[HOPS-1385] Yarn quota for “hadoop 3”

Goals:

- Making the Yarn quota system working with the way the GPUs are handled in hadoop3
- Making the quota computation more in line with the resource usage displayed in the Yarn UI
- Simplifying the quota system implementation

Goals details:

Compatibility with Hadoop 3:

Hadoop 3 comes with support for GPU, FPGA and other kinds of resources included. As a result re removed our implementation of the GPU support to replace it by the Hadoop 3 one. This break the GPU support in the quota.
The goal here is only to fix the GPU support. Support for FPGA and other kinds of resources will be added later.

Quota computation more in line with the UI

The Yarn UI display some resource utilisation. The value displayed by the UI is based on the time of creation of the container object. The quota system was basing its measurement on the first reception of a node manager heartbeat containing this container. This results in number being different for the quota system and for the Yarn UI, which can be confusing for the user. We want to align the way these two values are computed to avoid confusion.

Simplification:

There are two levels of simplification:

- The quota system was developed with HopsYarn distributed architecture in mind. As part of the move to hadoop 3 we have removed the distributed architecture. As a result, everything is now running in the active resource manager and we should be able to reduce the complexity of the quota system taking advantage of the centralization of info. We should be able to reduce the number of service needed from 3 to 2.

- The current variable price system freeze the price applied to containers for a given time period. The problem is that it is impossible for the user to know what price is applied to each of their containers and to know when the period started and when it will finish. This results in a complicated system which is not understood by the users. We propose to simplify the system by not freezing the pricing and always apply the current price at the time of charging to the containers. This way it is easy to display the current pricing live and for the users to see if they want to keep running at the current price or stop their application. More advance pricing system (ex: application based, queue based, etc.) is future work.

Design:

We propose to divide the quota system in two services:

- A price multiplicator services in charge of computing the price multiplicator in regard to the cluster utilisation, for variable pricing.
- A quota service which follow the containers resource usage and charge the projects accordingly.

The Price multiplicator service (PriceMultiplicatorService):

The price multiplicator service consists in a thread that run at a configurable frequency, check the different cluster resources utilisations and compute a price multiplicator in function of this usage. The current multiplicator value can then be requested by calling a get on the service. The multiplicator can then be applied to the configured base price to get the current pricing for the resource.

The only resources supported for this implementation are CPUs, Memory and GPUs. The CPU and Memory are used to compute a GENERAL price, which is based on the most used of the two resources. The GPUs have their own pricing. For both the GENERAL price and the GPU price the following can be configured:

- A tipping point giving the cluster utilisation of the resource to exceed before to start increasing the prices.
- An increment which is used to decide by how much the price should be incremented for each percentage of resource usage above the tipping point.

The formulas to compute the multiplicator are as follows:

incrementBase = max(PercentageResourceUsage - ResourceTippingPoint, 0)
*multiplicator = 1 + incrementBase * ResourceIncrementFactor*

For non general resources the final multiplicator is the max of the resource specific multiplicator and the general multiplicator.

The multiplicators are saved in the database in order to be recovered in case of failover. This is to avoid a period of time where the resource manager could think that the cluster is overused or underused due to the recovery period.

Quota service (QuotaService):

The quota service compute the quota usage for running containers. For this it handle 4 kinds of events:

- CONTAINER_START
- CONTAINER_UPDATE
- CONTAINER_FINISH
- QUOTA_COMPUTE

CONTAINER_START is triggered by RMContainerImpl when the container is created and is used to register that the container should now be charged.

CONTAINER_FINISH is triggered by RMContainerImpl when the container enter in a final state. The quota service then compute the resource usage that have not been charged yet, charge the project for it and stop monitoring the container

CONTAINER_UPDATE is triggered by RMContainerImpl when the resources allocated to the container are updated. The quota service then compute the resource usage that have not been charged yet, charge the project for it and start monitoring the container with the new resource allocation.

QUOTA_COMPUTE is triggered at a configured interval by a thread running in the quota service. The quota service then compute the resource usage that have not been charged yet and charge the project for it. This is done to charge long running applications while they are running and to handle resource manager failover (more details below).

Computing and applying charge:

When a container finish, is updated or get a quota compute event, the charge corresponding to this container resource usage is computed and applied. For this the Quota service keep a map of when the container was last charged. It computes the amount of time spent between the last time the container was charged and now. It then computes the price for each of the resources used and charge for the dominant resource. The formula to compute the price is as follows:

> *Price = timeUsage * ResourceMultiplicator * basePrice * (AllocatedResource / UnitResource)*

With *timeUsage* the time computed previously in second. *ResrouceMultiplicator* the current multiplicator for the given resource (computed by the price multiplicator service)

basePrice a configurable base price for the resource, *AllocatedResource* the amount of the resource allocated to the container and *UnitResource* a configurable unit of resource (ex: 1Vcore, 1GB, 1GPU).
Once the price for the container usage is computed it is charged on the project corresponding to the application and the charging time is saved.

Resource manager Failover:

In case of failover the new active resource manager will start with an empty list of running container to monitor. This list will then be filled by the creation of RMContainer when the resource manager recover the information about the running containers on each of the node managers.
What does this imply for the charging of the containers:

- If a container start and the failover happen before than the quota service handle the CONTAINER_START event, the container will only start to be charged when the container is recovered. So the price for the time of the failover is lost.
- If containers were running before the failover, they will have been last charged during their last QUOTA_COMPUTE. They will then restart to be charged from the time of their recovery. They won't be charged for their resource utilisation between these two times.

In conclusion, in case of failover, containers will not be charged for the time between the last QUOTA_COMPUTE trigger and the failover recovery. We think that this can be mitigated by triggering QUOTA_COMPUTE often enough to make the time between QUOTA_COMPUTE and the start of the failover small and that it is fair to not charge the user for the time of the failover as the failover is due to a problem on the administrator side of the cluster.
The only way not to lose information during the failover would be to persist starting time for each of the containers in the database, but we think that this will greatly increase the load of the database for little benefit.