's go with it for a minute.



RabbitMQ provides a feature whereby a crashed consumer–or one that declares it was unable to process a message–may *return* the message to the queue with a *negative-ack*. The message will then be redelivered to some other consumer later. We can use a queue containing a single message as a shared mutex–a lock that can only be held by a single consumer process at a time.

To acquire the lock, we attempt to consume the message from the queue. When we get the message, we hold the lock. When we wish to release the lock, we issue a negative-ack for the message, and Rabbit puts it back in the queue. It may be some time before another process comes to get the message from the queue, but even in the limiting case, where another process is waiting immediately, a linearizable queue guarantees the safety of this mutex. In order to successfully acquire the mutex, the mutex has to be free in the queue. In order to be free, the mutex *had* to have been released by another process, and if released, that process has already agreed to give up the lock–so at no point can two processes hold the mutex at the same time.

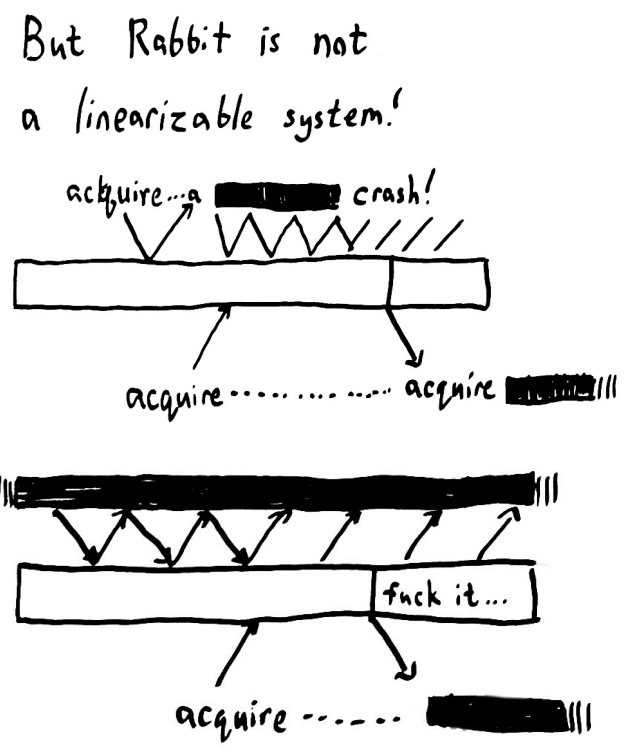
RabbitMQ, however, is *not* a linearizable queue.

It *can't* be linearizable, because as a queue, Rabbit needs to be tolerant of client failures. Imagine that a client process crashes. That process is *never* going to send Rabbit a negative-ack message. Rabbit has to *infer* the process has crashed because it fails to respond to heartbeat messages, or because the TCP connection drops, and, when it decides the process has crashed, it re-enqueues the message for delivery to a second process.

This is still a *safe* mutex, because a truly crashed process can't *do* anything, so it has essentially given up the lock. RabbitMQ can safely hand the message to another process, allowing that process to recover the lock.

But then an ugly thought occurs: in an asynchronous network, reliable failure detectors are really darn hard. RabbitMQ can't tell the difference between a client that's *crashed*, and one that's simply *unresponsive*. Perhaps the network failed, or perhaps the node is undergoing a GC pause, or the VM hiccuped, or the thread servicing RabbitMQ crashed but the thread using the mutex is still running, etc etc etc.

When this happens, Rabbit assumes the process crashed, and sensibly re-enqueues the message for another process to consume. Now our mutex is held by *two* processes concurrently. It is, in short, no longer mutually exclusive.



A demonstration

I've written a basic mutex client using this technique in Jepsen. We'll invoke this client with a stream of alternating `:acquire` and `:release` operations, like so:

```
(deftest mutex-test
  (let [test (run!
              (assoc
                noop-test
                :name      "rabbitmq-mutex"
                :os        debian/os
                :db        db
                :client    (mutex)
                :checker   (checker/compose {:html   timeline/html
                                             :linear checker/linearizable})
                :model     (model/mutex)
                :nemesis   (nemesis/partition-random-halves)
                :generator (gen/phases
                            (->> (gen/seq
                                  (cycle [{:type :invoke :f :acquire}
                                          {:type :invoke :f :release}]))
                                 gen/each
                                 (gen/delay 180)
                                 (gen/nemesis
                                   (gen/seq
                                     (cycle [(gen/sleep 5)
                                             {:type :info :f :start}
                                             (gen/sleep 100)
                                             {:type :info :f :stop}])))
                                 (gen/time-limit 500))))))]
    (is (:valid? (:results test)))
    (report/linearizability (:linear (:results test)))))
```

We're starting out with a basic test skeleton called `noop-test`, and merging in a slew of options that tell Jepsen how to set up, run, and validate the test. We'll use the DB and client from RabbitMQ's namespace. Then we'll *check* the results using both an HTML visualization and Jepsen's linearizability checker, powered by Knossos. Our *model* for this system is a simple mutex

```
(defrecord Mutex [locked?]
  Model
  (step [r op]
```

```
    (condp = (:f op)
      :acquire (if locked?
                 (inconsistent "already held")
                 (Mutex. true))
      :release (if locked?
                 (Mutex. false)
                 (inconsistent "not held")))))
```

… which can be acquired and released, but never double-acquired or double-released. To create failures we'll use the `partition-random-halves` nemesis: a special Jepsen client that cuts the network into randomly selected halves. Then we give a generator: a monadic structure that determines what operations are emitted and when.

```
                :generator (->> (gen/seq
                                 (cycle [{:type :invoke :f :acquire}
                                         {:type :invoke :f :release}]))
                                gen/each
                                (gen/delay 180)
                                (gen/nemesis
                                 (gen/seq
                                  (cycle [(gen/sleep 5)
                                          {:type :info :f :start}
                                          (gen/sleep 100)
                                          {:type :info :f :stop}])))
                                (gen/time-limit 500))))]
```

We start with an infinite sequence of alternating `:acquire` and `:release` operations, wrap it in a generator using `gen/seq`, then scope that generator to each client independently using `gen/each`. We make each client wait for 180 seconds before the next operation by using `gen/delay`, to simulate holding the lock for some time.

Meanwhile, on the nemesis client, we cycle through four operations: sleeping for five seconds, emitting a `:start` operation, sleeping for 100 seconds, and emitting a `:stop`. This creates a random network partition that lasts a hundred seconds, resolves it for five seconds, then creates a new partition.

Finally, we limit the entire test to 500 seconds. During the test, we'll see partition messages in the RabbitMQ logs, like

```
=ERROR REPORT==== 10-Apr-2014::13:16:08 ===
** Node rabbit@n3 not responding **
** Removing (timedout) connection **

=INFO REPORT==== 10-Apr-2014::13:16:29 ===
rabbit on node rabbit@n5 down

=ERROR REPORT==== 10-Apr-2014::13:16:45 ===
Mnesia(rabbit@n1): ** ERROR ** mnesia_event got {inconsistent_database,
running_partitioned_network, rabbit@n3}
```

When the test completes, Jepsen tells us that the mutex failed to linearize. The *linearizable prefix* is the part of the history where Knossos was able to find *some* valid linearization. The first line tells us that the `:nemesis` process reported an `:info` message with function `:start` (and no value). A few lines later, process `1` `:invoke`s the `:acquire` function, concurrently with the other four processes. Most of those processes result in a `:fail` op, but process `1` sucessfully acquires the lock with an `:ok`.

Note that Knossos *fills in* the values for invocations with the known values for their completions, which is why some invocations–like the failed acquire attempts near the end of the history–have values "from the future".

```
Not linearizable. Linearizable prefix was:
:nemesis  :info   :start    nil
:nemesis  :info   :start    "partitioned into [(:n4 :n5) (:n1 :n3 :n2)]"
:nemesis  :info   :stop     nil
:nemesis  :info   :stop     "fully connected"
```

```
:nemesis  :info   :start    nil
:nemesis  :info   :start    "partitioned into [(:n1 :n5) (:n4 :n2 :n3)]"
1         :invoke :acquire  1
3         :invoke :acquire  nil
0         :invoke :acquire  nil
2         :invoke :acquire  nil
4         :invoke :acquire  nil
3         :fail   :acquire  nil
2         :fail   :acquire  nil
1         :ok     :acquire  1
:nemesis  :info   :stop     nil
4         :info   :acquire  "indeterminate: channel error; reason:
{#method<channel.close>(reply-code=404, reply-text=NOT_FOUND - home node 'rabbit@n2' of
durable queue 'jepsen.semaphore' in vhost '/' is down or inaccessible, class-id=60,
method-id=70), null, \"\"}"
:nemesis  :info   :stop     "fully connected"
0 :info :acquire  "indeterminate: channel error; reason: {#method<channel.close>(reply-
code=404, reply-text=NOT_FOUND - home node 'rabbit@n2' of durable queue
'jepsen.semaphore' in vhost '/' is down or inaccessible, class-id=60, method-id=70),
null, \"\"}"
:nemesis  :info   :start    nil
:nemesis  :info   :start    "partitioned into [(:n2 :n4) (:n1 :n5 :n3)]"
:nemesis  :info   :stop     nil
:nemesis  :info   :stop     "fully connected"
:nemesis  :info   :start    nil
:nemesis  :info   :start    "partitioned into [(:n3 :n2) (:n1 :n5 :n4)]"
3         :invoke :release  :not-held
2         :invoke :release  :not-held
3         :fail   :release  :not-held
2         :fail   :release  :not-held
1         :invoke :release  nil
1         :ok     :release  nil
9         :invoke :acquire  "clean connection shutdown; reason: Attempt to use closed
channel"
9         :fail   :acquire  "clean connection shutdown; reason: Attempt to use closed
channel"
5         :invoke :acquire  "clean connection shutdown; reason: Attempt to use closed
channel"
5         :fail   :acquire  "clean connection shutdown; reason: Attempt to use closed
channel"
:nemesis  :info   :stop     nil
:nemesis  :info   :stop     "fully connected"
:nemesis  :info   :start    nil
:nemesis  :info   :start    "partitioned into [(:n1 :n4) (:n3 :n5 :n2)]"
3         :invoke :acquire  nil
2         :invoke :acquire  1
2         :ok     :acquire  1
1         :invoke :acquire  2
```

Next, Jepsen tells us which operation invalidated every consistent interpretation of the history.

```
Followed by inconsistent operation:
1        :ok      :acquire  2
```

Process `2` successfully acquired the lock in the final lines of the linearizable prefix. It was OK for process `1` to *try* to acquire the lock, so long as that invocation didn't go through until process `2` released it. However, process `1`'s acquisition succeeded *before* the lock was released! Jepsen can show us the possible states of the system just prior to that moment:

```
Last consistent worlds were: ----------------

World from fixed history:
1 :invoke :acquire  1
1 :invoke :release  nil
2 :invoke :acquire  1
and current state  #jepsen.model.Mutex{:locked true}
with pending operations:
3 :invoke :acquire  nil
1 :invoke :acquire  2
0 :invoke :acquire  nil
4 :invoke :acquire  nil
-----------------------------------------
```

In this world, the lock was acquired by `1`, released by `1`, and acquired by `2`. That led to the state `Mutex{:locked true}`, with four outstanding acquire ops in progress. But when process `1` successfully acquired the lock, we were forced to eliminate this possibility. Why? Because

```
Inconsistent state transitions:
([{:locked true} "already held"])
```

The state `{:locked true}` couldn't be followed by a second `:acquire` op, because the lock was *already held*.

This shouldn't be surprising: RabbitMQ is designed to ensure message delivery, and its recovery semantics require that it deliver messages more than once. This is a *good property* for a queue! It's just not the right fit for a lock service.

Thinking a little deeper, FLP and Two-Generals suggests that in the presence of a faulty process or network, the queue and the consumer *can't* agree on whether or not to consume a given message. Acknowledge before processing the message, and a crash can cause data loss. Acknowledge after processing, and a crash can cause duplicate delivery. *No* distributed queue can offer exactly-once delivery–the best they can do is at-least-once or at-most-once.

So, the question becomes: does RabbitMQ offer at-least-once delivery in the presence of network partitions?

Rabbit as a queue

For this test, we'll use a different kind of client one that takes `:enqueue` and `:dequeue` operations, and applies them to a RabbitMQ queue. We'll be using durable, triple-mirrored writes across our five-node cluster, with the publisher confirms extension enabled so we only consider messages successful once acked by RabbitMQ. Here's the generator for this test case:

```
              :generator  (gen/phases
                           (->> (gen/queue)
                                (gen/delay 1/10)
                                (gen/nemesis
                                  (gen/seq
                                    (cycle [(gen/sleep 60)
                                            {:type :info :f :start}
                                            (gen/sleep 60)
                                            {:type :info :f :stop}])))
                                (gen/time-limit 360))
                           (gen/nemesis
                             (gen/once {:type :info, :f :stop}))
                           (gen/log "waiting for recovery")
                           (gen/sleep 60)
                           (gen/clients
                             (gen/each
                               (gen/once {:type :invoke
                                          :f    :drain})))))]
```

This test proceeds in several distinct *phases*: all clients must complete a phase before moving, together, to the next. In the first phase, we generate random `:enqueue` and `:dequeue` requests for sequential integers, with a 10^{th} of a second delay in between ops. Meanwhile, the nemesis cuts the network into random halves every sixty seconds, then repairs it for sixty seconds, and so on. This proceeds for 360 seconds in total.

In the next phase, the nemesis repairs the network. We log a message, then sleep for sixty seconds to allow the cluster to stabilize. I've reduced the Erlang `net_tick` times and timeouts to allow for faster convergence, but the same results hold with the stock configuration and longer failure durations.

Finally, each process issues a single `:drain` operation, which the client uses to generate a *whole series* of `:dequeue` ops, for any remaining messages in the queue. This ensures we should see every enqueued message dequeued at least once.

Rabbit has several distribution mechanisms, so I want to be specific: we're testing *clustering*, not the Federation or Shovel systems. We're aiming for the *safest*, most consistent settings, so let's consult Rabbit's documentation about distribution:

Summary

Federation / Shovel	Clustering
Brokers are logically separate and may have different owners.	A cluster forms a single logical broker.
Brokers can run different versions of RabbitMQ and Erlang.	Nodes must run the same version of RabbitMQ, and frequently Erlang.
Brokers can be connected via unreliable WAN links. Communication is via AMQP (optionally secured by SSL), requiring appropriate users and permissions to be set up.	Brokers must be connected via reliable LAN links. Communication is via Erlang internode messaging, requiring a shared Erlang cookie.
Brokers can be connected in whatever topology you arrange. Links can be one- or two-way.	All nodes connect to all other nodes in both directions.
Chooses Availability and Partition Tolerance from the **CAP theorem**.	Chooses Consistency and Availability (or optionally **Consistency and Partition Tolerance**) from the CAP theorem.
Some exchanges in a broker may be federated while some may be local.	Clustering is all-or-nothing.
A client connecting to any broker can only see queues in that broker.	A client connecting to any node can see queues on all nodes.

The column on the right suggests that we can choose either consistency and availability, or consistency and partition tolerance. We've talked about how sacrificing partition tolerance leads to terrible things happening, so let's opt for consistency and partition tolerance. The CP link points to https://www.rabbitmq.com/partitions.html#cp-mode, which says:

> In pause-minority mode RabbitMQ will automatically pause cluster nodes which determine themselves to be in a minority (i.e. fewer or equal than half the total number of nodes) after seeing other nodes go down. It therefore chooses partition tolerance over availability from the CAP theorem. This ensures that in the event of a network partition, at most the nodes in a single partition will continue to run.

CP mode sounds like the safest option, so we'll enable `pause_minority` in our configuration, and expect to see failures from the minority nodes. Rabbit can't guarantee exactly-once delivery, so it can't *really* be CP, but shutting down the minority component should reduce duplicate deliveries as compared to allowing every node to respond to dequeues. Theoretically every node could accept enqueues during the partition (sacrificing the relative order of delivery), so we'll hope to see that behavior as well.

During the partition, Jepsen shows that processes talking to the majority partition can successfully enqueue and dequeue messages, but processes talking to the minority component fail.

```
INFO  jepsen.util - 15    :invoke :dequeue  nil
INFO  jepsen.util - 15    :ok     :dequeue  2935
INFO  jepsen.util - 2551  :invoke :enqueue  3519
INFO  jepsen.util - 2551  :ok     :enqueue  3519
INFO  jepsen.util - 2501  :invoke :dequeue  nil
```

```
WARN  jepsen.core - Process 2501 indeterminate
java.util.concurrent.ExecutionException: java.io.IOException
  at java.util.concurrent.FutureTask.report(FutureTask.java:122)
  at java.util.concurrent.FutureTask.get(FutureTask.java:202)
        ...
Caused by: com.rabbitmq.client.ShutdownSignalException: channel error; reason:
{#method<channel.close>(reply-code=404, reply-text=NOT_FOUND - home node 'rabbit@n1' of
durable queue 'jepsen.queue' in vhost '/' is down or inaccessible, class-id=60, method-
id=70), null, ""}
```

Process 2501 has *crashed*: it's uncertain whether its enqueue suceeded or not. Because the enqueued message might succeed at some later time, we log an `:info` operation rather than an `:ok` or `:fail`. Because processes are singlethreaded in this formalism, we *abandon* that client and start a new client with a unique process ID. Process 2501 will never appear again in this history–but process 2506 will replace it and carry on performing operations against that node in the cluster.

At the end of the test, Jepsen heals the cluster, waits, and drains the queue:

```
INFO  jepsen.util - 2896  :invoke :drain    nil
INFO  jepsen.util - 2896  :ok     :dequeue  3724
INFO  jepsen.util - 9     :invoke :drain    nil
INFO  jepsen.util - 15    :invoke :drain    nil
INFO  jepsen.util - 9     :ok     :dequeue  3733
INFO  jepsen.util - 15    :ok     :dequeue  3730
INFO  jepsen.util - 273   :invoke :drain    nil
INFO  jepsen.util - 273   :ok     :dequeue  3741
INFO  jepsen.util - 2872  :invoke :drain    nil
INFO  jepsen.util - 2872  :ok     :dequeue  3746
INFO  jepsen.util - 9     :ok     :drain    :exhausted
INFO  jepsen.util - 2896  :ok     :drain    :exhausted
INFO  jepsen.util - 15    :ok     :drain    :exhausted
INFO  jepsen.util - 2872  :ok     :drain    :exhausted
INFO  jepsen.util - 273   :ok     :drain    :exhausted
INFO  jepsen.core - Worker 1 done
INFO  jepsen.core - Worker 4 done
INFO  jepsen.core - Worker 0 done
INFO  jepsen.core - Worker 3 done
INFO  jepsen.core - Worker 2 done
INFO  jepsen.core - Run complete, writing
```

We run two checkers on the resulting history. The first, `:queue`, asserts that every enqueue is balanced by exactly one dequeue. It fails quickly: 487 is dequeued twice in this history, after one node crashes. We *expect* this behavior from a robust queue.

```
FAIL in (rabbit-test) (rabbitmq_test.clj:77)
expected: (:valid? (:results test))
  actual: false
{:valid? false,
 :queue {:valid? false, :error "can't dequeue 487"},
```

The `:total-queue` checker, though, is more forgiving for lossy systems. It tells us about four kinds of operations:

- *OK* messages are enqueued and dequeued successfully.
- *Recovered* messages are where we didn't know if the enqueue was successful, but it was dequeued

nonetheless.

- *Unexpected* messages are those which came out of the queue despite never having been enqueued. This includes cases where the same message is dequeued more than once.
- *Lost* messages were successfully enqueued but never came back out. This is the worst type of failure: the loss of acknowledged writes.

```
:total-queue
{:valid? false,
 :lost
 #{2558 3359 3616 3173 1858 2030 2372 3135 3671 3534 3358 2855 3251
   3429 2615 3248 2479 1976 2430 3478 3693 2388 3174 3484 3638 2813
   3280 2282 2475 3239 2973 1984 3630 2264 2523 2565 2462 3278 3425

   ... lots more lines ...

   3313 3413 3443 2048 3513 2705 3392 3001 2215 3097 3364 3531 2605
   2411 2220 2042 1923 2314 3592 3538 3128 2801 3636 1861 3500 3143
   3276 1991 3343 3656 3233 3611 3244 3717 3314 2922 3404 3708},
 :unexpected
 #{487 497 491 510 493 490 508 504 505 502 495 506 500 496 501 498 507
   494 489 492 503 509 499 488},
 :recovered
 #{519 521 529 510 527 518 512 517 516 515 523 531 525 528 522 1398
   520 524 513 509 511},
 :ok-frac 786/1249,
 :unexpected-frac 8/1249,
 :lost-frac 1312/3747,
 :recovered-frac 7/1249}}
```

The fractions all have the same denominator: attempted enqueues. 2358 of 3747 attempted enqueues were successfully dequeued–which we might expect, given the `pause_minority` mode shut down 2/5 nodes for a good part of the test. We also saw 24 duplicate deliveries, which, again, we expect from a distributed queue, and 28 recovered writes from indeterminate enqueues–we'll take as many of those as we can get.

The real problem is those 1312 lost messages: **RabbitMQ lost ~35% of acknowledged writes.**

You guessed right, folks. When a RabbitMQ node rejoins the cluster, it *wipes* its local state and adopts whatever the current primary node thinks the queue should contain. Are there any constraints on which node takes precedence? As far as I can ascertain, no–you can induce arbitrary data loss by carefully manipulating the order of partitions.

This is not a theoretical problem. I know of at least two RabbitMQ deployments which have hit this in production.

Recommendations

If you use RabbitMQ clustering, I recommend you disable automatic partition handling. `pause_minority` reduces availability but still allows massive data loss. `autoheal` improves availability but still allows for massive data loss. Rabbit advises that you use `ignore` only if your network is "really reliable", but I suspect it is the only mode which offers a chance of preventing the loss of acknowledged messages.

If you use the `ignore` partition mode, Rabbit will allow primary replicas on both sides of a partition, and those nodes will run independently. You can't dequeue messages written to the other node, but at least neither will overwrite the other's state. When the network partition recovers, you can isolate one of the nodes from all clients, drain all of its messages, and enqueue them into a selected primary. Finally, restart that node and it'll pick up the primary's state. Repeat the process for node which was isolated, and you'll have a single authoritative cluster again–albeit with duplicates for each copy of the message on each node.

Alternatively, you can eschew RabbitMQ clustering altogether, and connect all nodes via Federation or the Shovel: an external process that ferries messages from one cluster to another. The RabbitMQ team recommends these for unreliable links, like those between datacenters, but if you're concerned about partitions in your local network, you might choose to deploy them in lieu of clustering. I haven't figured out how to use either of those systems in lieu of clustering yet. The Shovel, in particular, is a single point of failure.

To the RabbitMQ team, I suggest that the `autoheal` and `pause_minority` recover by taking the union of the messages extant on both nodes, rather than blindly destroying all data on one replica. I know this may be a big change to make, given the way Rabbit uses mnesia–but I submit that duplicate delivery and reordering are almost certainly preferable to message loss.

To recap

We used Knossos and Jepsen to prove the obvious: RabbitMQ is not a lock service. That investigation led to a discovery hinted at by the documentation: in the presence of partitions, RabbitMQ clustering will not only deliver duplicate messages, but will also drop huge volumes of acknowledged messages on the floor. This is not a new result, but it may be surprising if you haven't read the docs closely–especially if you interpreted the phrase "chooses Consistency and Partition Tolerance" to mean, well, either of those things.

I'd like to conclude by mentioning that RabbitMQ *does documentation right*. Most of the distributed systems I research say nothing about failure modes–but Rabbit's docs have an entire section devoted to partition tolerance, reliable delivery, and truly comprehensive descriptions of the various failure modes in the API and distribution layer. I wish more distributed systems shared Rabbit's integrity and attention to detail.

This research was made possible by a generous grant from Comcast's open-source fund, and by the invaluable assistance of Alvaro Videla from the RabbitMQ team. I'm also indebted to Michael Klishnin and Alex P for their hard work on the Langohr RabbitMQ client. Thanks for reading!

Next up: etcd and Consul.

Patrick Bourke, on 2014/06/09

"We'll be using durable, triple-mirrored writes"

It looks like you only have mirroring configured to two nodes:

https://github.com/aphyr/jepsen/blob/master/rabbitmq/src/jepsen/system/rabbitmq.clj#L82

which currently reads: "{\"ha-mode\": \"exactly\", "ha-params\": 2

This mirrors a queue between the node that its created on and one other.

Brian, on 2014/06/10

If you're going for maximum reliability wouldn't you want to set the :mandatory flag on publishes?

https://github.com/aphyr/jepsen/blob/master/rabbitmq/src/jepsen/system/rabbitmq.clj#L153

Aphyr, on 2014/06/11

Patrick, Brian, sorry 'bout that! I messed around with a bunch of different tuning parameters while testing, but accidentally committed with these params. Neither prevents this kind of data loss (the queue is still routable on those nodes, so mandatory doesn't help AFAICT, and more mirrors just means more copies to get out of sync), but I've updated the source to bring back these parameter choices. Good catch, thank you! :)

Ed Fine, on 2014/06/11

Thanks for an interesting article. What were the responses of the RabbitMQ team to your suggestions? Do they have an explanation for the anomalies you saw?

Aphyr, on 2014/06/12

Haven't heard an official response yet, Ed, but Rabbit's been aware of these problems for years. I think it's just gonna take time; either redesigning RabbitMQ clustering, or

improving federation/shovel to the point where they can support the same use cases.

jsosic, on 2014/06/16

You suggest union of queues after re-connection of partitions, but wouldn’t that mean for already consumed messages on majority partition to be re-queued from minority partition(s)?

Also, would setting up haproxy in front of RabbitMQs help, in a way that haproxy would always point to a single node while other nodes would be marked as backup. That way clients would always publish to master via haproxy, and if partition occurs, master would go down, hence nothing could be published until haproxy switches over to backup node, which has to be in an active partition (if RabbitMQ cluster is set up via pause_minority).

Jonathan, on 2014/06/25

jsosic, the problem is that masters can be different for every queue, so to which node would you configure your haproxy (or any load balancing mechanism) to point to ? Difficult to maintain this. And if you decide to create your queues on the same node to have the same master, this scales really badly as all the resource management will occur on the same node.

Post a Comment

Name

Email

Http

Supports github-flavored markdown for [links](http://foo.com/), *emphasis*, _underline_, `code`, and > blockquotes. Use ```clj on its own line to start a Clojure code block, and ``` to end the block.

Post Comment