BP-2 - Resource aware data placement

Status

Current state: *Done*

Released: 4.5.0

Problem

It is likely that in some Bookkeeper clusters, we'll have machines with different hardware capabilities. Due to the need for expanding capacity in a cluster, it is possible that we end up having bookies of different SKUs(No. of CPUs, memory/storage capacity, etc) in a single cluster. In such environments, we can have a policy that makes sure that bookies with smaller storage capacity don't run out of disk space and turn read-only quickly. Instead we want the storage usage on those bookies to grow at a pace that is proportional to their capacity. Another policy could be to select bookies that are experiencing lower system load more often during ledger creation, than the ones that are experiencing higher load. Yet another approach could take each bookies network bandwidth into account and use it as a weight.

Proposal

1.1 Overview

Even though this proposal tries to broadly address all possible resource usage based placement of ledgers, below we'll describe in detail how disk free space based placement is envisioned to work. Other resource based policies will follow similar approach but we'll not go into their details here.

Free Storage Capacity based placement

1. We create a new boolean client configuration called 'diskWeightBasedPlacementEnabled'. Which when set to true will cause each of the existing ledger placement policies to take into account the free disk space of a bookie in addition to the existing placement logic.
2. Each bookkeeper client retrieves the free disk space info from all the RW bookies periodically. The frequency of this operation will be configurable. It can be once every 1 hour, 24 hours. etc.
3. This requires us to add a new protocol message between client and bookie. We'll make this message generic such that it can be used to query any bookie metadata like amount of average IOPS in last N seconds, free disk space, network bandwidth, etc. and call it GET_BOOKIE_INFO. The client caches this info in memory.
4. Every time a new ledger needs to be created, the ensemble is chosen based on the free disk space on the bookies. Bookies that have higher free disk space will be chosen more frequently than the ones with lower free disk space. This algorithm will give preference to bookies that have more free space.
5. As an example, suppose there are 6 bookies: B1, B2, B3, B4, B5, and B6 and the free space on each of them is 100GB, 100GB, 200GB, 200GB, 300GB, 100GB for a total of 1TB. With the new algorithm the probability of picking B1, B2, B6 will be 0.1, for B3 and B4 it is 0.2, finally for B4 it is 0.3. With the default policy the probabilities for each of the bookies is (0.16).
6. This ensures that the bookies with higher free disk space will be selected more often. Thus balancing the disk usage over time.
7. If a bookie goes down the client will purge the corresponding free disk space info from its cache. If a new bookie comes up, it will retrieve the corresponding free disk space info from the new bookie. The bookkeeper client gets notification from BookieWatcher for these two events and updates its in memory map.

1.2 Hotspots

The higher the free disk space on a node, the more write load it will be subjected to. This could cause newly added nodes with lots of free disk space to be bombarded with a lot of write traffic. One way to mitigate this is to have an upper bound on the weight for any bookie. There are a few ways to minimize the impact:

1. We set the maximum probability of selecting any single bookie to be 'N' times the probability of the bookie with the minimum weight. This should stop all new ledgers from getting created on the bookie with maximum free disk space. E.g. suppose we have bookies B1, B2, B3, B4, B5 with free disk space of 200GB, 200GB, 300GB, 500GB, and 1TB. Their respective weight becomes 0.090(i.e. 200GB/2200GB), 0.090, 0.136, 0.227, 0.4545. Since B4 and B5 have a very high weight they'll get selected for every third and second ledger create respectively. So we put a limit on it saying any bookie whose weight >= 2*min_weight, we limit it to 2*min_weight. In this example min_weight is 0.09, so the normalized weight would be: 0.090, 0.090, 0.136, 0.18, 0.18. This limits the maximum traffic B4 and B5 will be subjected. More generally, we'll implement it as N*min_weight, where N will be a configurable value.
2. An alternative but somewhat similar approach would cap the max weight based on the median of the weights. In the above example, if the starting weights are 0.090, 0.090, 0.136, 0.227, 0.4545 and we say cap using 2*median_weight, the new weights would be: 0.090, 0.090, 0.136, 0.227, 0.272.
3. A third alternative is to take into account the load on each of the systems in addition to disk weight. This requires that we collect the average load information from each of the bookies on a very regular basis, say we collect the average IOPS or disk utilization for the last 5 minutes from every bookie every 5 minutes, we could use that information in combination with the disk based weight. Bookies that are experiencing high disk utilization of say >70% will be given lower priority than the one experiencing ~30% disk utilization. Even if the bookie with the 70% disk utilization has a lot more free disk space. This approach requires a lot more communication between bookies and the clients and hence could be expensive.

The first two approaches are similar while the last approach, although more load sensitive, requires a lot more data at the client on a more regular basis, from all the bookies. One of the drawbacks of the first approach over the second is that if there is a bookie with a very small weight while the rest of the cluster have significantly more weight, this scheme will reset the weight on all the remaining nodes. For e.g. if the weights were 0.01, 0.04, 0.05, 0.1, 0.3, 0.3 and 0.3, because the smallest weight is at least 4 times smaller than the next smallest, we would end up changing (N*min_weight, where N=2) the weights of the bookies to: 0.01, 0.02, 0.02, 0.02, 0.02, 0.02. Whereas if we used the median based approach, the weights would be 0.01, 0.04, 0.05, 0.1, 0.2, 0.2, 0.2. Hence we plan to use the median as a basis for calculating the max weight.

1.3 Auto Recovery

Operations like auto recovery could cause imbalance if the weight is not taken into account. During recovery, bookies get hold of an under replicated ledger and copy data to themselves from other bookies. Since the bookies use bookkeeper client libraries to do the read and write, they should have access to the free disk space usage on all the bookies. One simple solution would be that bookies with lower weight pause after replicating a ledger. Whereas the bookies that have higher weight will take shorter pauses or no pauses at all. This guarantees that bookies that have more free disk space will end up copying more under replicated ledgers to themselves.

1.4 Implementation

1.4.1 Ensemble Placement Policy

EnsemblePlacementPolicy is an interface between BookieWatcher and the various placement policies. This interface will be enhanced and a new method will be added called 'updateBookieInfo()' to provide the up to date mapping of bookies to their metrics such as free disk space, load, etc.

The following ensemble placement policies are supported in Bookkeeper. We describe how this proposal affects these policies:

1. DefaultEnsemblePlacementPolicy: Changes will be made to newEnsemble() and replaceBookie() interfaces to select the bookies based on their free disk space based weight when the configuration suggests that weight based placement is to be honored.
2. RackawareEnsemblePlacementPolicy: Enhancements will be made to selectRandomFromRack() and selectRandomInternal() to take into account the weights of the bookies. No changes will be made to the rack selection logic. But once a rack has been selected the bookie selection within that rack will be based on disk weight.
3. RegionAwareEnsemblePlacementPolicy: No changes are needed to the region selection logic. In this policy once a region is selected, the bookie selection is done using RackawareEnsemblePlacementPolicy.

1.4.2 BookieInfo

The following protocol message is being added to retrieve the BookieInfo details. This message can be extended to retrieve other information such as network bandwidth in the future.

```
message GetBookieInfoRequest {
    enum Flags {
        TOTAL_DISK_CAPACITY = 0x01;
        FREE_DISK_SPACE = 0x02;
     }
     // bitwise OR of Flags
     optional int64 requested = 1;
}

message GetBookieInfoResponse {
    required StatusCode status = 1;
    optional int64 totalDiskCapacity = 2;
    optional int64 freeDiskSpace = 3;
}
```

1.4.3 BookieInfoReader

An instance of BookieInfoReader is created to read periodically the bookie information such as free disk space, bookie load etc. This singleton is instantiated once per Bookkeeper object. It uses a SingleThreadedExecutor to query the bookie info from all the bookies. It then communicates the updated information to the ensemble policies via the updateBookieInfo() interface described above. Changes in the number of bookies in the cluster also triggers this operation. The event is triggered from BookieWatcher singleton.

1.5 Alternatives and Strategies

1.5.1 Using zookeeper for collecting free disk space info:

One of the alternative mechanisms considered for getting the free disk space info from the bookies was to use zookeeper for storing the free disk space info. This is how it would work:

1. All the bookies report the total amount of free space they have to the metadata server(zookeeper) on a periodic basis; say every 15 minutes. On zookeeper, this info can be stored in the data part of the bookie's znode at /ledger/available/<bookieId:PortNo>. There isn't anything stored in the data part right now. Note that this info will be updated only for the RW bookies.
2. Each bookkeeper client retrieve the free disk space info of all the bookies from zk periodically. The clients' cache this info in memory.

When there are 100s or 1000s of bookies in the cluster and an equivalent number of clients, the load on zookeeper could be very high. Hence we decided against this approach.

1.6 External Interface

1.6.1 External APIs exposed

The following new client configuration parameters will be exposed:

1. ‘’diskWeightBasedPlacementEnabled’: If set to ‘true’ each of the placement policies will take into account the weight of a bookie for ledger placement.
2. ‘getBookieInfoIntervalSeconds’: will be used to control a client’s frequency of polling the bookies free disk space info.
3. ‘bookieMaxMultipleForWeightBasedPlacement’: Controls the maximum weight any given node will be given. Bookies whose natural weight is >=median_weight*‘bookieMaxMultipleForWeightBasedPlacement’, will be lowered to bookieMaxMultipleForWeightBasedPlacement*median_weight.

Since we want auto recovery to also honor the weight, these two parameters will be made part of AbstractConfiguration.

2 Monitoring and Debuggability

A new bookie shell command ‘bookkeeper shell bookieinfo’ will be added that will read the free disk space info from all the bookies and display it to the user for all the RW bookies. This would be helpful for monitoring purposes. The client will expose the weightage it is giving to each of the bookies via counters/histograms on a periodic basis.

3 Installation Rollout, Upgrade and Migration

The feature is enabled via the configuration parameter mentioned above. Since new protocol message is being introduced between client and bookie, both the client and bookie need to be upgraded before this feature can be turned on. The upgrade step would be:

- Upgrade the bookies one by one to the new build with the changes.
- Enable the weight based placement configuration parameters on the clients, upgrade the client to the new build.
- After this all new ledger creations should be weight based.

Action

The code changes will be checked in as part of Jira: https://issues.apache.org/jira/browse/BOOKKEEPER-950