Erasure Coding Worker

Overview

Ref. the HDFS EC design attached in HDFS-7285, we need to extend DataNode and provide relevant support to perform the EC encoding and decoding work. Before dive into implementation details, let's discuss high level use cases and tasks we should handle here in this part.

Use Cases

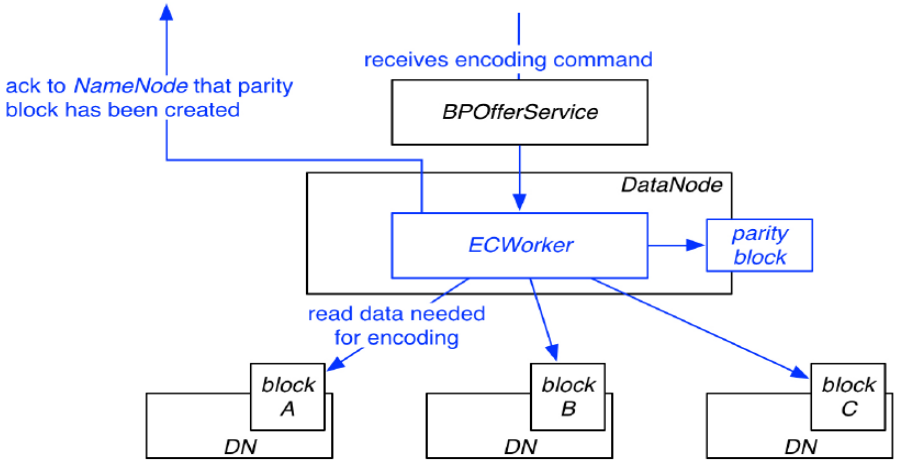
1. Convert replication blocks into **stripping** coding blocks
2. Recover erased **stripping** coding blocks
3. Convert replication blocks into **non-stripping** erasure coding blocks
4. Recover erased **non-stripping** erasure coding blocks
5. On-demand recovery serving client request for erased **non-stripping** erasure coding blocks

Scope

This doc focuses on the design level discussion, and for high level strategies we prefer to cover them in the overall doc in HDFS-7285. For example, how to form a block group in an encoding considering which files to choose blocks from? Limit to a large file only or combine multiple small files? Etc. In all cases it starts with received coding commands from NameNode or whatever services/tools.

ECWorker

For convenience, we copy the figure from the overall design doc in HDFS-7285 here, as follows.

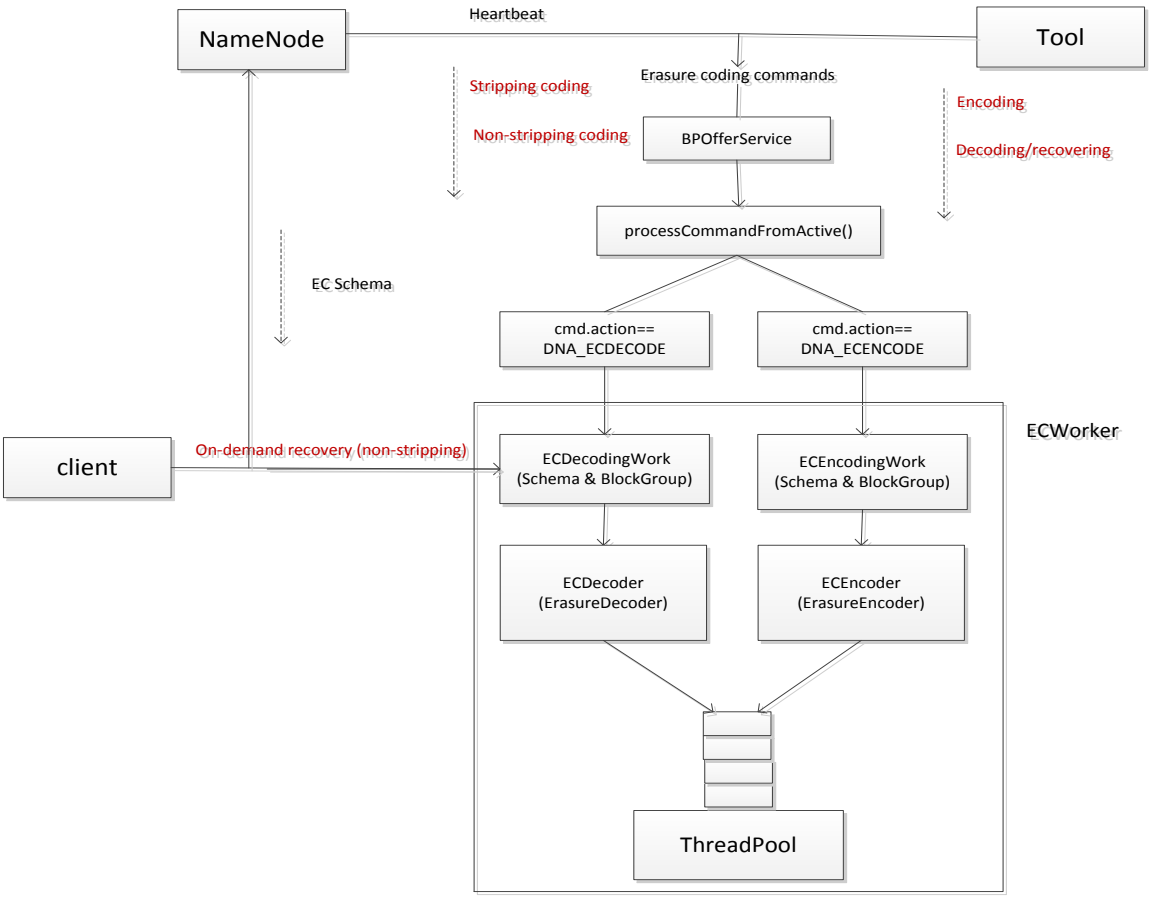


DataNode extensions

This doc will focus on how to implement ECWorker targeting for the concrete use cases. It's desired for good discussion before implementation as the change should be careful not to break existing functionalities and involve big performance impact.

ECWorker would be a real construction to organize all the related codes together. In a whole it can be instanced during DataNode service bootstrap, and initialized with configurations. If the DataNode isn't to perform any coding work at all, then its overhead should be ignorable, like no thread pool and coding chunks pool will be created at all. It provides relevant methods to be hooked into DataNode relevant components. The modification into existing components should be minimized and all the related work should be delegated to and handled in ECWorker.

The following figure illustrates the overall ECWorker and how encoding/decoding works are processed.



ECWorker

1. NameNode assigns encoding/decoding work to DataNodes thru heartbeat commands.
2. BPOfferService will parse commands from NameNode and constructs ECEncodingWork or ECDecodingWork accordingly, which wraps BlockGroup plus schema info for the coding work.
3. ECWorker processes ECWork (either ECEncodingWork or ECDecodingWork) by creating ECEncoder or ECDecoder accordingly. The resultant ECEncoder or ECDecoder will be put into ECCoderQueue and processed by a thread pool. Note in most often time there may be just a coding work in a DataNode, there may be also many coding works concurrently.
4. We may have a tool to compute and distribute coding works to DataNodes in some cases to share NameNode workload.
5. Decoding work can also occur in the passive recovery case to serve clients for erased blocks on demand in non-stripping case.

BlockGroup

From ErasureCoder defined in the codec framework allows encoding or decoding a block group in the whole and hides the specific encoding/decoding logic for various erasure codes. To utilize ErasureCoder a BlockGroup should be constructed according to received coding command in all cases.

ECCodingWork

From received erasure coding command, in addition to a BlockGroup, we should also get the associated EC schema info, so we will be able to construct an ECCodingWork that wraps all the necessary information together and provides convenient facility methods. We might have ECEncodingWork or ECDecodingWork, even ECStrippingEncodingWork and ECStrippingDecodingWork if it helps.

ECCoder

Given an ECCodingWork, an ECCoder, either ECEncoder or ECDecoder can be constructed, which can be a runnable, to put into a coders queue for a thread pool to execute. A coder utilizes BlockReader to read source input blocks locally or remotely, and BlockWriter to write resultant output blocks locally or remotely. For the erasure coding computation, it creates and utilizes an ErasureCoder with the codec name specified in the schema. A coder may have priority so urgent ones can be scheduled and dispatched first. Below figure is an illustration of ECEncoder, ECDecoder would be similar.

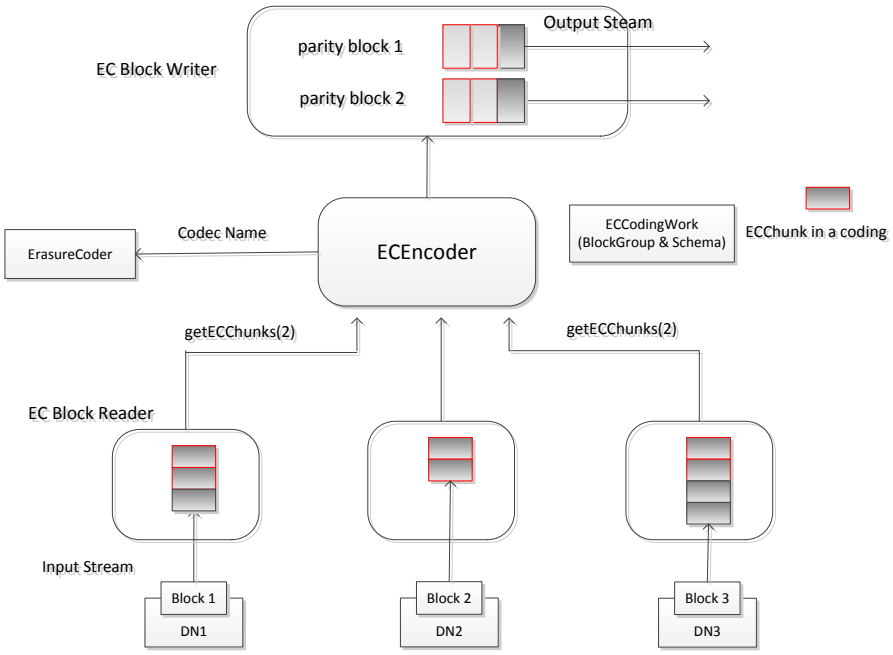


Figure 3 ECEncoder

BlockReader/Writer

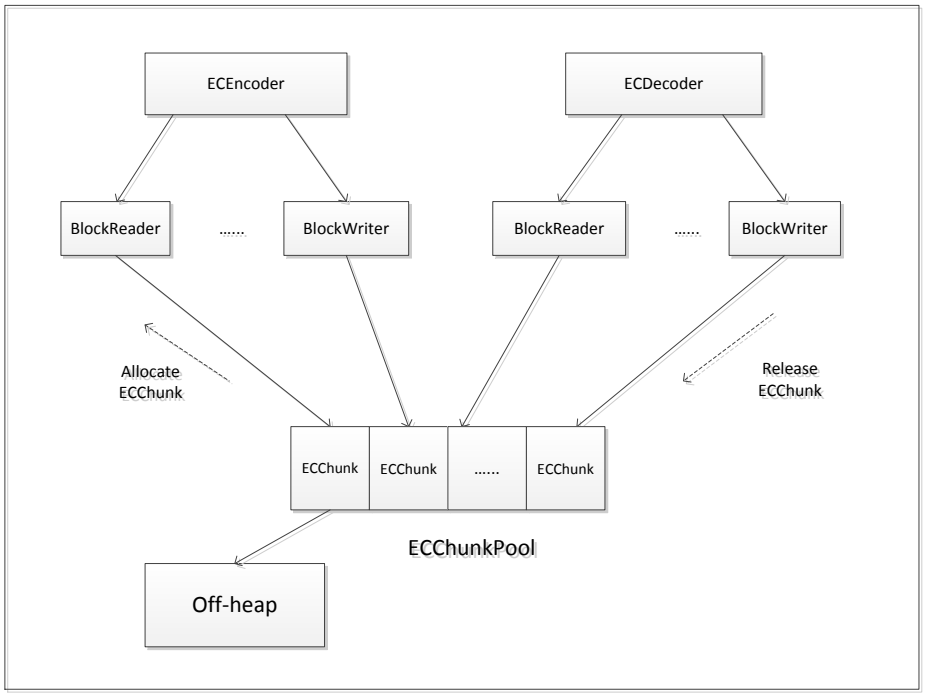
To perform erasure coding against a group of source data blocks, we need a BlockReader to read block data locally or remotely. It's good to be a general one in DataNode, but if not possible then an EC dedicated one would also work. Similarly, to persist the coding resultant output blocks, we also need a BlockWriter to write block data locally or remotely. It's good to be a general one in DataNode, but if not possible then an EC dedicated one would also work.

There is discussion about unifying BlockReader/Writer along with above discussed ECCoder for both client and DataNode sides. It may be possible and we desire such effort, but leave it for future consideration after we have both codes done in the two sides.

ECChunkPool

To manage coding chunk buffers for ECCoders, ECChunkPool is used as follows. BlockReader and BlockWriter require ECChunkPool to allocate a chunk buffer when needed; in most time, ECChunkPool will return one from the pool if available, otherwise allocates multiple chunks from off-heap one time. When not used any longer, ECChunk buffers will be released and returned back to ECChunkPool for future reuse.

ECWorker



Handling failures

In the overall design discussion, each coding work once is done, an ACK will be sent back to the scheduler (NameNode or other services/tools), for the scheduler to perform following cleanup work, like delete redundant replicas. It's possible to fail during block reading, coding and block writing. In some cases we may just abort the work, so the scheduler can schedule another one with some adjustment. In some cases we may wish not to restart the whole work at all, for example, in an encoding work where an output target DataNode disconnects, it could still be able to go on if we can simply choose another target DataNode to write the output block, or just persist it locally for the scheduler to move it later.

At the very beginning, we would write output blocks locally in order to avoid writing-remote failures, if the simplifying can help for the related tasks.

Passive Recovery

In non-stripping erasure coding, client may request a block that's erased and to be recovered on demand. In such case, NameNode may return a block location for the recovering. In case the target DataNode fails, more than block locations can be returned as virtual locations. When a DataNode is requested for a virtual block to be recovered, it should start to reconstruct the block immediately and serve data on command. For the reconstruction, DataNode needs to request to NameNode to retrieve block group and schema infor using an RPC sync call. Such RPC could be desired by DataNode in other chances as well and provides possible optimization opportunities in future.

Future Considerations

1. How to detect edible DataNodes for erasure coding work
   In a cluster not all DataNodes are of the powerful capability necessarily that's edible to perform erasure coding tasks. In this design we regard all DataNodes are the same powerful. Ideally we should be able to detect the hardware properties automatically during the service bootstrap and report the capabilities to NameNode or schedulers for it to consider when scheduling coding tasks.
2. How an ECWorker can coordinate other ECWorkers for a complex coding work
   There is good discussion that a coding work can be break down into sub-tasks so the ECWorker can distribute them to other ECWorkers for sharing the work load and also data locality. As this isn't trivial so we'd better leave this consideration for future to improve.