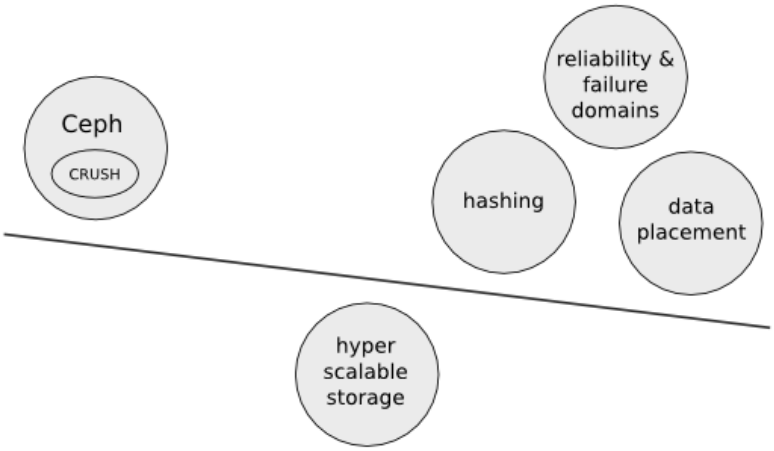
Old Habits Die Hard

just another scalable weblog

Scalable placement of replicated data in Ceph

By Javier on 30 April 2016 11:00 PM | Permalink | Comments (3)

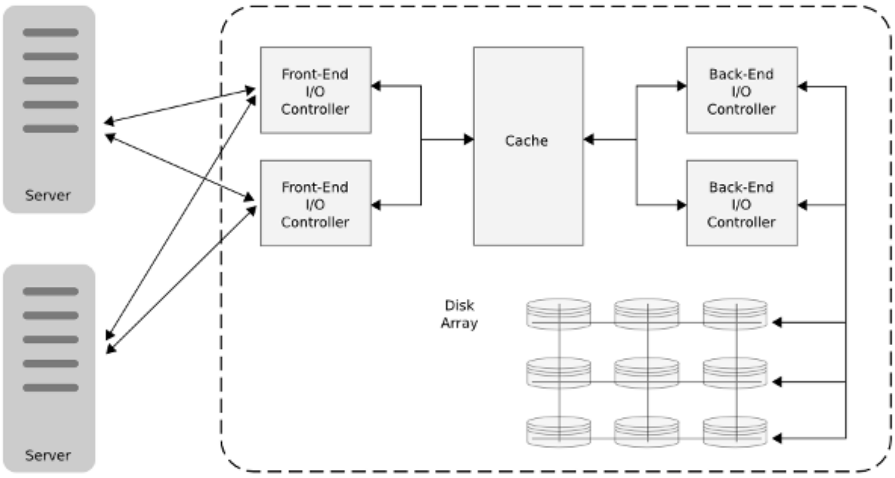


One of the most interesting and powerful features in Ceph is the way how it computes the placement and storage of a growing amount of data at hyperscale.

This computation avoids the need to look up data locations in a central directory in order to allow nodes to be added or removed, moving as few objects as possible while still maintaining balance across new cluster configurations.

Ceph and the challenges of data placement at scale

Take into consideration the traditional standard storage array in the industry. It has the usual two controller units (for redundancy) and a lot of disk trays. Those controllers connect with a storage area network (SAN) in order to provide storage to clients (servers mainly).



The disk trays are connected to the storage controllers and all storage clients use the disks through those controllers. Scaling up the capacity and performance of the array requires adding/handling more disks to the controllers.

Contact Info

Javier M. Mellid
hacking at Igalia

jmunhoz@igalia.com

Other places

Follow Javier on twitter if you want to know what he is current reading or thinking about.

Syndication

Subscribe to this weblog's atom feed or rss feed

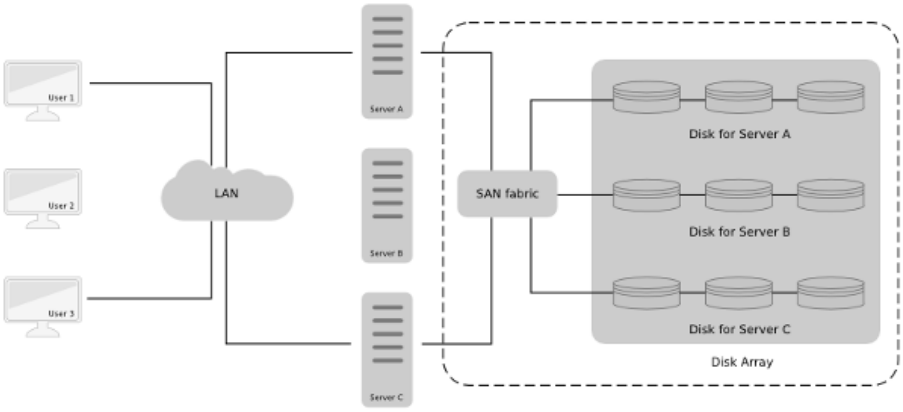
Archives

All postings

Tags

Ceph | Drones | Kernel | Security | Testing

Recent Entries

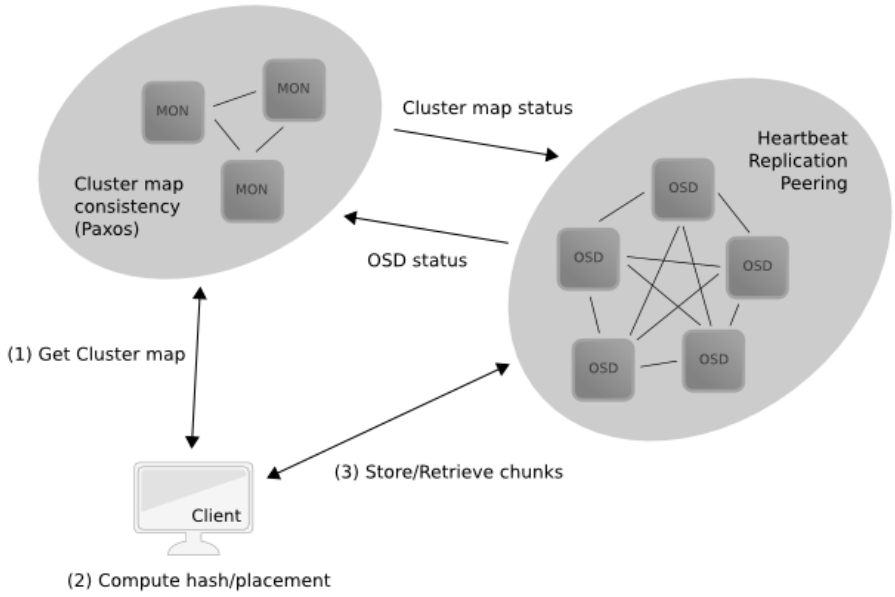


As expected, raising the order of scale the controllers become a bottleneck. They will get overloaded in some moment. The usual solution for this bottleneck is buying a new pair of controllers with a new array of disks and moving some load onto the new hardware.

This solution is expensive and not very flexible. It can work in the terabyte scale but it doesn't fit really well with the new Cloud industry demmanding elasticity and extreme flexibility on the petabyte/exabyte scale.

The Ceph approach to cope with these challenges is the design and implementation of a new scale-out distributed storage system based on commodity hardware communicating over a regular TCP/IP network.

This approach enables a fast and more affordable non-disruptive operational process on demmand while supporting the new petabyte scale with a very high performance. This kind of hyper scalable storage fits really well with the Cloud models.



The way how Ceph enables hyperscale storage is avoiding any kind of centralized coordination via autonomous and smart storage nodes. In this scale the design of the cluster software determines how many nodes the cluster can handle. If the cluster scales out to a hundred of nodes we will be working around the petabytes and millions of I/O operations per second (IOPS)

Another point to consider in this context is the inclusion of the new failure domains coming from the scale-out architecture. In this configuration each node is a new failure domain. This is usually handled by the software coordinating the cluster via copies of all data on at least two nodes. This approach is known as replica-based data protection.

Ceph is able to enable hyperscale storage via CRUSH, a hash-based algorithm for calculating how and where to store and retrieve data in a distributed-object storage cluster.

This major challenge of 'data placement at scale', and how it is designed and implemented, has a huge impact on the scalability and performance of massive storage solutions.

A "bird's-eye view" in the Ceph's data placement

Open Source UAV and mobile cellular networks
CVE-2005-3252 - Snort 2.4.0-2 remote code execution
Attending LibreCon 2017
Open Source UAV API, DroneKit-Python and Geopy
Open Source UAV Autopilot with Ardupilot and Pixhawk
Building and running RISC-V Linux rev 1.9 on QEMU
AWS4 browser-based upload goes upstream in Ceph
CVE-2017-7269 - Binary patch diffing
CVE-2017-7269 - IIS 6.0 WebDAV remote code execution
Ceph RGW AWS4 presigned URLs working with the Minio Cloud client
Multipart Upload (Copy part) goes upstream in Ceph
Attending ApacheCon and Apache Big Data Europe 2016
AWS4 chunked upload goes upstream in Ceph RGW S3
Virtual Data and Control Paths in Software-Defined Storage
Ansible AWS S3 core module now supports Ceph RGW S3
The Ceph RGW storage driver goes upstream in Libcloud
Scalable placement of replicated data in Ceph
The Outscale OSU driver goes upstream in Libcloud
Requester Pays Bucket goes upstream in Ceph
AWS Signature Version 4 goes upstream in Ceph
Ceph, a free unified distributed storage system
On S3, endpoints, regions, signatures and Boto 3
Windows 10 Kernel debugging on QEMU
Pflua and high performance packet filtering
Visit to INTECO's Cyber-Security Headquarters
Collaborating with the Carnegie Mellon Software Engineering Institute on browser security
Detecting and removing computer virus with OCaml
On Technology of Controls for Accelerators and Detectors

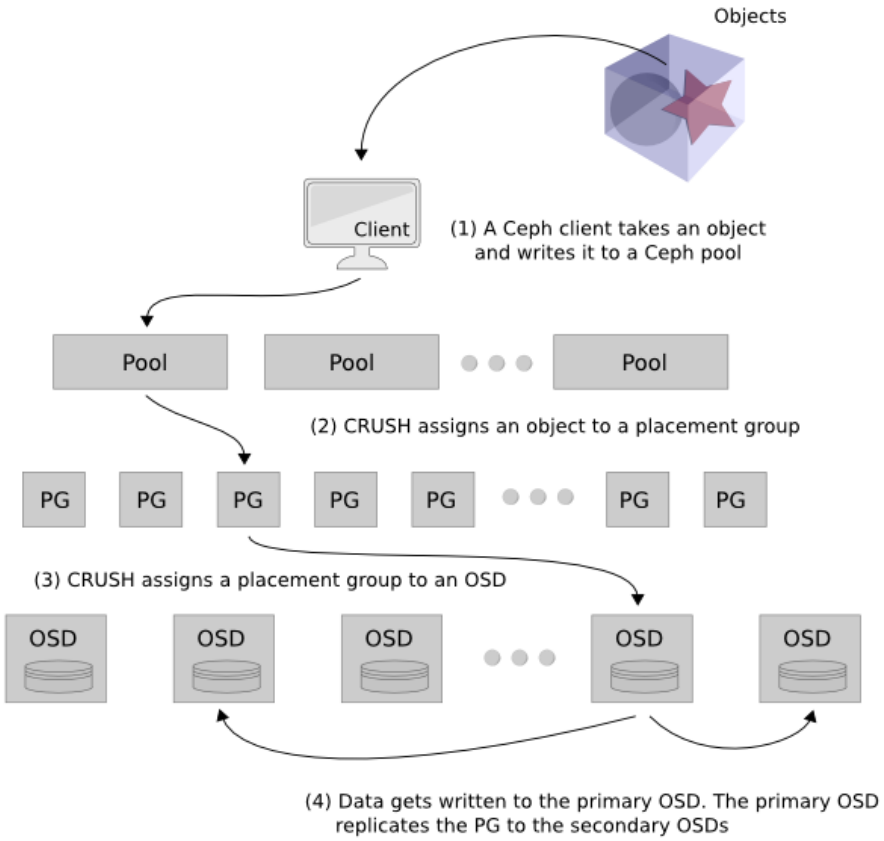
In order to explore CRUSH we need to understand concepts such as pools, placement groups, OSDs and so on. If you are not familiar with the Ceph architecture I would suggest reading one of my previous posts where all this information is summarised together with pointers to the official documentation.

From now on I will consider you are familiar with the general concepts and we will keep the focus on CRUSH and related components.

Ceph uses CRUSH to determine how to store and retrieve data by computing data storage locations. Those data are handled as objects of variable size by Ceph in a simple flat namespace. On top of these native objects, Ceph build the rest of abstractions such as blocks, file systems or S3 buckets.

In this point we can reformulate the data placement problem how an object-to-osd mapping problem without loss of generality.

In Ceph, the objects belong to pools and pools are comprised of placement groups. Each placement group maps to a list of OSDs. This is the critical path you need to understand.



The pool is the way how Ceph divides the global storage. This division or partition is the abstraction used to define the resilience (number of replicas, etc.), the number of placement groups, the CRUSH ruleset, the ownership and so on. You can consider this abstraction as the right place to define the configuration of your policies, so each pool handles its own number of replicas, number of placement groups, etc.

The placement group is the abstraction used by Ceph to map objects to OSDs in a dynamic way. You can consider it as the placement or distribution unit in Ceph.

So how we go from objects to OSDs via pools and placements groups? It is straight.

In Ceph one object will be stored in a concrete pool so the pool identifier (a number) and the name of the object are used to uniquely identify the object in the system.

Those two values, the pool id and the name of the object, are used to get a placement group via hashing.

When a pool is created it is assigned a number of placement groups (PGs). One object is always stored in a concrete pool so the pool identifier (a number) and the name of the object is used to uniquely identify each object in the system.

With the pool identifier and the hashed name of the object, Ceph will compute the hash modulo the number of PGs to retrieve the dynamic list of OSDs.

In detail, the steps to compute for one placement group for the object named 'tree' in the pool 'images' (pool id 7) with 65536 placement groups would be...

1. Hash the object name : hash('tree') = 0xA062B8CF
2. Calculates the hash modulo the number of PGs : 0xA062B8CF % 65536 = 0xB8CF
3. Get the pool id : 'images' = 7
4. Prepends the pool id to 0xB8CF to get the placement group: 7.B8CF

Ceph uses this new placement group (7.B8CF) together with the cluster map and the placement rules to get the dynamic list of OSDs...

- CRUSH('7.B8CF') = [4, 19, 3]

The size of this list is the number of replicas configured in the pool. The first OSD in the list is the primary, the next one is the secondary and so on.

Understanding CRUSH

The CRUSH (Controlled Replication Under Scalable Hashing) algorithm determines how to store and retrieve data by computing data storage locations. As mentioned in the previous lines, Ceph uses this approach to overcome the data placement at scale.

Under the hood, the algorithm requires knowing how the cluster storage is organized (device locations, hierarchies and so on). All this information is defined in the CRUSH map.

The roles and responsibilities of the CRUSH map are the following:

- Together with a ruleset for each hierarchy, the map determines how Ceph stores data.
- It contains at least one hierarchy of nodes and leaves.
- The nodes of a hierarchy are called 'buckets'. Those buckets are defined by their type.
- The data objects are distributed among the storage devices according to a per-device weight value, approximating an uniform probability distribution.
- The hierarchies are arbitraries. They are defined according to your own needs but the leaf nodes always represent the OSDs, and each leaf belong to one node or bucket.

Let's see one example of default hierarchy...

```
$ ceph osd tree
ID WEIGHT  TYPE NAME       UP/DOWN REWEIGHT PRIMARY-AFFINITY
-1 0.16672 root default
-2 0.05557     host node-1
 0 0.02779         osd.0        up  1.00000          1.00000
 1 0.02779         osd.1        up  1.00000          1.00000
-3 0.05557     host node-2
 2 0.02779         osd.2        up  1.00000          1.00000
 3 0.02779         osd.3        up  1.00000          1.00000
-4 0.05557     host node-3
 4 0.02779         osd.4        up  1.00000          1.00000
 5 0.02779         osd.5        up  1.00000          1.00000
$
```

In this case the bucket hierarchy has six leaf buckets (osd 0-5), three host buckets (node 1-3) and one root node (default)

Having a look in the decompiled map we get the following content...

```
$ ceph osd getcrushmap -o compiled-crush-map.bin
got crush map from osdmap epoch 65
$ crushtool -d compiled-crush-map.bin -o decompiled-crush-map.txt
$ cat decompiled-crush-map.txt
# begin crush map
tunable choose_local_tries 0
tunable choose_local_fallback_tries 0
tunable choose_total_tries 50
tunable chooseleaf_descend_once 1
tunable chooseleaf_vary_r 1
tunable straw_calc_version 1

# devices
device 0 osd.0
device 1 osd.1
device 2 osd.2
device 3 osd.3
device 4 osd.4
```

```
device 5 osd.5

# types
type 0 osd
type 1 host
type 2 chassis
type 3 rack
type 4 row
type 5 pdu
type 6 pod
type 7 room
type 8 datacenter
type 9 region
type 10 root

# buckets
host node-1 {
id -20 # do not change unnecessarily
# weight 0.056
alg straw
hash 0 # rjenkins1
item osd.0 weight 0.028
item osd.1 weight 0.028
}
host node-2 {
id -32 # do not change unnecessarily
# weight 0.056
alg straw
hash 0 # rjenkins1
item osd.2 weight 0.028
item osd.3 weight 0.028
}
host node-3 {
id -48 # do not change unnecessarily
# weight 0.056
alg straw
hash 0 # rjenkins1
item osd.4 weight 0.028
item osd.5 weight 0.028
}
root default {
id -10 # do not change unnecessarily
# weight 0.167
alg straw
hash 0 # rjenkins1
item node-1 weight 0.056
item node-2 weight 0.056
item node-3 weight 0.056
}

# rules
rule replicated_ruleset {
ruleset 0
type replicated
min_size 1
max_size 10
step take default
step chooseleaf firstn 0 type host
step emit
}

# end crush map
$
```

As you can see the bucket declaration requires specifying its type, an unique name, weight and hash algorithm.

```
[bucket-type] [bucket-name] {
   id [a unique negative numeric id]
   weight [the relative capacity/capability of the item(s)]
   alg [uniform, list, tree, straw, straw2]
   hash [the hash type]
   item [item-name] weight [weight]
}
```

The kinds of buckets (uniform, list, tree, straw2, etc) represent internal (non-leaf) nodes in the cluster hierarchies. Those buckets are based on different internal data structures and utilize different functions for pseudo-random choosing nested items.

The typical example is the uniform bucket, in this case all selected items are restricted in that they must contain items that are all of the same weight.

The hash type represent the hash algorithm used in the functions associated with the different kinds of buckets.

Finally, the CRUSH rules define how a Ceph client and OSDs select buckets.

```
rule  {
   ruleset
   type [ replicated | raid4]
   min_size
   max_size
   step take
   step [choose|chooseleaf] [firstn|indep]
   step emit
}
```

The Jenkins hash function

We mentioned Ceph, and CRUSH, use a hash function as part of their logic but we didn't comment anything about this function yet.

This function is known as the Jenkins hash function, a hash function for hash table lookups. One paper covering the technical details on this hash function is available here.

The paper presents fast and reliable hash functions for table lookup using 32-bit or 64-bit arithmetic together with a framework for evaluating hash functions.

In Ceph, the Jenkins function is not only used in CRUSH as part of the replicas selection. It is used along the Ceph codebase when some hashing requirement is needed.

As Jenkins comments on his paper, these hashes work equally well on all types of inputs, including text, numbers, compressed data, counting sequences, and sparse bit arrays.

I would highlight the following points related to the hash function design. They are relevant to understand the hashing code while sequencing, masking, etc. in Ceph/CRUSH the different input values.

1. If the hash value needs to be smaller that 32 (64) bits, you can mask the high bits.
2. The hash functions work best if the size of the hash table is a power of 2.
3. If the hash table has more than 232 (264) entries, this can be handled by calling the hash function twice with different initial initvals then concatenating the results.
4. If the key consists of multiple strings, the strings can be hashed sequentially, passing in the hash value from the previous string as the initval for the next.
5. Hashing a key with different initial initvals produces independent hash values.

Ceph and data placement in practice

Let's locate one object...

```
$ ceph health
HEALTH_OK
$ ceph osd tree
ID WEIGHT  TYPE NAME       UP/DOWN REWEIGHT PRIMARY-AFFINITY
-1 0.16672 root default
-2 0.05557     host node-1
 0 0.02779         osd.0        up  1.00000          1.00000
 1 0.02779         osd.1        up  1.00000          1.00000
-3 0.05557     host node-2
 2 0.02779         osd.2        up  1.00000          1.00000
 3 0.02779         osd.3        up  1.00000          1.00000
-4 0.05557     host node-3
 4 0.02779         osd.4        up  1.00000          1.00000
 5 0.02779         osd.5        up  1.00000          1.00000
$ ceph osd pool create mypool 256 256 replicated
pool 'mypool' created
$ ceph osd lspools
0 rbd,1 mypool,
$ ceph osd dump | grep mypool
pool 1 'mypool' replicated size 3 min_size 2 crush_ruleset 0 object_hash rjenkins
pg_num 256 pgp_num 256 last_change 59 flags hashpspool stripe_width 0
$ dd if=/dev/zero of=myobject bs=25M count=1 2> /dev/null
$ ls -sh myobject
25M myobject
$ rados -p mypool put myobject myobject
$ rados -p mypool stat myobject
mypool/myobject mtime 2016-04-30 10:12:10.000000, size 26214400
$ rados -p mypool ls
myobject
$ ceph osd map mypool myobject
osdmap e60 pool 'mypool' (1) object 'myobject' -> pg 1.5da41c62 (1.62)
```

```
-> up ([4,2,1], p4) acting ([4,2,1], p4)
$
```

It is mapping the object 'myobject' in the pool 'mypool' to pg 1.5da41c62 (1.62) and OSDs 4, 2 and 1

Let's see how it balances/replicates the object when OSDs go down...

```
$ ceph osd dump | grep mypool
pool 1 'mypool' replicated size 3 min_size 2 crush_ruleset 0 object_hash rjenkins
pg_num 256 pgp_num 256 last_change 59 flags hashpspool stripe_width 0
$ ceph osd map mypool myobject
osdmap e60 pool 'mypool' (1) object 'myobject' -> pg 1.5da41c62 (1.62)
-> up ([4,2,1], p4) acting ([4,2,1], p4)
$ ceph osd tree
ID WEIGHT  TYPE NAME       UP/DOWN REWEIGHT PRIMARY-AFFINITY
-1 0.16672 root default
-2 0.05557     host node-1
 0 0.02779         osd.0        up  1.00000          1.00000
 1 0.02779         osd.1        up  1.00000          1.00000
-3 0.05557     host node-2
 2 0.02779         osd.2      down  1.00000          1.00000
 3 0.02779         osd.3        up  1.00000          1.00000
-4 0.05557     host node-3
 4 0.02779         osd.4        up  1.00000          1.00000
 5 0.02779         osd.5        up  1.00000          1.00000
$ ceph osd map mypool myobject
osdmap e66 pool 'mypool' (1) object 'myobject' -> pg 1.5da41c62 (1.62)
-> up ([4,1], p4) acting ([4,1], p4)
$ ceph osd tree
ID WEIGHT  TYPE NAME       UP/DOWN REWEIGHT PRIMARY-AFFINITY
-1 0.16672 root default
-2 0.05557     host node-1
 0 0.02779         osd.0        up  1.00000          1.00000
 1 0.02779         osd.1      down  1.00000          1.00000
-3 0.05557     host node-2
 2 0.02779         osd.2      down  1.00000          1.00000
 3 0.02779         osd.3        up  1.00000          1.00000
-4 0.05557     host node-3
 4 0.02779         osd.4        up  1.00000          1.00000
 5 0.02779         osd.5        up  1.00000          1.00000
$ ceph osd map mypool myobject
osdmap e68 pool 'mypool' (1) object 'myobject' -> pg 1.5da41c62 (1.62)
-> up ([4], p4) acting ([4], p4)
$ ceph osd map mypool myobject
osdmap e96 pool 'mypool' (1) object 'myobject' -> pg 1.5da41c62 (1.62)
-> up ([4,3], p4) acting ([4,3], p4)
$ ceph osd tree
ID WEIGHT  TYPE NAME       UP/DOWN REWEIGHT PRIMARY-AFFINITY
-1 0.16672 root default
-2 0.05557     host node-1
 0 0.02779         osd.0        up  1.00000          1.00000
 1 0.02779         osd.1        up  1.00000          1.00000
-3 0.05557     host node-2
 2 0.02779         osd.2      down        0          1.00000
 3 0.02779         osd.3        up  1.00000          1.00000
-4 0.05557     host node-3
 4 0.02779         osd.4        up  1.00000          1.00000
 5 0.02779         osd.5        up  1.00000          1.00000
$ ceph osd map mypool myobject
osdmap e103 pool 'mypool' (1) object 'myobject' -> pg 1.5da41c62 (1.62)
-> up ([4,1,3], p4) acting ([4,1,3], p4)
$ ceph osd tree
ID WEIGHT  TYPE NAME       UP/DOWN REWEIGHT PRIMARY-AFFINITY
-1 0.16672 root default
-2 0.05557     host node-1
 0 0.02779         osd.0        up  1.00000          1.00000
 1 0.02779         osd.1        up  1.00000          1.00000
-3 0.05557     host node-2
 2 0.02779         osd.2        up  1.00000          1.00000
 3 0.02779         osd.3        up  1.00000          1.00000
-4 0.05557     host node-3
 4 0.02779         osd.4        up  1.00000          1.00000
 5 0.02779         osd.5        up  1.00000          1.00000
$ ceph osd map mypool myobject
osdmap e108 pool 'mypool' (1) object 'myobject' -> pg 1.5da41c62 (1.62)
-> up ([4,2,1], p4) acting ([4,2,1], p4)
$ ceph osd tree
ID WEIGHT  TYPE NAME       UP/DOWN REWEIGHT PRIMARY-AFFINITY
-1 0.16672 root default
-2 0.05557     host node-1
 0 0.02779         osd.0        up  1.00000          1.00000
 1 0.02779         osd.1        up  1.00000          1.00000
-3 0.05557     host node-2
 2 0.02779         osd.2        up  1.00000          1.00000
 3 0.02779         osd.3        up  1.00000          1.00000
-4 0.05557     host node-3
```

```
 4 0.02779         osd.4      down  1.00000          1.00000
 5 0.02779         osd.5        up  1.00000          1.00000
$ ceph osd map mypool myobject
osdmap e110 pool 'mypool' (1) object 'myobject' -> pg 1.5da41c62 (1.62)
-> up ([2,1], p2) acting ([2,1], p2)
$ ceph osd tree
ID WEIGHT  TYPE NAME       UP/DOWN REWEIGHT PRIMARY-AFFINITY
-1 0.16672 root default
-2 0.05557     host node-1
 0 0.02779         osd.0        up  1.00000          1.00000
 1 0.02779         osd.1        up  1.00000          1.00000
-3 0.05557     host node-2
 2 0.02779         osd.2        up  1.00000          1.00000
 3 0.02779         osd.3        up  1.00000          1.00000
-4 0.05557     host node-3
 4 0.02779         osd.4        up  1.00000          1.00000
 5 0.02779         osd.5        up  1.00000          1.00000
$ ceph osd map mypool myobject
osdmap e115 pool 'mypool' (1) object 'myobject' -> pg 1.5da41c62 (1.62)
-> up ([4,2,1], p4) acting ([4,2,1], p4)
```

It works as expected. It uses three replicas with a minimal size of two replicas.

Wrap-up

Along this post some challenges of data placement at hyperscale were introduced together with the Ceph approach (smart storage nodes, CRUSH algorithm, Jenkins hash function...) to address them.

Some practical examples to illustrate the mapping and data placement path are also available in the previous lines. They were tested on the new Ceph Jewel release.

As usual, I would say the primary reference to understand the current Ceph data placement is the source code. I would suggest to read the Sage's thesis (6.2, 6.3 and 6.4 sections) to know more on the roots of the current solution too. These sections cover reliable autonomic storage, performance and scalability. Beyond of these references you might also find useful the official documentation.

If you are looking for some kind of support related to development, design, deployment, etc. in Ceph or you would love to see some new feature in the next releases. Feel free to contact me!

Related posts

- Requester Pays Bucket goes upstream in Ceph
- AWS Signature Version 4 goes upstream in Ceph
- Ceph, a free unified distributed storage system
- On S3, endpoints, regions, signatures and Boto 3

« The Outscale OSU driver goes upstream in Libcloud

The Ceph RGW storage driver goes upstream in Libcloud »

Comments

Generated by Jekyll