Incremental Cooperative Rebalancing for Streams

TL;DR

This is an addendum of the parent page on reducing rebalance cost, with a focus on Kafka Streams: as of today rebalancing is a costly operation that we need to optimize to achieve faster and more efficient application starts/restarts/shutdowns, failovers, elasticity.

Goals

- At a high-level, we want to **strengthen the operability** of Streams applications.
- To achieve faster and more efficient **application startups / restarts / shutdowns** (notably rolling restarts and rolling upgrades)
  - "More efficient" includes less unnecessary network traffic and load on both app instances and on the backing Kafka clusters.
- Very similar to the previous point, to achieve faster and more efficient **failovers** (e.g. when 1 app instance out of 10 has died and the remaining 9 app instances need to take over work).
  - "Very similar" because, from a rebalancing standpoint, there's not much difference between a planned app instance restart and an unplanned failover event, for example.
- To achieve faster and more efficient **elasticity** (scale in/scale out).

Desired features

We'd like to present the returned value in categories of scenarios. Note that sticky assignment and standby replication would be relevant determining the impact of each scenario.

Also I'm sorting the scenarios by their commonness and user impact (subjective and open for discussion):

1. Application start: when multi-instance application is started, multiple rebalances are required to migrate states to newly started instances. **Standby-replication will not help.**
2. Application shutdown: when multi-instance application is shutting down, multiple rebalances are required. **Standby-replication only slightly remedy this situation.**
3. Application scale out: when a new instance is started, one rebalance is executed to shuffle all assignment. **Standby-replication will not help.**
4. Application scale in: when an existing instance gracefully shutdown, once rebalance is executed to shuffle all assignment. **Standby-replication will largely help in this situation.**
5. Application instance bounce (upgrade, config change etc): one instance shut down and then restart, it will trigger two rebalances. **Standby-replication will largely help in this situation.**
6. Application instance failure: one instance failed, and probably a new instance start to take its assignment, it will trigger two rebalances. The different with 3) above is that new instance would not have local cached tasks. **Standby-replication will not help.**

Proposal

We have two proposals to generally improve the rebalance protocol in Incremental Cooperative Rebalancing: Support and Policies (we consider the "Incremental Imbalance" as a follow-up of "Delayed Imbalance").

1) **Simple** approach tackles on not involving all the partitions in each assignment as it will incurs committing costs; instead it introduces a partiton revokation field in the protocol such that a second join can be triggered to finally move the assignment.

2) **Delayed Imbalance** approach takes one step further on the Simple approach, that it defers (by a configured timeout) the second rebalance to really migrate the partitions; note in Simple the second rebalance to migrate the partitions always happen immediately.

There is a third semi-orthogonal proposal dependent on the Simple approach above:

3) **Standby Bootstrap** approach targeted to reduce new member taking restoration with long latency, by letting the new joining member to be assigned standby tasks only at first, and then when it has bootstraped completed trigger a another join group to move the active task.

I'd like to summarize their values on the above scenarios below compared with what we have today (counting the existing optimizations we have done as of 2017.Q4).

Note again the LOE is my personal estimates:

Approach / LOE	App Start	App Shutdown	App Scale-Up	App Scale-Down	Instance Bounce	Instance Failureover

Current	**MAYBE OK** • KIP-134 would help reduce #. rebalances with right configs	**MAYBE OK** • Disable leave-group would help reduce #. rebalances	**BAD** • Rebalance cannot be saved • New member always needs time to restore • KAFKA-6144 / 6145	**MAYBE OK** With Standby: • Rebalance would be cheap, as we pay the suspension cost for all tasks **BAD** Without Standby: • Without standby rebalance always requires restoration for both non-related tasks and related tasks (assuming it is a complete shuffle)	**MAYBE OK** • Disable leave-group may reduce to one rebalance, but in practice it may less likely • That single rebalance would be cheap with sticky partitionor	**MAYBE OK** With Standby: • Most likely triggers two rebalances • With standby the first rebalance would be cheap, the second rebalance needs restoration **BAD** Without standby: • Most likely triggers two rebalances • Without standby two rebalances would be expensive due to restoration
Simple	**MAYBE BETTER** • Similar to *Current*. • May save task suspension cost but incur more rebalances	**MAYBE BETTER** • Similar to *Current*. • May save task suspension cost but incur more rebalances	**BAD** • Same to *Current*.	**BEST** With standby: • Would be very cheap because all we need is to pick the standby host as the new active host while keeping all other tasks un-touched; hence we can save even the task suspension cost for non related tasks **BETTER** Without standby: • Rebalance always requires restoration for related tasks, although for other tasks we can save suspension cost	**BETER** • Similar to *Current*. • That single rebalance would be even cheaper because we save task suspension cost	**MAYBE BETTER** With Standby: • Similar to *Current*. • May save task suspension cost but incur one more rebalance • With standby the first rebalance would be cheap, the second rebalance would be cheap, the third would require restoration **BAD** Without Standby: • Similar to *Current*. • May save task suspension cost but incur one more rebalanc. • Without standby the first rebalance would require restoration, the second rebalance would be cheap, the third would require restoration
Delayed Imbalance	**BETTER** • Could subsume KIP-134 • Reduce #. rebalances with the right config	**BEST** • Could subsume leave-group disabling. • Could reduce to no heavy rebalance at all with the right config	**BAD** • Same to *Current*.	**BEST** With standby: • Same to *Simple*. **BETTER** Without standby: • Same to *Simple*.	**BEST** • Same to *Simple*.	**BETTER** With Standby: • Compared to *Simple*, The first rebalance would be cheaper, as it would not cause anyone to take over the partition and restore. **BETTER** Without standby: • The first rebalance would be cheap, as it would not cause anyone to take over the partition and restore.

Standby Bootstrap	**MAYBE BETTER** • Same to *Simple.*	**MAYBE BETTER** • Same to *Simple.*	**BEST** • Would require three rebalances, the first one to assign the standby, the second to notify the exising to revoke, and the third to migrate the active task.	**BEST** With standby: • Same to *Simple*. **BEST** Without standby: • Require one more rebalance, but the migrated task would bootstrap via standby first.	**BEST** • Same to *Simple*.	**BETTER** With Standby: • Compared to *Sim ple*, the third rebalance will be shorter as the previous rebalance will make the new member to bootstrap first **STILL NOT GOOD** Without standby: • Compared to *Sim ple*, the third rebalance will be shorter as the previous rebalance will make the new member to bootstrap first • However, the first rebalance would still be expensive due to restoration.
Delayed Imbalance + Standby Bootstrap	**BETTER** • Simple as *Delaye d Imbalance*.	**BEST** • Simple as *Delaye d Imbalance*.	**BEST** • Same as *Standby Bootstrap*.	**BEST** With standby: • Same as *Simple*. **BEST** Without standby: • Same as *Standby Bootstrap*.	**BEST** • Same to *Simple*.	**BEST** With Standby: • Only requires two rebalance, the first is for bootstrap the new member, and the second for assigning the active task. **BEST** Without standby: • Same as above without standby tasks. • Only requires two rebalance, the first is for bootstrap the new member, and the second for assigning the active task.