Actions at the NameNode

NameNode as part of its initialization, starts commons services, and activates the **BlockManager** as part of it.

BlockManager periodically runs a redundancy monitor thread to compute block reconstruction work that can be scheduled on data-nodes.

The method computeReconstructionWorkForBlocks() in **BlockManager** calls validateReconstructionWork().

This method does checks to validate that the reconstruction of the block is actually required and calls addTaskToDatanode() of **ErasureCodingWork** class.

This way, the datanode is informed about this recovery work at the next heartbeat.

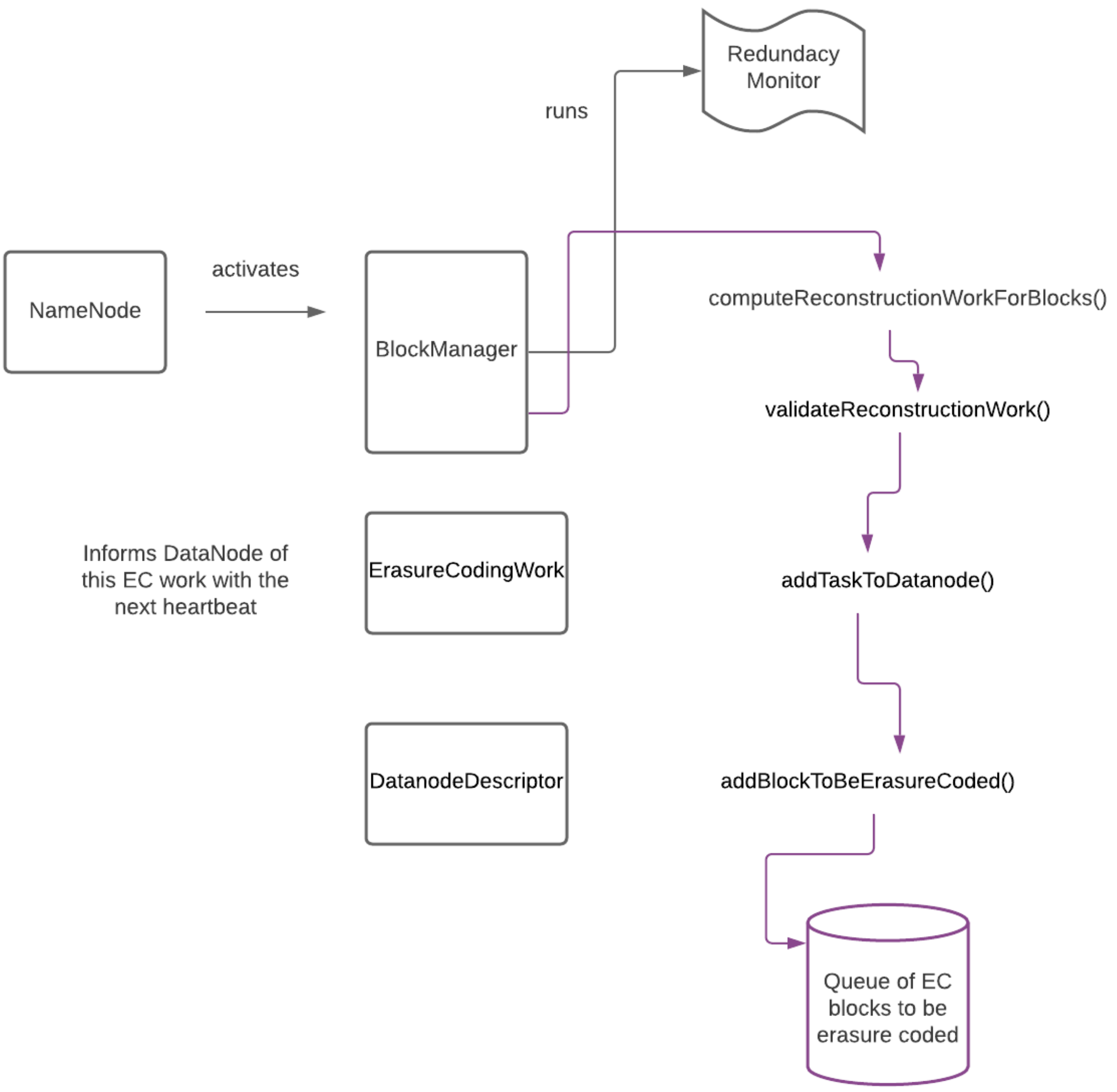
The method addTaskToDatanode() calls addBlockToBeErasureCoded() in the class **DatanodeDescriptor**.

addTaskToDatanode() constructs a new erasure coding task, encapsulated as a **BlockECReconstructionInfo** having details**:**
`block, sources, targets, liveBlockIndices, ecPolicy`.

It adds this task to the queue of EC blocks to be erasure coded by this datanode.

BlockECReconstructionCommand creates a EC reconstruction command from a collection of **BlockECReconstructionInfo**, each representing a reconstruction task.

BlockECReconstructionCommand is an instruction to a DataNode to reconstruct a striped block group with missing blocks.



Redundancy Monitor
runs
NameNode
activates
BlockManager
computeReconstructionWorkForBlocks()
validateReconstructionWork()
Informs DataNode of this EC work with the next heartbeat
ErasureCodingWork
addTaskToDatanode()
DatanodeDescriptor
addBlockToBeErasureCoded()
Queue of EC blocks to be erasure coded

Actions at the NameNode for EC reconstruction

Fig. 1

Actions at the DataNode

Datanodes run **BPServiceActor** which is a thread that connects to the NameNode for pre-registration handshake, registration with NameNode, sending periodic heartbeats to the NameNode and handle basic commands received from the NameNode.

It processes datanode commands calling processCommand() with a list of datanode commands issued by the Namenode.

processCommand() calls processCommandFromActor() and then processCommandFromActive() in **BPOfferService**.

BPOfferService handles heartbeats to the NameNode.

processCommandFromActive() calls processErasureCodingTasks() in **ErasureCodingWorker** with a list of EC tasks encapsulated as a collection of **BlockECReconstructionInfo.**

ErasureCodingWorker is the class that handles EC reconstruction commands at the datanode.

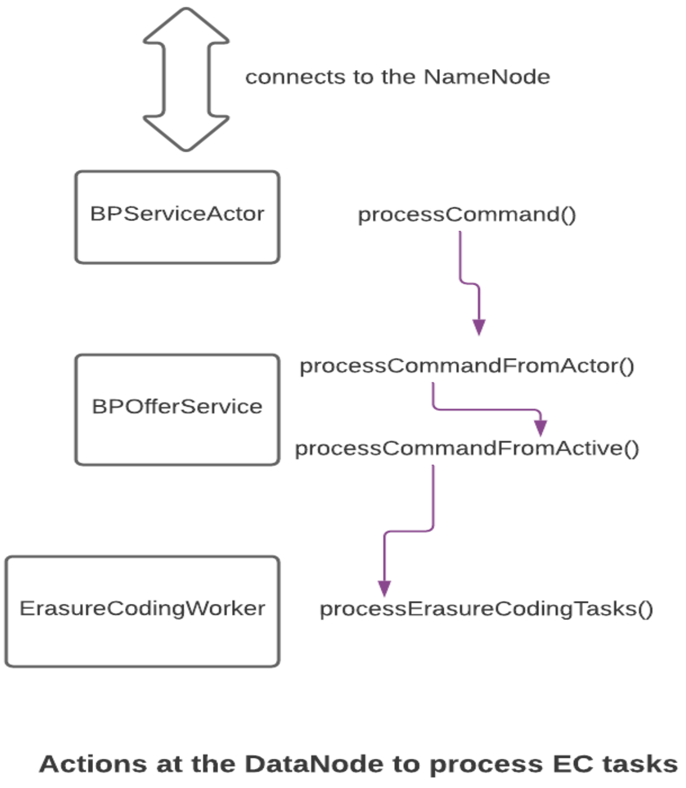

connects to the NameNode
BPServiceActor
processCommand()
BPOfferService
processCommandFromActor()
processCommandFromActive()
ErasureCodingWorker
processErasureCodingTasks()
Actions at the DataNode to process EC tasks

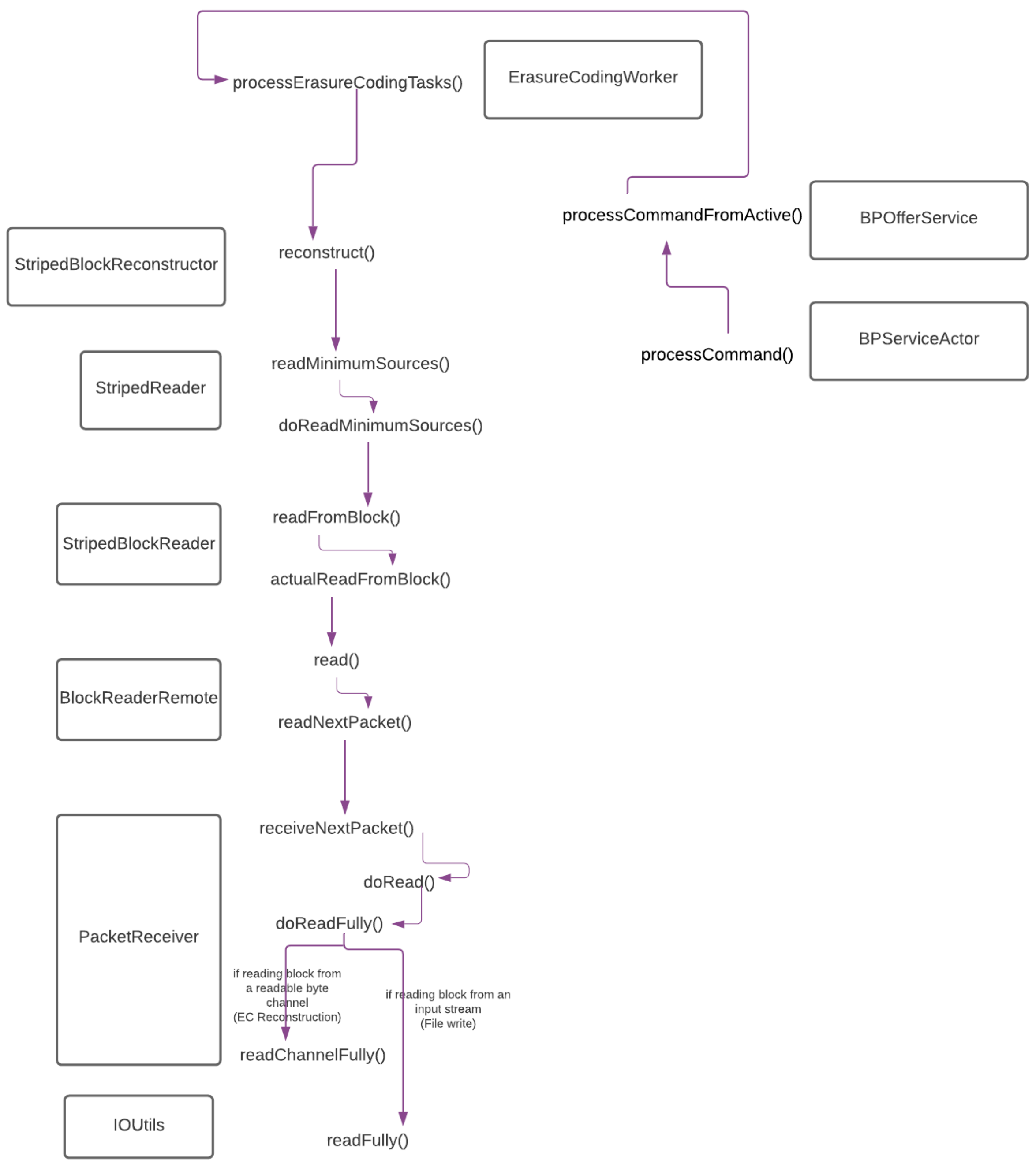
Fig. 2

BlockInfo class: for a given EC blockgroup, this class maintains the datanodes where the blocks belonging to this coding block group are stored.

BlockInfoStriped classes is a subclass of BlockInfo.

Reading blocks for reconstruction

processErasureCodingTasks() construtcs a **StripedBlockReconstructor** task by encapsulating **ErasureCodingWorker** and **StripedReconstructionInfo.**
It submits the EC task to a ThreadPool**,** which instantiates run() in **StripedBlockReconstructor.**



Process Flow of reading blocks during EC reconstruction at the DataNode

Fig. 3

Let us revisit packet structure and the ideas of block, chunk and packet.
Block: 128 MB (default)
Chunk: A block is divided into chunks, each chunk comes with a checksum (CRC). We want transfers to be chunk-aligned, to be able to verify checksums.
Packet: A grouping of chunks used for transport. It has the below structure.
Each packet looks like:

```
//   PLEN    HLEN      HEADER     CHECKSUMS  DATA
//   32-bit    16-bit       <protobuf>     <variable length>
//
// PLEN:      Payload length
//            = length(PLEN) + length(CHECKSUMS) + length(DATA)
//            This length includes its own encoded length in
//            the sum.
//
// HLEN:      Header length
//            = length(HEADER)
//
// HEADER:    the actual packet header fields, encoded in protobuf
// CHECKSUMS: the CRCs for the data chunk. May be missing if
//            checksums were not requested
// DATA:      the actual block data
```

How is an entire block read from a source node and sent to a recipient?

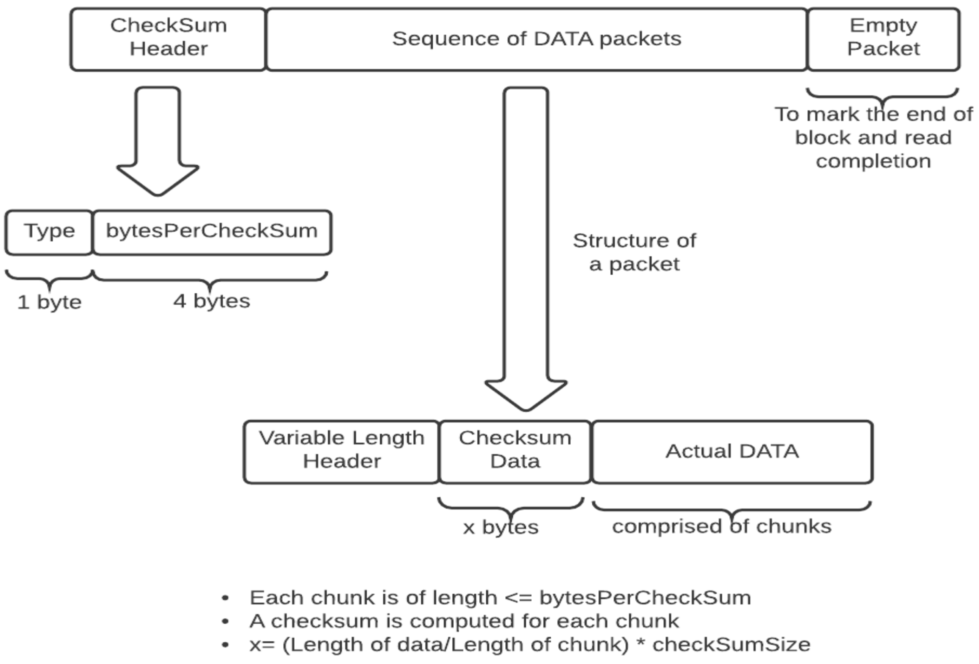

CheckSum Header
Sequence of DATA packets
Empty Packet
To mark the end of block and read completion
Type
bytesPerCheckSum
1 byte
4 bytes
Structure of a packet
Variable Length Header
Checksum Data
Actual DATA
x bytes
comprised of chunks
• Each chunk is of length <= bytesPerCheckSum
• A checksum is computed for each chunk
• x= (Length of data/Length of chunk) * checkSumSize

Fig. 4

Actions in the run() of **StripedBlockReconstructor** thread

1. Initialize decoder – to create a raw decoder using ecPolicy from **StripedReconstructionInfo** if not initialized already
2. Initialize **StripedReader**
   i. initialize readers – create a set of readers for reading from required source nodes
   ii. initialize bufferSize – allocate readbuffers of size 64KB (default) and make sure bufferSize is always a multiple of bytesPerChecksum (algorithms defined in **DataChecksum**)
   iii. initialize zeroStrip – allocate zero padded buffers for partial stripes
3. Initialize **StripedWriter** – sets chunkSize, maxChunksPerPacket, maxPacketSize and allocates buffers for packet and checksum (these buffers are used to send reconstructed data to target nodes)
4. Call reconstruct()
5. Send an empty packet to mark end of the block
6. Close **StripedReader** and **StripedWriter**

Method reconstruct() explained in detail

It makes 3 calls

1. readMinimumSources() in **StripedReader** (explained in detail in Fig. 3)
   (Once this call returns, all blocks required for reconstruction are read as packets and are available for the EC reconstruction task.)
2. reconstructTargets()
   - Constructs ByteBuffer arrays for inputs and outputs
   - Constructs integer array for erased indices
   - Calls decode() of **RawErasureDecoder**
   - Calls updateRealTargetBuffers() of **StripedWriter**
3. transferData2Targets() in **StripedWriter**

Method decode() explained in detail

It decodes with inputs and erasedIndexes and generates outputs.
Preparing for inputs is done as follows:
1. Create an array containing data units + parity units. Please note the data units should be first or before the parity units.
2. Set null in the array locations specified via erasedIndexes to indicate they're erased and no data are to read from;
3. Set null in the array locations for extra redundant items, as they're not necessary to read when decoding.

For example in RS-6-3, if only 1 unit is really erased, then we have 2 extra items as redundant. They can be set as null to indicate no data will be used from them.

For an example using RS (6, 3), assuming sources (d0, d1, d2, d3, d4, d5) and parities (p0, p1, p2), d2 being erased.
We can and may want to use only 6 units like (d1, d3, d4, d5, p0, p2) to recover d2.
We will have:

    inputs = [null(d0), d1, null(d2), d3, d4, d5, p0, null(p1), p2]
    erasedIndexes = [2] (index of d2 in inputs array)
    outputs = [a-writable-buffer]

Decode/Encode buffers can be **direct or non-direct buffers.**
For a direct byte buffer, the Java virtual machine will make a best effort to perform native I/O operations directly upon it. Using direct buffer is faster and more efficient. Direct byte buffer is outside of JVM and memory is not allocated from Java heap memory.

Non-direct byte buffers are just a wrapper around byte array and they reside in Java Heap memory and are subject to normal garbage collection.

DecodingState maintains the state of decoding during a decode call. It has two implementations: one for direct byte buffer and the other for byte array.
All implementations of encoder and decoder have functionality to handle both buffer inputs.

The process flow of decode() is explained in the below figure:

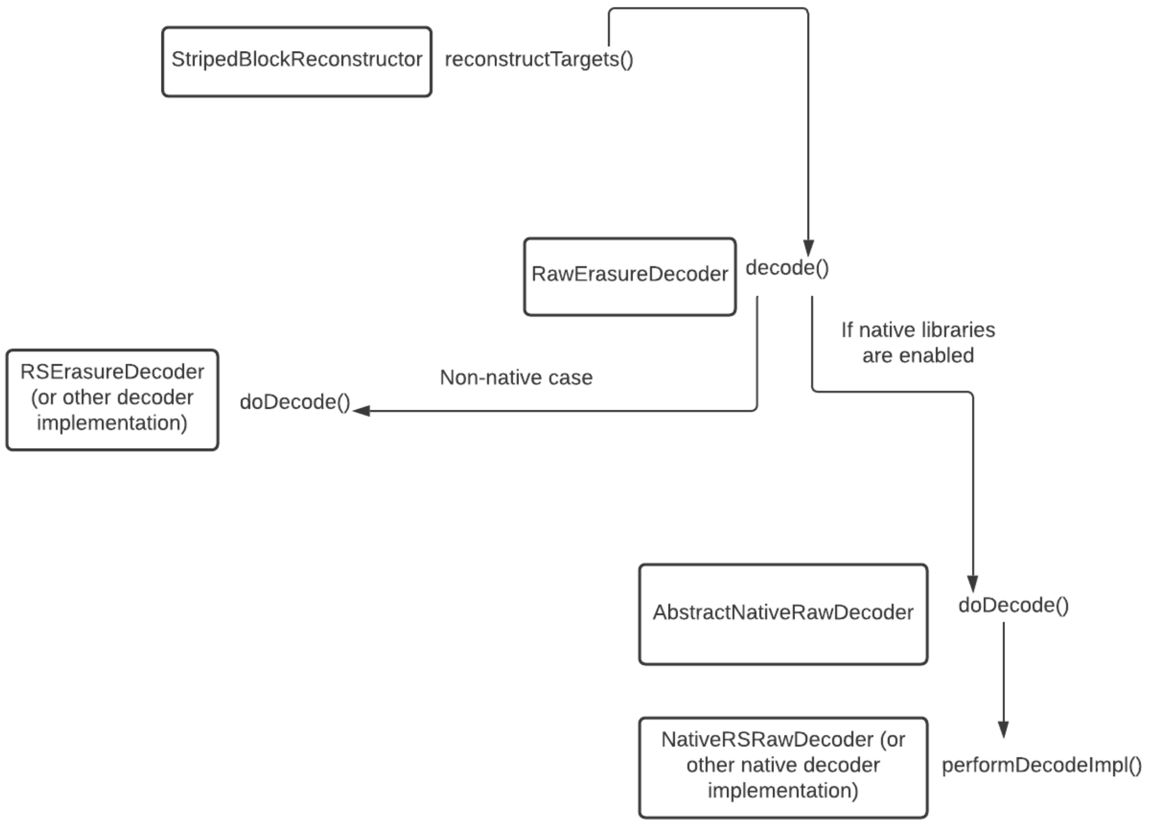

StripedBlockReconstructor
reconstructTargets()
RawErasureDecoder
decode()
If native libraries are enabled
RSErasureDecoder (or other decoder implementation)
doDecode()
Non-native case
AbstractNativeRawDecoder
doDecode()
NativeRSRawDecoder (or other native decoder implementation)
performDecodeImpl()

Fig. 5

After decode() call, the reconstructTargets() method calls updateRealTargetBuffers() in **StripedWriter**. This is to update the remaining bytes to be read for reconstruction, based on the position in the block after the current decode.

After reconstruction completes, the method transferData2Targets() of **StripedWriter** is called to send reconstructed data to target nodes. The writing of buffer contents to traget nodes takes places by constucting a packet of type **DFSPacket** and calling the methods writeCheckSum(), writeData() and writeTo().

Packages where the java classes are located:

Class Name	**Package**
NameNode	hadoop-hdfs/../org/apache/hadoop/hdfs/server/namenode
BlockManager	hadoop-hdfs/../org/apache/hadoop/hdfs/server/ blockmanagement
ErasureCodingWork	hadoop-hdfs/../org/apache/hadoop/hdfs/server/ blockmanagement
BlockInfoStriped	hadoop-hdfs/../org/apache/hadoop/hdfs/server/ blockmanagement
BlockInfo	hadoop-hdfs/../org/apache/hadoop/hdfs/server/ blockmanagement
DatanodeDescriptor	hadoop-hdfs/../org/apache/hadoop/hdfs/server/ blockmanagement
BlockECReconstructionInfo/ BlockECReconstructionCommand	hadoop-hdfs/../org/apache/hadoop/hdfs/server/protocol
BPServiceActor	hadoop-hdfs/../org/apache/hadoop/hdfs/server/datanode
BPOfferService	hadoop-hdfs/../org/apache/hadoop/hdfs/server/datanode
ErasureCodingWorker	hadoop-hdfs/../org/apache/hadoop/hdfs/server/datanode/ erasurecode
StripedBlockReconstructor	hadoop-hdfs/../org/apache/hadoop/hdfs/server/datanode/ erasurecode
StripedReader	hadoop-hdfs/../org/apache/hadoop/hdfs/server/datanode/ erasurecode
StripedWriter	hadoop-hdfs/../org/apache/hadoop/hdfs/server/datanode/ erasurecode
StripedBlockReader	hadoop-hdfs/../org/apache/hadoop/hdfs/server/datanode/ erasurecode
BlockReaderRemote	hadoop-hdfs-client/../org/apache/hadoop/hdfs/client/impl
PacketReceiver PacketHeader	hadoop-hdfs-client/../org/apache/hadoop/hdfs/protocol/ datatransfer
IOUtils	hadoop-common/../org/apache/hadoop/io
DataChecksum	hadoop-common/../org/apache/hadoop/util
RawErasureDecoder	hadoop-common/../org/apache/hadoop/io/erasurecode/ rawcoder
RSRawEncoder	hadoop-common/../org/apache/hadoop/io/erasurecode/ rawcoder
RSUtil	hadoop-common/../org/apache/hadoop/io/erasurecode/ rawcoder/util

GaloisField	hadoop-common/../org/apache/hadoop/io/erasurecode/rawcoder/util
GF256	hadoop-common/../org/apache/hadoop/io/erasurecode/rawcoder/util
CoderUtil	hadoop-common/../org/apache/hadoop/io/erasurecode/rawcoder/
RSRawDecoder	hadoop-common/../org/apache/hadoop/io/erasurecode/rawcoder/
AbstractNativeRawDecoder	hadoop-common/../org/apache/hadoop/io/erasurecode/rawcoder/
DecodingState/ ByteArrayDecodingState/ ByteBufferDecodingState	hadoop-common/../org/apache/hadoop/io/erasurecode/rawcoder/
DFSPacket	hadoop-hdfs-client/../org/apache/hadoop/hdfs/