LOCALLY REPAIRABLE ERASURE CODE PLUGIN

With the *jerasure* plugin, when an erasure coded object is stored on multiple OSDs, recovering from the loss of one OSD requires reading from all the others. For instance if *jerasure* is configured with *k=8* and *m=4*, losing one OSD requires reading from the eleven others to repair.

The *lrc* erasure code plugin creates local parity chunks to be able to recover using less OSDs. For instance if *lrc* is configured with *k=8*, *m=4* and *l=4*, it will create an additional parity chunk for every four OSDs. When a single OSD is lost, it can be recovered with only four OSDs instead of eleven.

ERASURE CODE PROFILE EXAMPLES

REDUCE RECOVERY BANDWIDTH BETWEEN HOSTS

Although it is probably not an interesting use case when all hosts are connected to the same switch, reduced bandwidth usage can actually be observed.:

```
$ ceph osd erasure-code-profile set LRCprofile \
     plugin=lrc \
     k=4 m=2 l=3 \
     crush-failure-domain=host
$ ceph osd pool create lrcpool 12 12 erasure LRCprofile
```

REDUCE RECOVERY BANDWIDTH BETWEEN RACKS

In Firefly the reduced bandwidth will only be observed if the primary OSD is in the same rack as the lost chunk.:

```
$ ceph osd erasure-code-profile set LRCprofile \
     plugin=lrc \
     k=4 m=2 l=3 \
     crush-locality=rack \
     crush-failure-domain=host
$ ceph osd pool create lrcpool 12 12 erasure LRCprofile
```

CREATE AN LRC PROFILE

To create a new lrc erasure code profile:

```
ceph osd erasure-code-profile set {name} \
     plugin=lrc \
     k={data-chunks} \
     m={coding-chunks} \
     l={locality} \
     [crush-root={root}] \
     [crush-locality={bucket-type}] \
     [crush-failure-domain={bucket-type}] \
     [crush-device-class={device-class}] \
     [directory={directory}] \
     [--force]
```

Where:

`k={data chunks}`

Description: Each object is split in **data-chunks** parts, each stored on a different OSD.
Type: Integer
Required: Yes.
Example: 4

`m={coding-chunks}`

Description: Compute **coding chunks** for each object and store them on different OSDs. The number of coding chunks is also the number of OSDs that can be down without losing data.
Type: Integer
Required: Yes.
Example: 2

`l={locality}`

Description: Group the coding and data chunks into sets of size **locality**. For instance, for **k=4** and **m=2**, when **locality=3** two groups of three are created. Each set can be recovered without reading chunks from another set.
Type: Integer
Required: Yes.
Example: 3

`crush-root={root}`

Description: The name of the crush bucket used for the first step of the CRUSH rule. For intance **step take default**.
Type: String
Required: No.
Default: default

`crush-locality={bucket-type}`

Description: The type of the crush bucket in which each set of chunks defined by **l** will be stored. For instance, if it is set to **rack**, each group of **l** chunks will be placed in a different rack. It is used to create a CRUSH rule step such as **step choose rack**. If it is not set, no such grouping is done.
Type: String
Required: No.

`crush-failure-domain={bucket-type}`

Description: Ensure that no two chunks are in a bucket with the same failure domain. For instance, if the failure domain is **host** no two chunks will be stored on the same host. It is used to create a CRUSH rule step such as **step chooseleaf host**.
Type: String
Required: No.
Default: host

`crush-device-class={device-class}`

Description: Restrict placement to devices of a specific class (e.g., `ssd` or `hdd`), using the crush device class names in the CRUSH map.
Type: String
Required: No.
Default:

`directory={directory}`

Description: Set the **directory** name from which the erasure code plugin is loaded.
Type: String
Required: No.
Default: /usr/lib/ceph/erasure-code

`--force`

Description: Override an existing profile by the same name.
Type: String
Required: No.

LOW LEVEL PLUGIN CONFIGURATION

The sum of **k** and **m** must be a multiple of the **l** parameter. The low level configuration parameters do not impose such a restriction and it may be more convienient to use it for specific purposes. It is for instance possible to define two groups, one with 4 chunks and another with 3 chunks. It is also possible to recursively define locality sets, for instance datacenters and racks into datacenters. The **k/m/l** are implemented by generating a low level configuration.

The *lrc* erasure code plugin recursively applies erasure code techniques so that recovering from the loss of some chunks only requires a subset of the available chunks, most of the time.

For instance, when three coding steps are described as:

```
chunk nr    01234567
step 1      _cDD_cDD
step 2      cDDD____
step 3      ____cDDD
```

where *c* are coding chunks calculated from the data chunks *D*, the loss of chunk *7* can be recovered with the last four chunks. And the loss of chunk *2* chunk can be recovered with the first four chunks.

ERASURE CODE PROFILE EXAMPLES USING LOW LEVEL CONFIGURATION

MINIMAL TESTING

It is strictly equivalent to using the default erasure code profile. The *DD* implies *K=2*, the *c* implies *M=1* and the *jerasure* plugin is used by default.:

```
$ ceph osd erasure-code-profile set LRCprofile \
     plugin=lrc \
     mapping=DD_ \
     layers='[ [ "DDc", "" ] ]'
$ ceph osd pool create lrcpool 12 12 erasure LRCprofile
```

REDUCE RECOVERY BANDWIDTH BETWEEN HOSTS

Although it is probably not an interesting use case when all hosts are connected to the same switch, reduced bandwidth usage can actually be observed. It is equivalent to **k=4**, **m=2** and **l=3** although the layout of the chunks is different:

```
$ ceph osd erasure-code-profile set LRCprofile \
     plugin=lrc \
     mapping=__DD__DD \
     layers='[
               [ "_cDD_cDD", "" ],
               [ "cDDD____", "" ],
               [ "____cDDD", "" ],
             ]'
$ ceph osd pool create lrcpool 12 12 erasure LRCprofile
```

REDUCE RECOVERY BANDWIDTH BETWEEN RACKS

In Firefly the reduced bandwidth will only be observed if the primary OSD is in the same rack as the lost chunk.:

```
$ ceph osd erasure-code-profile set LRCprofile \
     plugin=lrc \
     mapping=__DD__DD \
     layers='[
               [ "_cDD_cDD", "" ],
               [ "cDDD____", "" ],
               [ "____cDDD", "" ],
             ]' \
     crush-steps='[
                    [ "choose", "rack", 2 ],
                    [ "chooseleaf", "host", 4 ],
                  ]'
```

```
$ ceph osd pool create lrcpool 12 12 erasure LRCprofile
```

TESTING WITH DIFFERENT ERASURE CODE BACKENDS

LRC now uses jerasure as the default EC backend. It is possible to specify the EC backend/algorithm on a per layer basis using the low level configuration. The second argument in layers=’[[“DDc”, “”]]’ is actually an erasure code profile to be used for this level. The example below specifies the ISA backend with the cauchy technique to be used in the lrcpool.:

```
$ ceph osd erasure-code-profile set LRCprofile \
     plugin=lrc \
     mapping=DD_ \
     layers='[ [ "DDc", "plugin=isa technique=cauchy" ] ]'
$ ceph osd pool create lrcpool 12 12 erasure LRCprofile
```

You could also use a different erasure code profile for for each layer.:

```
$ ceph osd erasure-code-profile set LRCprofile \
     plugin=lrc \
     mapping=__DD__DD \
     layers='[
               [ "_cDD_cDD", "plugin=isa technique=cauchy" ],
               [ "cDDD____", "plugin=isa" ],
               [ "____cDDD", "plugin=jerasure" ],
             ]'
$ ceph osd pool create lrcpool 12 12 erasure LRCprofile
```

ERASURE CODING AND DECODING ALGORITHM

The steps found in the layers description:

```
chunk nr    01234567

step 1      _cDD_cDD
step 2      cDDD____
step 3      ____cDDD
```

are applied in order. For instance, if a 4K object is encoded, it will first go thru *step 1* and be divided in four 1K chunks (the four uppercase D). They are stored in the chunks 2, 3, 6 and 7, in order. From these, two coding chunks are calculated (the two lowercase c). The coding chunks are stored in the chunks 1 and 5, respectively.

The *step 2* re-uses the content created by *step 1* in a similar fashion and stores a single coding chunk *c* at position 0. The last four chunks, marked with an underscore (_) for readability, are ignored.

The *step 3* stores a single coding chunk *c* at position 4. The three chunks created by *step 1* are used to compute this coding chunk, i.e. the coding chunk from *step 1* becomes a data chunk in *step 3*.

If chunk *2* is lost:

```
chunk nr    01234567

step 1      _c D_cDD
step 2      cD D____
step 3      __ _cDDD
```

decoding will attempt to recover it by walking the steps in reverse order: *step 3* then *step 2* and finally *step 1*.

The *step 3* knows nothing about chunk *2* (i.e. it is an underscore) and is skipped.

The coding chunk from *step 2*, stored in chunk *0*, allows it to recover the content of chunk *2*. There are no more chunks to recover and the process stops, without considering *step 1*.

Recovering chunk *2* requires reading chunks *0, 1, 3* and writing back chunk *2*.

If chunk *2*, *3*, *6* are lost:

```
chunk nr    01234567

step 1      _c  _c D
step 2      cD  __ _
step 3      __  cD D
```

The *step 3* can recover the content of chunk *6*:

```
chunk nr    01234567

step 1      _c  _cDD
step 2      cD  ____
step 3      __  cDDD
```

The *step 2* fails to recover and is skipped because there are two chunks missing (*2*, *3*) and it can only recover from one missing chunk.

The coding chunk from *step 1*, stored in chunk *1*, *5*, allows it to recover the content of chunk *2*, *3*:

```
chunk nr    01234567

step 1      _cDD_cDD
step 2      cDDD____
step 3      ____cDDD
```

CONTROLLING CRUSH PLACEMENT

The default CRUSH rule provides OSDs that are on different hosts. For instance:

```
chunk nr    01234567

step 1      _cDD_cDD
step 2      cDDD____
step 3      ____cDDD
```

needs exactly *8* OSDs, one for each chunk. If the hosts are in two adjacent racks, the first four chunks can be placed in the first rack and the last four in the second rack. So that recovering from the loss of a single OSD does not require using bandwidth between the two racks.

For instance:

```
crush-steps='[ [ "choose", "rack", 2 ], [ "chooseleaf", "host", 4 ] ]'
```

will create a rule that will select two crush buckets of type *rack* and for each of them choose four OSDs, each of them located in different buckets of type *host*.

The CRUSH rule can also be manually crafted for finer control.