ERASURE CODE DEVELOPER NOTES

INTRODUCTION

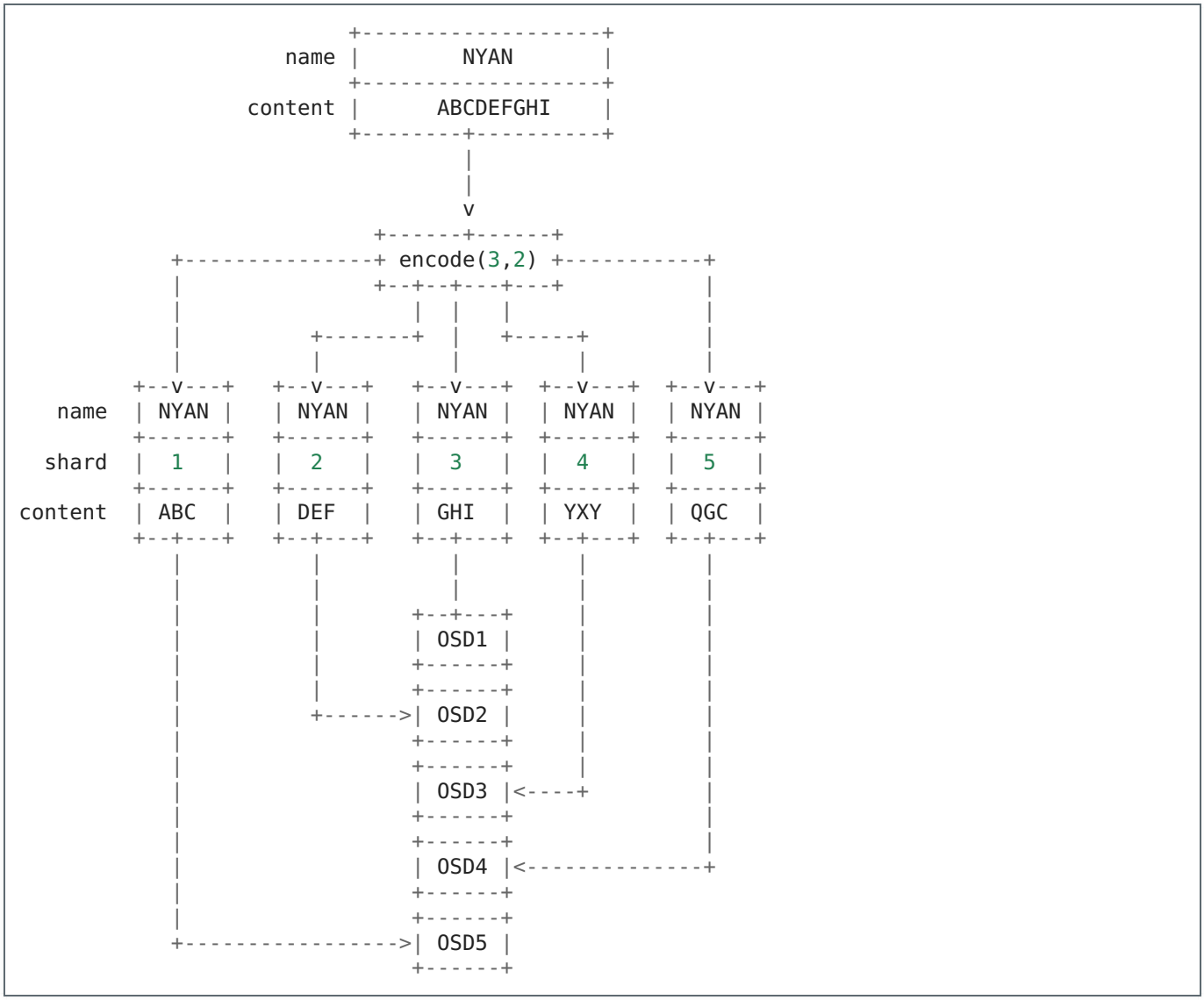
Each chapter of this document explains an aspect of the implementation of the erasure code within Ceph. It is mostly based on examples being explained to demonstrate how things work.

READING AND WRITING ENCODED CHUNKS FROM AND TO OSDS

An erasure coded pool stores each object as K+M chunks. It is divided into K data chunks and M coding chunks. The pool is configured to have a size of K+M so that each chunk is stored in an OSD in the acting set. The rank of the chunk is stored as an attribute of the object.

Let's say an erasure coded pool is created to use five OSDs (K+M = 5) and sustain the loss of two of them (M = 2).

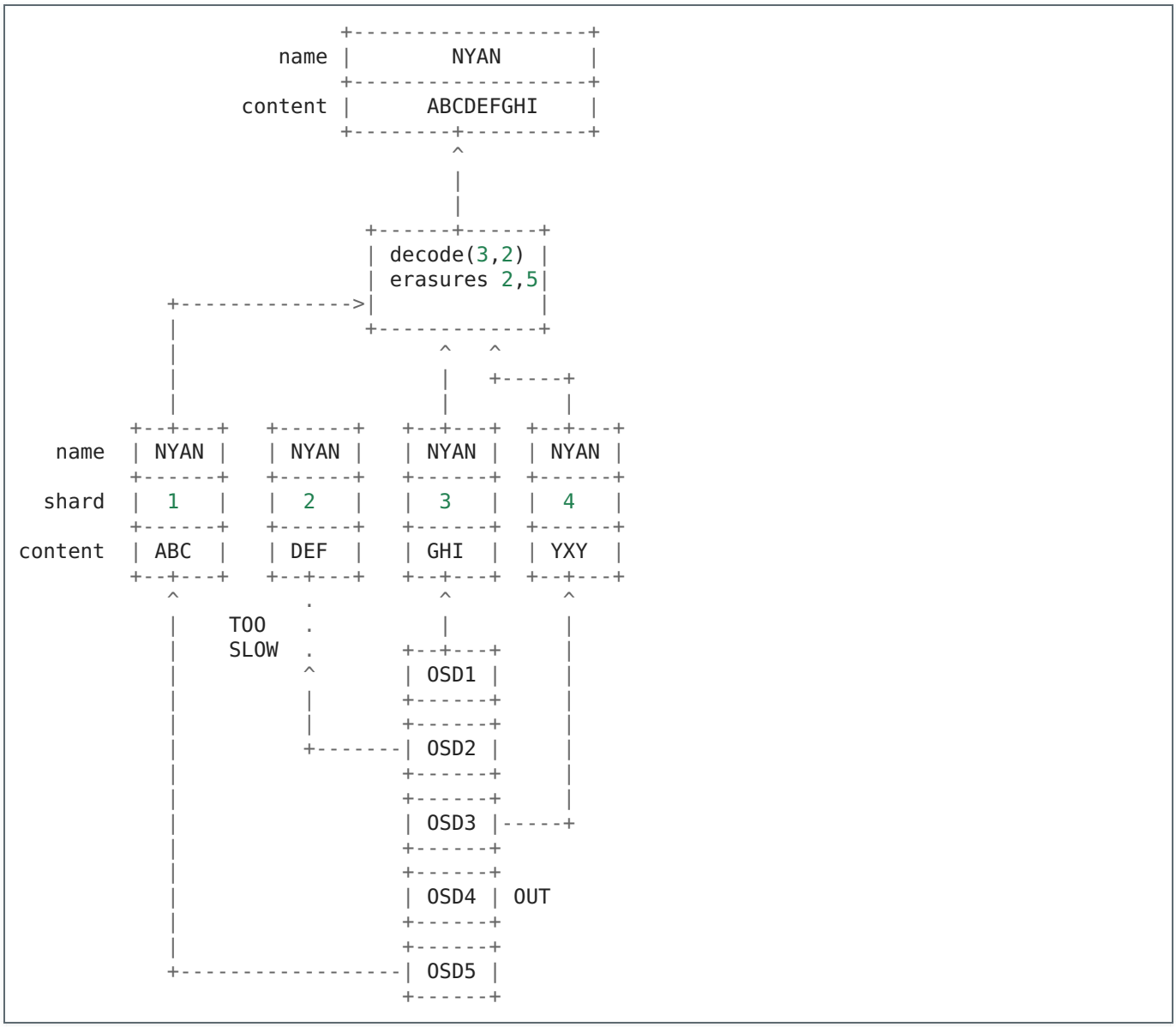
When the object *NYAN* containing *ABCDEFGHI* is written to it, the erasure encoding function splits the content in three data chunks, simply by dividing the content in three : the first contains *ABC*, the second *DEF* and the last *GHI*. The content will be padded if the content length is not a multiple of K. The function also creates two coding chunks : the fourth with *YXY* and the fifth with *GQC*. Each chunk is stored in an OSD in the acting set. The chunks are stored in objects that have the same name (*NYAN*) but reside on different OSDs. The order in which the chunks were created must be preserved and is stored as an attribute of the object (shard_t), in addition to its name. Chunk *1* contains *ABC* and is stored on *OSD5* while chunk *4* contains *YXY* and is stored on *OSD3*.



When the object *NYAN* is read from the erasure coded pool, the decoding function reads three chunks : chunk *1* containing *ABC*, chunk *3* containing *GHI* and chunk *4* containing *YXY* and rebuild the original content of the object *ABCDEFGHI*. The decoding function is informed that the chunks *2* and *5* are missing (they are called *erasures*). The chunk *5* could not be read

because the *OSD4* is *out*.

The decoding function could be called as soon as three chunks are read : *OSD2* was the slowest and its chunk does not need to be taken into account. This optimization is not implemented in Firefly.



ERASURE CODE LIBRARY

Using Reed-Solomon, with parameters K+M, object O is encoded by dividing it into chunks O1, O2, ... OM and computing coding chunks P1, P2, ... PK. Any K chunks out of the available K+M chunks can be used to obtain the original object. If data chunk O2 or coding chunk P2 are lost, they can be repaired using any K chunks out of the K+M chunks. If more than M chunks are lost, it is not possible to recover the object.

Reading the original content of object O can be a simple concatenation of O1, O2, ... OM, because the plugins are using systematic codes. Otherwise the chunks must be given to the erasure code library *decode* method to retrieve the content of the object.

Performance depend on the parameters to the encoding functions and is also influenced by the packet sizes used when calling the encoding functions (for Cauchy or Liberation for instance): smaller packets means more calls and more overhead.

Although Reed-Solomon is provided as a default, Ceph uses it via an abstract API designed to allow each pool to choose the plugin that implements it using key=value pairs stored in an erasure code profile.

```
$ ceph osd erasure-code-profile set myprofile \
    crush-failure-domain=osd
$ ceph osd erasure-code-profile get myprofile
directory=/usr/lib/ceph/erasure-code
k=2
m=1
plugin=jerasure
```

```
technique=reed_sol_van
crush-failure-domain=osd
$ ceph osd pool create ecpool 12 12 erasure myprofile
```

The *plugin* is dynamically loaded from *directory* and expected to implement the *int __erasure_code_init(char *plugin_name, char *directory)* function which is responsible for registering an object derived from *ErasureCodePlugin* in the registry. The ErasureCodePluginExample plugin reads:

```
ErasureCodePluginRegistry &instance =
                          ErasureCodePluginRegistry::instance();
instance.add(plugin_name, new ErasureCodePluginExample());
```

The *ErasureCodePlugin* derived object must provide a factory method from which the concrete implementation of the *ErasureCodeInterface* object can be generated. The ErasureCodePluginExample plugin reads:

```
virtual int factory(const map<std::string,std::string> &parameters,
                    ErasureCodeInterfaceRef *erasure_code) {
  *erasure_code = ErasureCodeInterfaceRef(new ErasureCodeExample(parameters));
  return 0;
}
```

The *parameters* argument is the list of *key=value* pairs that were set in the erasure code profile, before the pool was created.

```
ceph osd erasure-code-profile set myprofile \
   directory=<dir>         \ # mandatory
   plugin=jerasure         \ # mandatory
   m=10                    \ # optional and plugin dependant
   k=3                     \ # optional and plugin dependant
   technique=reed_sol_van  \ # optional and plugin dependant
```

NOTES

If the objects are large, it may be impractical to encode and decode them in memory. However, when using *RBD* a 1TB device is divided in many individual 4MB objects and *RGW* does the same.

Encoding and decoding is implemented in the OSD. Although it could be implemented client side for read write, the OSD must be able to encode and decode on its own when scrubbing.