Fault tolerant writer for Yarn Timeline Service v.2 requirements

This is a summary of the requirements, thoughts on possible implementation and list of open questions for a feature request for feature in HBase that allows us to spool mutations if we (temporarily) cannot push these to HBase through a BufferedMutator.
This is intended to get clarity around what behavior we would like to see in which kinds of failure scenarios.

Background

Yarn has a timeline service which captures metrics and lifecycle events in a single in-memory DB. This approach has severe scaling limitations, which as partially addressed in version v1.5. The YARN Application Timeline Service v2 is a re-design to address these concerns. It captures metrics and lifecycle events from running YARN applications and stores them in HBase. For example, the MapReduce counters such as HDFS bytes read & written, the # records processed per mapper, etc.

These metrics are emitted through a timeline writer to HBase. In the current implementation in v2. the timeline writer is implemented as an auxiliary service in the NodeManager.

Most metrics are emitted on a best-effort basis, and only for lifecycle events will we try to flush. Many metrics will come in at a fairly high velocity. For example, a large job with 10K tasks can emit ~100 counters per task and emit the data every minute. On larger installations there can be tens of thousands of jobs in a day.

In order to meet the throughput requirements we use a *BufferedMutator* to write to HBase tables. Only for lifecycle events (and possibly on a periodic timer-basis) will we call flush to enture the data is written through to HBase.

One of the objections we face while implementing this feature in Yarn is what will happen to Yarn applications when HBase is (temporarily) unavailable. While there are other kinds of failure scenarios that we would like to eventually tackle, it seems that the most elegant solution is to have the BufferedMutator spool mutations to a (distributed) filesystem/storage if they cannot be pushed to HBase.

References

Here are the jiras and some presentations with additional background on the Yarn Timeline Service v2 project:

- “YARN Timeline Service v.2: alpha 1” https://issues.apache.org/jira/browse/YARN-2928
- “YARN Timeline Service v.2: alpha 2” https://issues.apache.org/jira/browse/YARN-5355
- “Big data Squared How YARN Timeline Service v2 Unlocks 360 Degree Platform Insights at Scale”: https://www.youtube.com/watch?v=adV-DFa-8us and http://www.slideshare.net/vrushalivc/timeline-service-v2-at-the-hadoop-summit-sj-2016?qid=b77390fa-9093-4c21-9fcf-a234084a0e9f&v=&b=&from_search=1
- “[HBaseCon East 2016] Session 7: Update on the HBase-based Timeline Service V2” https://www.youtube.com/watch?v=RKQlOpCXJgI

Requirements

Even when HBase is down, we want to ensure that Yarn applications do not get blocked during the flush to the timeline writer.

First off, towards the end of the container lifecycle, a stalled HBase write can cause a YARN container to be killed before properly wrapping up. On the ResourceManager, a long stall would cause the entire cluster to become unstable. Writes will have to either succeed or fail within 30 seconds to possibly up to 1 minute maximum.

Events keep coming in, and jobs keep starting and finishing whether HBase is up or not. We want to be able to have HBase be down for upgrades, or temporarily unavailable without bringing all the YARN clusters writing to it down as well.

This feature discusses how we can spool to a filesystem in case HBase is temporarily down, and then later play-back these spooled mutations (puts). Note that on the writer side we have been very careful to have only put mutations. We do not do any reads in the ats writer.

Our puts are timestamped with event time during the initial write attempt so out of these puts can be processed out of order and still yield the correct results as long as they aren’t beyond a certain threshold (initially probably 1 day due to the way compaction works for readless aggregates).

The TimelineServiceV2 readers will be temporarily down when HBase is unavailable, that is not addressed in this particular issue. We are ok to have the reads be stale from the perspective of spooled writes. In other words, if HBase has been down, it is expected that the spooled writes are slightly behind and will catch up. During that time the reads will be somewhat stale.

Failure scenarios

There can be several different error conditions to consider:

1. HBase down for maintenance
2. Split brain: connection to HDFS OK, but cannot connect to HBASE
    a. HBase down
    b. ZK issues
    c. HBase up but network partitioned
3. HBase error. For this category we do not mean intermediate temporary slowness or errors such as an HBase region in transition, or a single node that causes writes to stall for a bit. We'll have to carefully tune the timeout values in ZK with the time that we decide to start spooling (perhaps make the latter a multiple of the former to ensure a consistent setup)
4. LOCAL NIC borked -> cannot write to HBase, nor lookup region location for rows. Spooling to HDFS will also not work, spooling to local filesystem (as backup?) could still work.
5. HBase up, but it's underlying HDFS unavailable
    a. If WAL is already open, writes may continue, if WAL needs to be rolled, then writes will stall
    b. Does region server - DN affinity for block placement work well with HDFS upgrade domains for HDFS upgrades under HBase cluster?
    c. Will rolling restart have a perf impact to the point where it is better to have nodes (temporarily) spool the puts?

Replaying spooled puts

Once a particular error condition is remedied, the spooled puts need to be replayed. Here are some of the considerations
Depending on whether writes are local or not (potentially as a backup for distributed writes) we need to determine who or what replays spooled logs. For the initial design I think we should focus on distributed writes only and initially forget about local writes to simplify things.

1. If puts are spooled to a distributed storage (HDFS, s3, gcs, etc), we could use MapReduce, Tez or similar frameworks to re-play logs This seems to be the simplest way to get to scale and throttling limits, easily centrally managed, yet running distributed. Therefore it seems reasonable to focus on spooling to distributed storage.
2. For locally spooled puts, a local runner needs to replay node local spooled files. Something needs to pick up these logs and replay. We need to ensure they replay through another buffered mutator, but not get write amplification by again spooling to disk in case of failures. In addition,
    a. We need to worry about how to rate limit these spooled writes. Could perhaps use hbase.client.statistics.backoff-policy?
    b. need coordination to ensure only one replay process is running
    c. need liveness check to ensure replay process isn't stuck

    d. What would we do if the machine goes down and never gets started again.
    e. How would this interact with blacklisting of a node in Yarn. Can logs be replayed from a blacklisted node?
3. Central process could de-dupe if needed. Have files named after cluster and sending hostname for namespacing
4. Central MapReduce can deal with dropping stale writes and/or pushing them to archive instead for later recovery. For the coprocessor aggregation we assume writes don't come in later than a certain configured period (probably ~1 day).
5. We can avoid circular dependencies by de-coupling the client writing to distributed storage from MR process to upload.
6. Except for the put serialization which is in a class / package dependant on MR (how to we avoid circular dependencies)

Distributed FS writes need to be limited in size and/or by time. HDFS blocks are available (except for last one until file close). S3 and gcs (?) writes are invisible until the objects are closed. Files first spooled to local disk could be uploaded in parallel, but otherwise these writes themselves are buffered in the respective client.

This creates a trade-off between file size (many small ones) and permanence / persistence. Configurable size limit and time limit (10 minute default?) seems reasonable.
a) Open file and keep writing until done. one file / object per write wouldn't be feasible.
b) Need to consider filesystem flush / hflush/sync on ats writer sync. Implications of s3 (& gcs files invisible) even after flush until close? Configurable sync meaning close semantics?
c) Close file on ats writer close?
d) One central shared writer process, or each writer creates its own files? Locally disk spooled files can be uploaded in batch.
e) Compression and file format configurable? Serialized puts presumably don't need record separators, so configured snappy or lzo compressed file should do.
f) Actual writer should be pluggable so that multiple implementations (i.e. HDFS, s3, gcs, etc) can be provided).

Recovering from errors

Aside from re-playing spooled puts, we need to consider how we get back to a normal situation. Do we let an exception bubble up to the client so that they can re-connect, or do we try to seamlessly handle recovery under the covers. A wrapped BufferedMutator should create a new one, swap out the delegation and call close on spooling one when we go back to online mode. We will likely need state diagram to model state transitions.

Note that BufferedMutator.mutate even in async mode may block on
    a) region server for row in this table lookups

b) waiting for maximum concurrent tasks (configurable, default=100) to clear. So if tasks are al blocked and waiting on HBase (when it has problems) submitting may block. this means we have to do async detection for HBase errors and get the blocked tasks in progress unstuck and/or make them fail fast w/o running into synchronization issues.
c) Potential mechanism for this might be to interrupt the working threads and InterruptedException. TODO: is that a no-no on a ThreadPoolExecutor? TODO: see if DelayingRunner swallowing the InterruptedException w/o resetting the thread interrupt status will mess with this plan.
d) If another task is already submitting to a region maxConcurrentTasksPerRegion defaults to 1, so if a write is currently going on, with our key-structure we're likely to hit the same region server multiple times. Current write will block the next one to same region with maxConcurrentTasksPerRegion = 1 (default)
e) If the server hosting a region (server) exceeds maxConcurrentTasksPerServer (default to 2).

In case of errors, it is through the (configurable!) backoff policy that determines the retry period. There is hope to have a coordinated un-wedge action when we know of HBase issues.

Implementation considerations

Although this document mostly focusses on the requirements and listing out failure scenarios, we did have some discussions with folks at the last HBase conference in NYC and looked through some of the code to see what might be feasible. Here are some rough notes on these discussions and some of our thoughts on implementation.

Would it be appropriate to use o.a.h.hbase.client.ClusterStatusListener (configurable) Potentially hbase.client.statistics.backoff-policy could be used to manipulate backoff policy and to dial down the re-tries and then dial back up if we know that a connection has been made (again).

Proposed change: make ConnectionManager#getBufferedMutator (line 774) configurable instead of new'ing up a BufferedMutatorImpl.

Alternatively make ap = new AsyncProcess(connection, conf, pool, rpcCallerFactory, true, rpcFactory);
configurable. unfortunately o.a.h.hbase.client.AsyncProcess is marked private.

Even if we have control of the Async processor and we detect a server error, there isn't anything we can do to compel the BufferedMutator to spill its guts. We'd have to wait until the BM decides to do this by itself. The TimelineServiceV2 client could do a periodic flush to force this behavior to limit the amount of data pending in memory.

Alternatively we can add this behavior to a modified bufferent mutator that auto-flushes every x seconds, whether currentWriteBufferSize > writeBufferSize or not.

Submitting a mutation to the BufferedMutator and through its AsyncProcessor can immediately cause a lock-up / hang / exception thrown when region lookup fails.
When regions are cached (?) this might not happen, otherwise mutate will hang right there until timeouts.
Submit on the AsyncMutator will certainly have to be modified in "currently spooling mode" to avoid trying to lookup region locations
when we know that isn't possible.

Open questions

1. Do we want to be able to make all writers stop writing to HBase and start spooling through admin command?
    a. need to be able to communicate this status to existing and new clients.
    b. need to be able to jigger clients to re-connect to HBase.
    c. Would we add something to the RegionServer-> client protocol to have a region server indicate that a client should back off, or perhaps even disconnect and start spooling (for example by sending a “IMAboutToGoDown” response) if such a mechanism is possible.
2. How to detect HBase is up again (through admin command or by itself)?
    a. central component doing testing. -> fewer pings, but still suffers from network partitions
    b. distributed detection in clients. More repeated work, but no coordination needed (other than admin commands)

Need to detect flapping HBase connections?
If HBase is up, do we need to avoid thundering herd? Rate limiting is simpler with spooled writes to a distributed storage and centrally managed replay than distributed node-local un-spooling. Perhaps HBase can provide some flow-control mechanism for overloaded region servers, or do we need to build an external mechanism to accomplish this if we were to use node-local spooling?

Should we make TimelineServiceV2 clients configure whether is is worth to re-connect, or just to keep on spooling?
    a) possible with HDFS (or S3 or gcs, or other distributed writes).

Do we reflect the fact that we’re spooling back up to how we handle a flush call to the timeline writer API?

Do we need to override / wrap Connection in order to intercept and un-wedge locateRegion calls?

In spooling mode we can intercept new writes going into the buffered mutator by wrapping the existing impl, however, what makes the BM spill the writes in the buffer (other than calling flush)?

Once a connection is in a "bad" state, can we get it working again, or do we need to create a new connection, toss the old BM and create a new one? That would be possible if we wrap the BM, and discard the old one during spooling mode.

a) does calling close on a BM when we know it is empty and we know HBase isn't reachable going to hang?

What do we do if the spooling filesystem has failures? For example, what if we write to s3 or gcs and an object put fails? What if a HDFS write fails? Do we just give up, ignore the last bits written, keep trying? Perhaps in the first version we just give up if our first back-up fails and later we can consider a more involved strategy of multiple layers of back-up filesystems.