Heron Stateful Storage API Review

Overview

This document is a review of current status of heron's stateful storage interface as of April 2018. It also contains issues and proposals of what needs to be done next.

The objective is to make the API stable so that logics above (like instances and checkpoint managers) and below (like storage implementations) can proceed independently and are not blocked by each other.

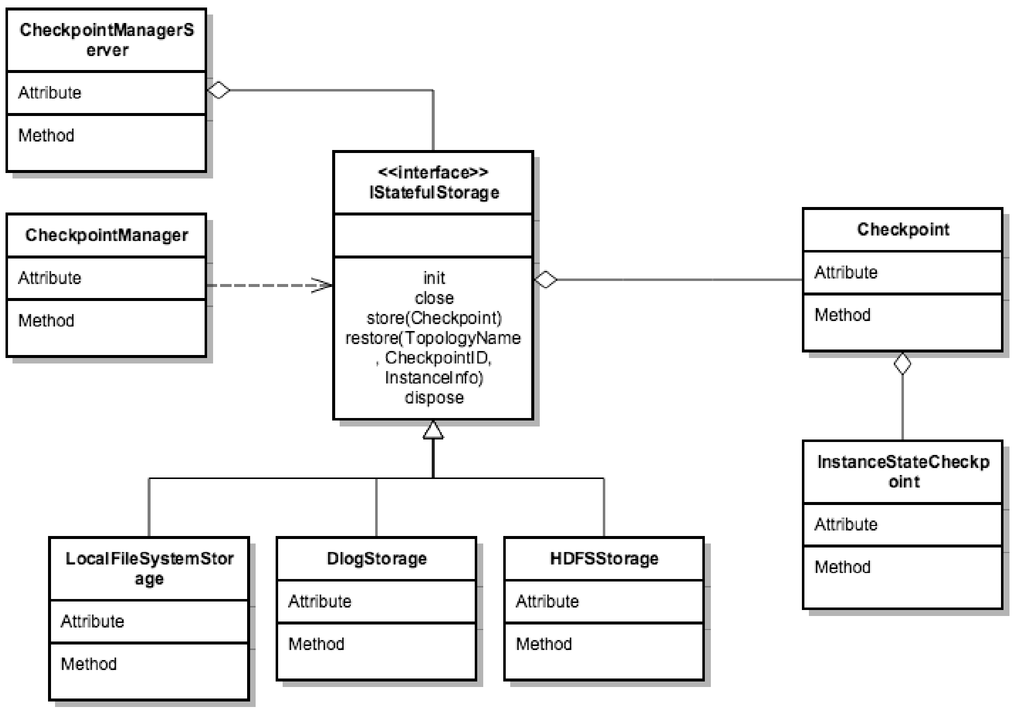
Repartitioning is being considered but not the focus or blocker of this doc.

Current Status

The overall design seems to be complete and reasonable: https://docs.google.com/document/d/1pNuE77diSrYHb7vHPuPO3DZqYdcxrhywH_f7IoVryCI/edit and https://docs.google.com/document/d/1YDFNvLTX6Sg3WDrNFKiWLaJvuEtK4eyxEaA0w9cVIG4/edit#heading=h.d6uy2uxfs2xq

One key point is that **stream manager** is responsible for communicating with instances to create checkpoint data and **checkpoint manager** is responsible for reading and writing stateful storage. Therefore the checkpoint manager is the only component that has storage access. This design hugely simplifies the stateful storage support.

Currently, there is IStatefulStorage interface and its implementations: LocalFileSystemStorage, DlogStorage and HDFSStorage. However not all of them are production ready yet (for example HDFSStorage is not used). (Also checkpoint cleanup logic is not working correctly with local storage and needs to be revisited. It is because tmaster is issuing the cleanup event and send to the checkpoint manager in container 0 which doesn't have access to local storages in other containers)



Issues

Here are few issues or TODOs in the current solution/implementation

- One major issue is that the currently API assumes one data blob for each instance, and the API doesn't have the grouping information. This causes unnecessary dependency between checkpoint manager and stateful storage and it is a big issue for repartitioning and it is a blocker for scale up and down stateful topologies.

- ~~The current kafka spout implementation is using Manhattan to keep kafka offsets. This logic might be refactored to use the general stateful storage interface later. The change should be quite local and there is no structural issues here. We can consider using either Manhattan or HDFS as backend.~~

Considerations

A few things are not ideal in the current structure:

- There is no difference between file-based and service-based storages currently. It could be cleaner to add another layer of abstraction. ~~One option is to provide a layer on top of the file-based storage to support a key-value interface.~~ File based storages normally support appending and random access, while service-based storage normally don't have the features, but some of them may have transaction support on the other hand. We don't see the need for a layer like FileBasedStorage and ServiceBasedStorage so far, but it could be considered to have interfaces like Appendable and RandomAccessible in future.
- instance info (instance id, strmgr id) is used to reference/index Checkpoint data. This design is making stateful storage depend topology logical plan and making repartitioning harder. In theory, it could be cleaner to consider stateful storage purely as a low level key-value store with metadata and the key is independent of any high level data like instance info.
- Currently stateful data doesn't have a version number. Users have to handle it by in there own code. It could be better for heron to provide a field for users to use. But this might be an overkill until the feature is supported. Extra discussion can be found in this PR.

Proposal

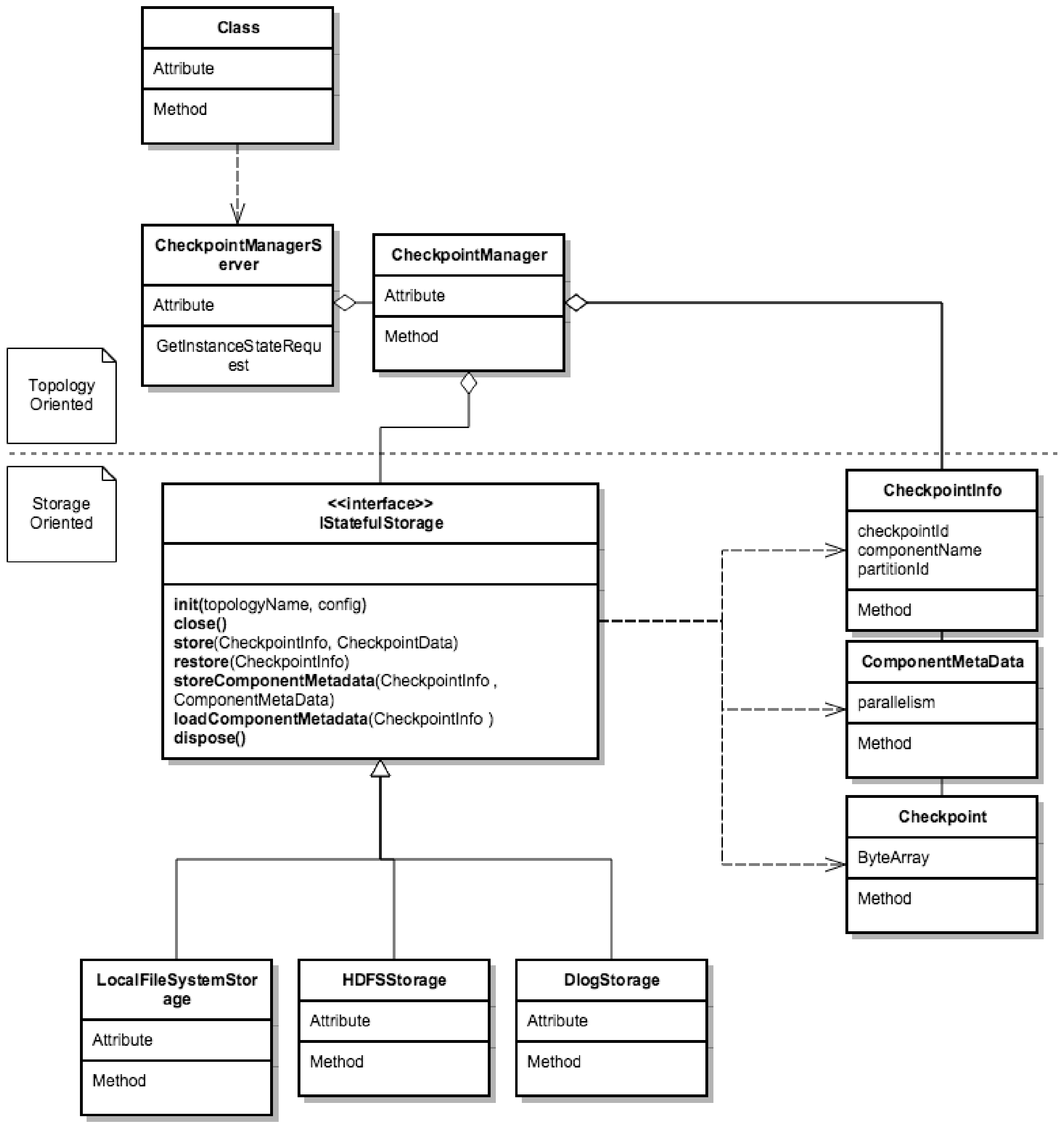
To make StatefulStorage layer more independent, we are going to make the following two changes:

- decouple it with the checkpoint manager layer by creating a new data structure as the key. So the StatefulStorage interface should work more like a key-value storage and it doesn't rely on higher level data. Checkpoint manager should be responsible for translating topology/instance information to storage key.
- Remove/decouple instance information from the key data structure. This could give us more flexibilities when implementing the partitioning logic later. So the data structure should have the following data:

    - Chec
    - Component name
    - Instance index in the component (this is likely to be used by file based storage only and ignored by kv store)
- Checkpoint data is a key-value map of [Partition id -> state data blob], where state data blob is likely to be a key-value map.
- Offset of data blob for each partition needs to stored as well

Furthermore, component level metadata should be added (and the API). It is critical for the repartitioning process to be added later. It should have the following data:

- Parallelism
- TBD



Top Level Tasks

1. Refactor IStatefulStorage and move Instance info out of the interface. Keep code backward compatible
2. ~~Add FileBasedStatefulStorage and ServiceBasedStatefulStorage. Keep code backward compatible.~~
3. Complete HDFSStorage class (basic functionality) and test it. Stateful topologies should work ok at this point.
4. Add in memory cache, file appending support, and other improvements.

About Checkpoint Data

Stateful Storage is used to save/load checkpoint data, so it is necessary to understand how checkpoint data is organized.

Topology Checkpoint -> Component Checkpoint (Component Metadata) -> Instance Checkpoint

Each of them contains the following information

- Topology checkpoint data
    - Last successful checkpoint ID
- Component checkpoint (Component Metadata)
    - Checkpoint ID
    - Component Parallelism (for repartitioning)
- Instance checkpoint
    - Checkpoint ID
    - A single partition, or multiple partitions as a Map of partition ID -> State(key-value pairs as ByteBuffer with serializer)
    - Currently
        - required string checkpoint_id = 1;
        - required bytes state = 2; // Deserialize to State<Serializable, Serializable> and stored to instanceState in Slave.

About Repartition Operation

Repartition operation is critical for stateful storage. There are a few options (https://docs.google.com/document/d/1X0pS9uwevn16nYYqrGjEvWQMWQLwti7Q0hmW6lmgyk8/edit#heading=h.io67iyscchuh) but the two step hashing solution used in other stream processing frameworks seems to be more straightforward.

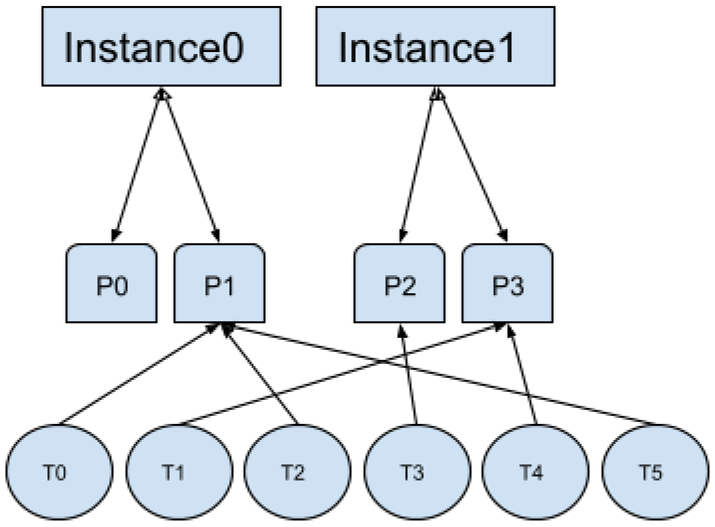
Repartition operation works differently for stateful groupings (field grouping and custom grouping) and stateless groupings(shuffle grouping).

For Stateful Grouping

In order to support repartition (scale up/down) for stateful grouping, another layer of partitioning between tuples and instances is needed and it should be applied in both the grouping calculation and storage operation in a consistent way.

Grouping Calculation

For each stateful component, instead of mapping to instances directly using the grouping logic, tuples need to be mapped to partitions based on the logic and multiple partitions share a deterministic owner instance. Then the partition information and the tuple are passed to the instance together. Each partition has its own state object and instance(stateful bolts) should choose the right state object to update for each tuple.



Basically the grouping logic is used to map each tuple to a virtual key first. Then the key is consistently assigned to an instance. The hashing result doesn't change when number of instance changes as long as the key space is the same. The requirements for the keys are:

- The key space should be big enough so that the per instance traffic is more balanced.
- The key will be the upper bound of the parallelism of the component.
- The key space shouldn't change when a topology starts running, otherwise the checkpoint won't be restored correctly.

The key space

Currently the greatest parallelism in our production environment is a few hundreds but it is possible that there will be bigger topologies in future. Let's assume the max to be 1000 for now. The key space should be a few times of the number of instance to have a better load balance. For example, if the factor is 2, which means there are 2000 partitions. When the number of instance is 900 (it is worse if the number of instance is 1100), instance 0 ~ 199 would own 3 partitions and instance 200~899 would own 2. As the result, if the data is evenly distributed to the partitions, instance 0 ~ 199 would each have (3 - 2) / 2 = 50% more traffic than the rest. If the factor is 10, the traffic difference would be (12 - 11) /11 = 9.1%.

Assuming this factor to be 10, the key space would be 10 * 1000 = 10k. A factor of 100 might be more preferable for big components. Larger key space would be helpful for better load balancing, but could cause more overhead to maintain the map in memory and storage.

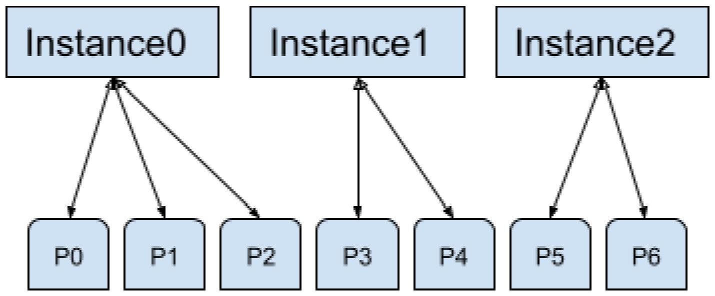
Assuming a tuple has been mapped to a partition, the equations between partition key (or partition id) and instance should be:

```
# number of partitions in each instance (floor)
Per_instance_partition_count = floor(num_of_partition / num_of_instance)
# number of partitions that are not allocated to instances. These partitions will be allocated to the instances at beginning, one overflow partition per instance.
overflow_partition_count = num_of_partition - per_instance_partition_count * num_of_instance
// partitions on the left of first_non_over_flow_partition_id are in fatter instances
first_non_over_flow_partition_id = Extra_partition_count * (per_instance_partition_count + 1)

Target_instance(partition_id) =
  if partition_id < first_non_over_flow_partition_id:
    Partition_id / (Per_instance_partition_count + 1)
  Else:
    (Partition_id - first_non_over_flow_partition_id) / per_instance_partition_count


first_partition_id_for_instance(instance_id) =
  If instance_id < overflow_partition_count:
    Instance_id * (per_instance_partition_count + 1)
  Else:
    First_non_over_flow_partition_id + (instance_id - overflow_partition_count) *
per_instance_partition_count

last_partition_id_for_instance(instance_id) =
  If instance_id < overflow_partition_count:
    first_partition_id_for_instance(instance_id) + per_instance_partition_count
  Else:
    first_partition_id_for_instance(instance_id) + per_instance_partition_count - 1
```



Note that the equations and the numbers of partition and instance need to be consistent across the whole topology.

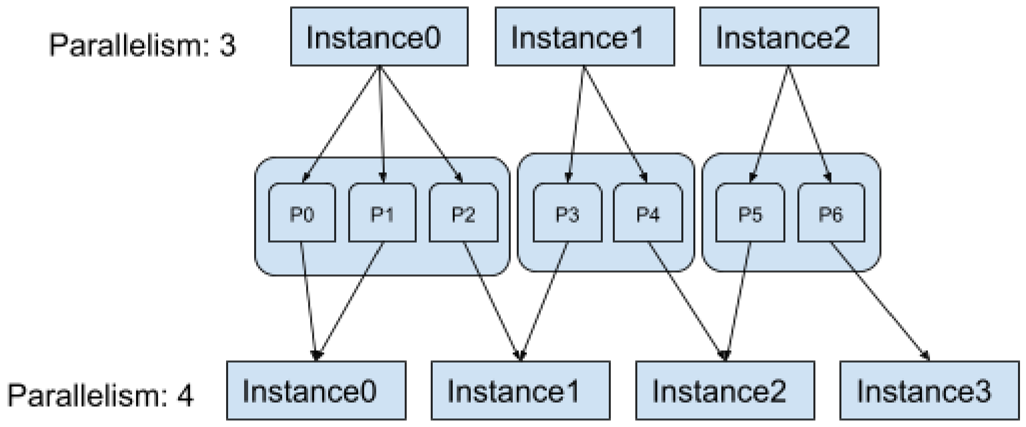
Note that “/” is used in the equations instead of “%” which could be simpler to implement. In this solution continuous partitions are assigned to each instance and it is friendly to stateful storage implementations.

In normal running (no scale up/down), partitions of a single instance could be stored in a single blob for better read and write efficiency. During repartitioning, the number of files to load for a specific instance is a lot less in this way. The data size to be loaded would increase (expecting X2 for scaling up and X2~3 for scaling down when parallelism change is not dramatic. Plus storage implementation can also have the freedom to split blobs transparently by itself if data size is really a concern), but scaling should be rare compare to checkpoint store and restore, and loading data twice is likely to be preferable than one request per partition. Another advantage is that choosing number of partitions is easier because fragmental data is not a concern.

Stateful Storage Operations

The state partitions for a single instance can be stored in one blob instead one blob per partition. This makes store and restore operations more efficient (less requests are needed).

If repartition happens (parallelism changed), stateful storage figures out the first and last partition ids for the instance and uses the old parallelism information to figure out the blobs to load and then pick the partitions.

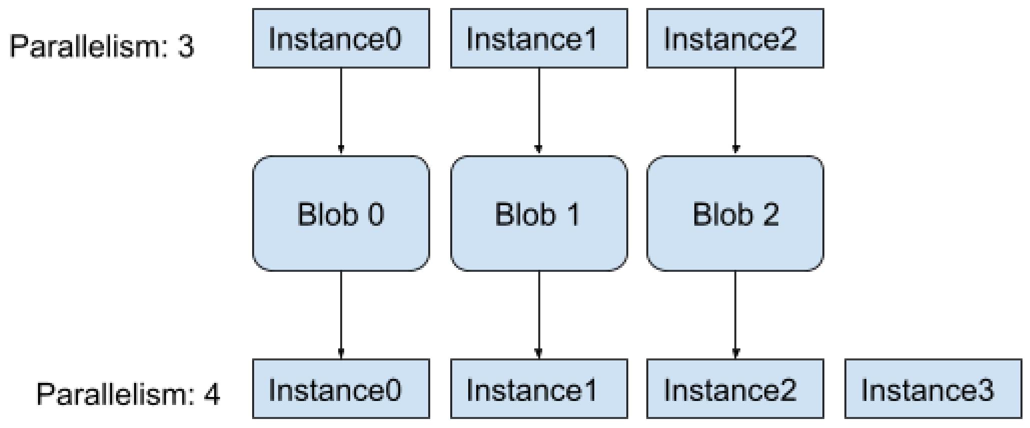


Note that repartitioning of stateful grouping is supported only by distributed stateful storage.

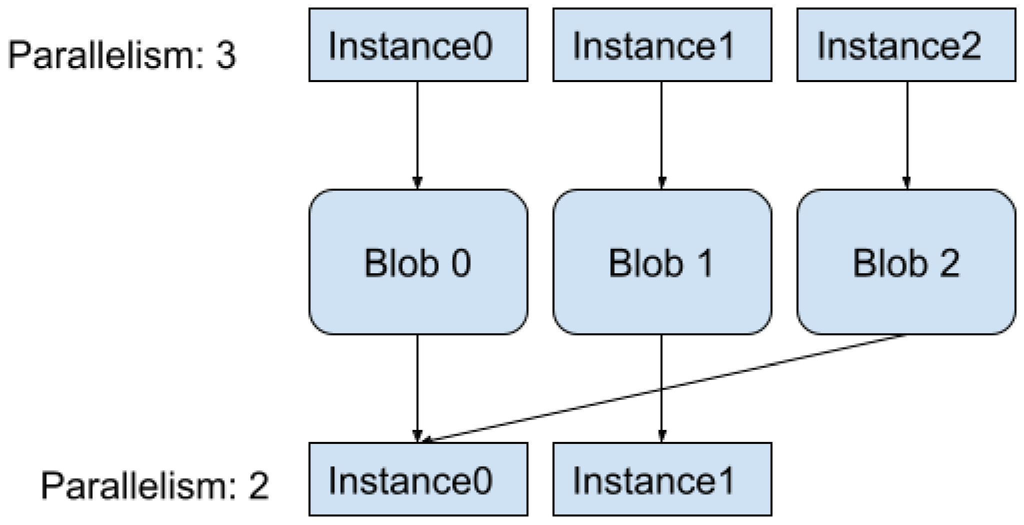
For Stateless Grouping

Repartition for stateless grouping is a lot easier. No extra partitioning layer between tuples and instances is needed.

Since it is ok for an instance to have a clean start, when adding more instances to a component, The extra instances just need to initialize an empty state object and start running(see the figure below).



When removing instances of a component (scaling down), there will be blobs left without owners. These extra stateful blobs need to be loaded by the instances left (see the figure below) and the instances need to merge the data with their own states.


Parallelism: 3
Instance0
Instance1
Instance2
Blob 0
Blob 1
Blob 2
Parallelism: 2
Instance0
Instance1