Append/Hflush/Read Design

Hairong Kuang, Konstantin Shvachko, Nicholas Sze, Sanjay Radia, Robert Chansler
Yahoo! HDFS team
08/06/2009

1. Design challenges

With hflush, HDFS needs to make the last block of an unclosed file visible to readers. This presents two challenges:

1. Read consistency. At a given time different replicas of the last block may have different number of bytes. What read consistency should HDFS provide and how to guarantee the consistency even in case of failures.
2. Data durability. When any error occurs, the recovery cannot simply throw the last block away. Instead the recovery needs to preserve at least the hflushed bytes while maintaining the read consistency.

2. Replica/Block States

This document will call a block at a DataNode a replica to differentiate it from a block at the NameNode.

2.1. Need for new states

Pre-append/hflush a replica at a DataNode is either finalized or temporary. When a replica is first created, it is in the temporary state on DataNode. A temporary replica becomes finalized upon the close request of a client when no more byte will be written to this replica. On a DataNode restart, temporary replicas are removed. This is acceptable pre-append/hflush because HDFS provides best effort durability for under-construction blocks. This is not acceptable after append/hflush are supported. HDFS needs to support strong durability for under-construction blocks that contain pre-append data and best effort durability for hflushed data. So some temporary replicas need to be preserved across DataNode restarts.

2.2. Replica states (DataNode)

At a DataNode, this design introduces a replica being written (rbw) state and other states for handling errors. In a DataNode's memory, a replica could be in any of the following state:

Finalized: A finalized replica has finalized its bytes. No new byte will be written to this replica unless it is reopened for append. Its data and meta data match. The other replicas of the same block id have the same bytes as this replica. But the generation stamp (GS) of a finalized replica does not remain constant. It may be bumped up as a result of error recovery.

Rbw (Replica Being Written to): Once a replica is created or appended, it is in the rbw state. Bytes are being written to this replica. It is always a replica of the last block of an unclosed file. Its length is not finalized yet. Its on-disk data and meta data may not match. Other replicas of the same block id may have more or less bytes than this one. Bytes (may not all) in an rbw replica are visible to readers. In case of any failure, bytes in an rbw replica will try to be preserved.

Rwr (Replica Waiting to be Recovered): If a DataNode dies and restarts, all its rbw replicas change to be in the rwr state. Rwr replicas will not be in any pipeline and therefore will not receive any new bytes. They will either become outdated or will participate in a lease recovery if the client also dies.

rur (Replica Under Recovery): A replica changes to be in the rur state when a replica recovery starts as a result of lease expiration. More details will be discussed in the lease recovery section.

Temporary: a temporary replica is also a replica under construction but is created for the purpose of block replication or cluster balancing. It shares many of the properties as an rbw replica, but its data are invisible to any reader. If the replica construction fails or its DataNode restarts, a temporary replica will be deleted.

On a DataNode's disk, each data directory will have three subdirectories: *current* contains finalized replicas, *tmp* contains temporary replicas, *rbw* contains rbw, rwr, and rur replicas. When a replica is first created by a request from a DFS client, it is put in the rbw directory. When a replica is first created for the purpose of replication or cluster balancing, it is put in the tmp directory. Once a replica is finalized, it is moved to the current directory. When a DataNode restarts, the replicas in the tmp directory are removed, the replicas in the rbw directory are loaded as rwr replicas, and the replicas in the current directory are loaded as finalized replicas.

During DataNode upgrade, all replicas in directories *current* and *rbw* need to be kept in a snapshot.

2.3. Block states (NameNode)

NameNode also introduces many new states for a block. A block could be in any of the following state:

UnderConstruction: Once a block is created or appended, it is in the UnderConstruction state. Bytes are being written to this block. It is the last block of an unclosed file. Its length and GS has not finalized yet. Data (may not all) in a block under construction are visible to readers. A block under construction keeps track of its write pipeline (i.e., locations of valid rbw replicas) and the locations of its rwr replicas if the client dies.

UnderRecovery: When a file's lease expires, if the last block is UnderConstruction, it is changed to be UnderRecovery state once block recovery starts.

Committed: A committed block has finalized its bytes and generation stamp (GS), but has not seen at least one GS/length matched finalized replica from DataNodes yet. No new byte will be written to this block and its GS will not be bumped unless it is reopened for append. In order to serve read requests, a committed block still needs to keep the locations of rbw replicas. It also needs to track the GS and length of its finalized replicas. An under construction block of an unclosed file is committed when NN is asked by the client to add a new block to the file or close the file. If the last block is in the committed state, the file cannot be closed and the client has to retry. AddBlock and close will be extended to include the last block's GS and length.

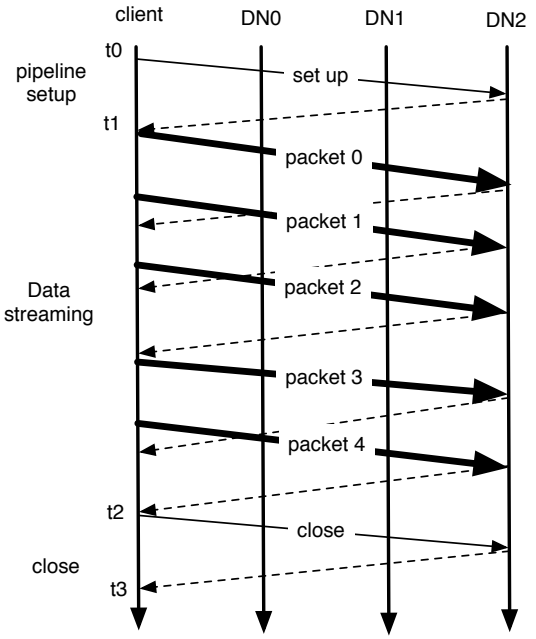
Complete: A complete block is a block whose length and GS are finalized and NameNode has seen a GS/len matched finalized replica of the block. A complete block keeps only finalized replicas' locations. Only when all blocks of a file become complete, a file could be closed.

Different from replica's states, a block's state does not persist on any disk. When NameNode restarts, the last block of an unclosed file is loaded as UnderConstruction. All the rest of the blocks are loaded as Complete.

More details about above replica/block states will be discussed in the rest of the document. A replica state transition diagram and a block state transition diagram will be summarized in the last section.

3. Write/hflush

3.1. Block Construction Pipeline



An HDFS file consists of multiple blocks. Each block is constructed through a write pipeline. Bytes are pushed to the pipeline packet by packet. If no error occurs, a block construction goes through three stages as shown in the following picture illustrated by a pipeline of 3 DataNodes (DN) and a block of 5 packets. In the picture, bold lines represent data packets, dotted lines represent ack messages, and regular lines represent control messages (setup/close). From t0 to t1 is the pipeline setup stage. T1 to t2 is the data streaming stage, where t1 is the time when the first data packet gets sent and t2 is the time that the ack to the last packet gets received. Note packet 2 is an hflushed packet. T2 to t3 is the pipeline close stage.

Stage 1. Set up a pipeline

A Write_Block request is sent by a client downstream along the pipeline. After the last DataNode receives the request, an ack is sent by the DataNode upstream along the pipeline back to the client. As a result of this, network connections along the pipeline are set up and each DataNode has created or opened a replica for writing.

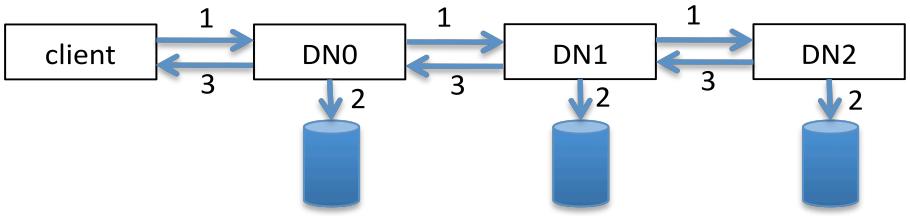
Stage 2. Data streaming

User data first buffer at the client side. After a packet is filled up, the data then get pushed to the pipeline. Next packet can be pushed to the pipeline before receiving the ack for the previous packet. The number of outstanding packets is limited by the outstanding packets window size at the client side. If the user application explicitly calls hflush, a packet is pushed to the pipeline before it is filled up. Hflush is a synchronous operation and no data can be written before an acknowledgement for the flushed packet comes back.

Stage 3. Close (finalize a block and shutdown pipeline)

The client sends a close request only after all packets have been acknowledged at the client side. This ensures that if data streaming fails, the recovery does not need to handle the case that some replicas have been finalized and some do not have all the data.

3.2. Packet handling at a DataNode



For each packet, a DataNode in the pipeline has to do 3 things.

1. Stream data
    a. Receive data from the upstream DataNode or the client
    b. Push the data to the downstream DataNode if there is any
2. Write the data/crc to its block file/meta file.
3. Stream ack
    a. Receive an ack from the downstream DataNode if there is any
    b. Send an ack to the upstream DataNode or the client

Note that the numbers above do not indicate the order that the three things must be executed in. Streaming ack (3) is done after streaming data (1) by the definition of a pipeline. But in theory writing data to disk (2) could be done anytime after 1.a. This algorithm chooses to do it right after 1.b and before receiving the next packet.

Each DataNode has two threads per pipeline. The data thread is responsible for data streaming and disk writing. For each packet, it does 1.a, 1.b, and 2 in sequence. Once a packet is flushed to the disk, it can be removed from the in-memory buffer. The ack thread is responsible for ack streaming. For each packet, it does 3.a and 3.b in sequence. Since the data thread and the ack thread run concurrently, there is no

guarantee on the order of (2) and (3). The ack of a packet might be sent before the packet is flushed to the disk.

This algorithm provides a tradeoff on the write performance, data durability, and algorithm simplicity. It

1. Improves data durability against failures by writing data to disk sooner than later when the ack is received;
2. Parallelizes data/ack streaming in downstream pipeline and on-disk writing;
3. Simplifies buffer management since there is at most one packet in-memory per pipeline.

3.3. Consistency support

- When a client reads bytes from an rbw replica, the DataNode that it reads from may not make all the bytes that it received visible to the client.
- Each rbw replica maintains two counters:
    1. BA: number of bytes that have been acknowledged by the downstream DataNodes. Those are the bytes that the DataNode makes visible to any reader. In the rest of the document, we may interchangeably call it the replica's visible length.
    2. BR: number of bytes that have been received for this block, including the bytes written to its block file and in-DataNode-buffer bytes.
- Assume initially all DataNodes in the pipeline have (BA, BR) = (a, a). Then a client pushes a packet of b bytes to the pipeline and no other packets are pushed to the pipeline before the client receives an ack for the packet.
    1. A DataNode changes its (BA, BR) to be (a, a+b) right after step 1.a.
    2. A DataNode changes its (BA, BR) to be (a+b, a+b) right after step 2.a.
    3. When a success ack is sent back to the client, all DataNodes in the pipeline have (BA, BR) = (a+b, a+b).
- A pipeline of N DataNodes DN^0, …, DN^i, …, DN^{N-1}, where DN^0 is the first in the pipeline, i.e., the closest to the writer, has the following property: at any given time *t*,

  $$BA_t^0 \le BA_t^i \le BA_t^{N-1} \le BR_t^{N-1} \le BR_t^i \le BR_t^0$$

  where BA_t^i is BA of the block at DN^i at time *t* and BR_t^i is BR of the block at DN^i at time *t*.

Note that this property guarantees that once a byte becomes visible all DataNodes in the pipeline has the byte.

- Assume BS_t^c is the number of bytes that a client has sent to the pipeline at time t and BA_t^c is the number of bytes that the client has received ack for. We have

  $$BA_t^c \le BA_t^0 \le \ldots \le BA_t^{N-1} \le BR_t^{N-1} \le \ldots \le BR_t^0 \le BS_t^c$$

4. Read

When an unclosed file is opened for read, the challenge is how to provide the consistency guarantee if the last block is under construction. The algorithm needs to make sure that a byte read at DataNode DN^i can also be read at another DataNode DN^j, even if $BA^i > BA^j$.

Algorithm 1:

- When a reader reads an under construction block, it first asks one of the replicas for its BA by sending a request to the DataNode.
- If an application tries to read a byte beyond BA of the block, the dfs client throws an EOFException.
- Only a read request that read from a position less than the visible length of the last block will be forwarded to a DataNode. When a DataNode gets a read request from a range of bytes that are less than its BR, return the bytes.
- Assume that a read request is a triple (blck, off, len), where blck contains a block id and its generation stamp, off is the starting offset in the block from which to read the block, and len is the number of bytes to read.
- A DataNode can serve the request if the DataNode has a replica with an equal or newer GS.
- The summation of off and len must be equal or less than BA^j, assuming DN^j is the DataNode where the dfs client fetched the block length.
- Assume that the read request is sent to DataNode DN^i and the replica's state is (BA^i, BR^i).
    1. If off+len<= BA^i, DN^i can safely send len bytes back to the dfs client starting at off.
    2. If off+len> BA^i, because off+len<=BA^j, BA^j>=BA^i. DN^i must be in the upstream in the pipeline to DN^j, i.e., is closer to the writer than DN^j is. So BR^i>= BR^j>=BA^j. Thus BR^i> BA^j, and therefore BR^i>off+len. This means DN^i must have the bytes that the dfs client wants to read. DN^i delivers the bytes to the client.
    3. Off+len should never be greater than BR^i. If this ever happens, the DataNode log the error and rejects the request.
- If DN^i goes down while serving the request, the dfs client can switch to read from any other DataNode containing a replica of the block.
- This algorithm is simple but it requires a reopen of a file to get new data because the length of the last block is fetched before read and a dfs client cannot read beyond the length of the last block.

Algorithm 2:

- This algorithm lets a dfs client, i.e., a reader, perform the consistency control, and DataNodes deliver bytes.
- A read request is a triple (blck, off, len), where blck contains a block id and its generation stamp, off is the starting offset in a block from which to read the block, and len is the number of bytes to read.

- A DataNode can serve the request if the DataNode has a replica with an equal or newer GS.
- Assume that the block has a state (BA^i, BR^i), DN^i sends bytes [off, MIN(off+len, BR^i)) to the client along with its BA^i.
- The client receives and buffers the data. It also keeps track of the maximum BA that it has seen and only delivers bytes to the application up to the maximum BA.
- If the read from DN^i fails, the dfs client can switch to read from any other DataNode containing a replica of the block.
- How read consistency is guaranteed?
  Assume we have a pipeline of N DataNodes DN^0, …, DN^i, …, DN^{N-1}, where DN^0 is the first in the pipeline. Assume that the number of bytes that a client delivers to an application at time *t* is BR_t^c. We have
  $$BR_t^c \le BA_t^{N-1} \le BR_t^{N-1} \le BR_t^i \le BR_t^0$$
  So no matter which DataNode it reads from, the DataNode should have the byte it read before.
- This algorithm requires a change of the read protocol and a dfs client is more complicated since it needs to control read consistency. But the algorithm does not require a reopen in order to read the new data.

For HADOOP 0.21, we are still discussing whether to implement algorithm 1 or algorithm 2.

5. Append

5.1. Append API support

1. Client sends an append request to NN.
2. NN checks the file and makes sure that it is closed. Then NN checks the file's last block. If it is not full and has no replica, fail append. Otherwise, change the file to be under construction. If the last block is full, NN allocates a new last block. If the last block is not full, NN changes this block to be an under construction block, with its finalized replicas as its initial pipeline. It returns the block id, generation stamp, length, and its locations. If the last block is not full, it also needs to return a new generation stamp.
3. Set up a pipeline for append if last block is not full. Otherwise set up a pipeline for create. Check pipeline set up section for more details.
4. If the last block does not end at a checksum chunk boundary, read the last partial crc chunk. This is for the purpose of calculating checksums.
5. The rest is the same as a regular write.

5.2. Durability support

- NN makes sure that the number of replicas for the Complete blocks that contain pre-append data meets the file's replication factor.

- The durability of an UnderConstruction block that contains pre-append data is omitted in this design for now.

6. Error Handling

6.1. Pipeline Recovery

When a block is under construction, error may occur at Stage 1 when a pipeline is being set up, at Stage 2 when data are streaming to the pipeline, or Stage 3 when the pipeline is being closed. The pipeline recovery handles the case when one of DataNodes in the pipeline has an error.

6.1.1. Recover from pipeline setup failure

If a DataNode detects a failure when a pipeline is being set up, a DataNode closes the block file and closes all its tcp/ip connections after a failure acknowledgement is sent to the upstream DataNode. Once the client detects the failure, it handles the failure differently depending on the purpose of setting up the pipeline.

- If the pipeline was built for creating a new block, the client simply abandons the block and asks NameNode for a new block. It then starts to build a pipeline for the new block.
- If the pipeline was built for appending to a block, it rebuilds a pipeline with the remaining DataNodes and bumps the block's generation stamp. See section 7 (pipeline setup) for more details.

One special case of pipeline setup failures is access token error: one of the DataNode complains that the access token is invalid when using an access token to set up a pipeline. If a pipeline set up failure is caused by an expired access token, the dfs client should rebuild the pipeline with all the DataNodes in the previous pipeline. Current trunk (0.21) avoids this special handling by always fetching a new access token from NameNode right before setting up a pipeline. This document will keep the same design.

6.1.2. Recover from data streaming failure

- At a DataNode an error may occur at either 1.a, 1.b, 2, 3.a, or 3.b as explained in Section 3.2. Whenever an error occurs, a DataNode takes itself out of the write pipeline: it closes all the tcp/ip connections, writes all buffered bytes onto disk if the error does not occur at 3, and closes the on-disk files.
- When the dfs client detects a failure, stops sending data to the pipeline.
- The dfs client reconstructs a write pipeline using the remaining DataNodes. See section 7 (pipeline setup) for more details. As a result of this, all replicas of the block are bumped to a new generation stamp.
- The dfs client resumes sending data with the new generation stamp starting from BA^c. Note an optimization could be that the client resumes sending bytes starting at MIN(BR^i, for all DataNodes DN^i in the new pipeline).
- When a DataNode receives a packet, if it already has the packet, the data stream simply pushes the data downstream without writing it to the disk.

This recovery algorithm has a nice property: any bytes that were visible to any client, even from a down DataNode with the largest BA of the old pipeline, continue to be visible to any reader during and after a pipeline recovery. This is because the pipeline recovery does not decrease any DataNode's BA and BR. Furthermore any time during the pipeline recovery the new pipleline maintains the property described in section 3.3 (consistency support).

6.1.3. Recover from a close failure

Once the client detects the failure, it rebuilds a pipeline with the remaining DataNodes. Each DataNode bumps the block's generation stamp and finalizes the replica if it is not finalized yet. The network connection is torn down after an ack is sent. See section 7 (pipeline setup) for more details.

6.2. DataNode Restart

- When a DataNode restarts, it reads each replica under directory *rbw* and loads the replica in memory as WaitingToBeRecovered. Its length is set to be the maximum number of bytes that match its crc.
- Any WaitingToBeRecovered replica does not serve any read and does not participate in a pipeline recovery.
- A WaitingToBeRecovered replica will either become outdated and be deleted by NN if the client is still alive or be changed to be finalized as a result of lease recovery if the client dies.

6.3. NameNode Restart

- None of block states are persisted on disk. So when NameNode restarts, it needs to restore each block's state. The last block of an unclosed file becomes UnderConstruction no matter what its pre-life state was. Other blocks become Complete.
- Ask each DataNode to register and send its block report including finalized, rbw, rwr, and rur replicas.
- NameNode does exit safemode unless the number of complete and under construction blocks that have received at least one replica reaches the pre-defined threshold.

6.4. Lease Recovery

When a file's lease is expired, NN needs to close the file for the sake of the client. There are two issues: (1) Concurrency control: what if a lease recovery is performed while the client is still alive either in the process of setting up pipeline, writing, close, or recovery. What if there are multiple concurrent lease recoveries? (2) Consistency guarantee: If the last block is under construction, all its replicas need to roll back to a consistent state: all replicas have the same on-disk length and the same new generation stamp.

1. NN renews lease, changes the file's leaseholder to be dfs and persists the change to its editlog. So if the client is still alive, any of the write-related requests like asking for a new generation stamp, getting a new block, or closing the file, will be rejected because the client is not the lease holder any

more. This prevents the client from concurrently changing an unclosed file if it ever contacts the NameNode.

2. NN checks the state of the last two blocks of its file. Other blocks should be in the complete state. The following table shows all the possible combinations and the action to take for each combination.

Penultimate block	Last block	Actions
Complete	Complete	Close the file
Complete	Committed	Retry closing the file when lease expires next time; Force to close the file after a certain number of retries
Committed	Complete	
Committed	Committed	
Complete	UnderConstruction	Starts block recovery for the last block
Committed	UnderConstruction	
Complete	UnderRecovery	Starts a new block recovery for the last block; stop recovery after a certain number of retries
Committed	UnderRecovery	

6.5. Block Recovery

1. NN chooses a primary DataNode (PD) to work as the proxy of NameNode to perform block recovery. PD could be a DataNode where one of its replicas resides. If none of its replicas are known, block recovery aborts.
2. NN gets a new GS, which marks the generation that the block is going to be bumped to when the recovery successfully finishes. It then changes the last block, if it is UnderConstruction, to be UnderRecovery. The UnderRecovery block is stamped with a unique recovery id, which is new GS that the block is going to be bumped to. Any communications from a PD to NN needs to match this recovery id. This is how concurrent block recoveries are handled. The basic rule is that the latest recovery always preempts previous recoveries.
3. NN then asks PD to recover the block. NN sends PD the new GS, block id and its generation stamp, and all its replica locations including finalized replicas, replicas being written to, and replicas waiting to be recovered.
4. PD performs block recovery:
    a. PD asks each DataNode, where one replica is located, to perform **replica recovery**.
        i. PD sends each DataNode the recovery id, block id and generation stamp;
        ii. Each DataNode checks its replica state:
            1. Check existence: If the DataNode does not have the replica or the replica is older than the block's GS in the request, or newer than the recovery id (this is not supposed to happen), throws a ReplicaNotExistsException.
            2. Stop writer: If it is a replica being written to and there is an ongoing writer thread, interrupts the writer and

waits for the writer to exit. When a writer thread is interrupted, if the thread is in the middle of receiving a packet, stops and throws away the partial packet. Before the thread exits, it makes sure that on-disk bytes are the same as BR and then closes the block and crc files. This handles concurrent client writes and block recovery at DataNodes. Block recovery preempts client writes, resulting in pipeline failure. Subsequent pipeline recovery will fail because the dfs client cannot get a new generation stamp from NN for a block under recovery.

3. Stop previous block recovery: If the replica is already in the rur state, throws a RecoveryInProgressException if its recovery id is greater than or equal to the new recovery id. If the new GS is greater, stamp the rur replica's recovery id to be the new one.
4. State change: Otherwise, change the replica to be rur. Set its recovery id to be the new recovery id and a reference to its old state. Any communications from a PD to itself needs to match this recovery id. Note 3 and 4 handle concurrent block recoveries at DataNodes. The latest recovery always preempts previous recovery and no two recoveries can be interleaved.
5. Crc check: Then perform a CRC check for the block file. If there is a mismatch, throws CorruptedReplicaException if the replica is rbw or finalized. If replica is rwr, truncate the block file to the last matched byte.

iii. If no exception is thrown, each DataNode returns PD its replica status <replica id, replica GS, replica on-disk len, pre-recovery state>.

b. After receiving a reply from each DataNode, PD decides the block length that all replicas should agree on.
   i. If one DataNode throws RecoveryInProgressException, PD aborts block recovery.
   ii. If all DataNodes throw an exception, aborts block recovery.
   iii. If max(Len_i for all reported DN_i) ==0, asks NN to remove this block.
   iv. Otherwise, check returned state of the replicas with non-zero length. The following table shows all the possible combination of states in an example of two replicas and the length to agree on for each combination.

Cases	Replica 1 state	Replica 2 state	Length to be agreed on
1	Finalized	Finalized	Two replicas should have the same length; If not the same,

			there is an error, logs it and aborts block recovery
2	Finalized	rbw	Two replicas should have the same length because the client must have died when pipeline is being set up or torn down. If they are not the same, exclude the rbw replica.
3	Finalized	rwr	Set new length to be the length of the finalized replica and exclude the rwr replica.
4	rbw	rbw	Set new length to be MIN(len_1, len_2) where len_i is the length of replica i.
5	rbw	rwr	Exclude rwr replica. This becomes the same as case 4.
6	rwr	rwr	Set the new length to be MIN(len_1, len_2). In this case, len_i may not equal to BRi, so no guarantee of visible bytes since both DataNodes died.

c. Recover replicas that participated in length agreement in step b.iv.
    i. PD asks each DataNode to recover the replica. PD sends the block id, new GS, new length.
    ii. If the DataNode does not have the replica in the rur state or its recovery id does not match the new GS, fail the replica recovery at the DataNode.
    iii. Otherwise, the DataNode changes the replica's GS to be the new GS both on disk and in memory. It then updates in memory replica length to be the new length and truncates the block file size to the new length and change crc file accordingly (may cause truncation and/or modification of last 4 crc bytes). It finalizes the replica if the replica has not finalized yet. The replica recovery succeeds.
d. PD checks the result of c. If no DataNode succeeds, block recovery fails. If some succeed and some fail, PD gets a new generation stamp from NN and repeats block recovery with the successful DataNodes. If all DataNodes succeed, PD notifies NN the new GS and length. NN finalizes the block and closes the file if all blocks of the file change to Complete state. NN forces the file to close after a limited number of close retries.

This lease recovery algorithm also guarantees that any bytes that were visible to a client does not get removed as a result of the recovery if at least one DataNode in the pipeline is still alive and its data are not corrupted. This is because

1. In cases 1, 2, and 3, there is a finalized replica. The client must have died away during block construction stages 1 and 3. The algorithm does not remove any byte.
2. In cases 4 and 5, all replicas to be recovered are in rbw state. The client must have died during block construction stage 2. Assume the pre-recovery pipeline has N DataNodes: DN_0, DN_1, …, DN_{N-1}. The length returned by DN^i step 4.a.ii must be equal to BR^i. Assume that a subset of the DataNodes S in the pipeline participates the length agreement, the new length is MIN(BR^i, for all DataNodes in S) >=BR^{N-1}>=BA^{N-1}>=..>=BA^0. This guarantees the lease recovery does not remove data that have delivered to any reader.
3. In case 6, the algorithm does not provide any guarantee since all DataNodes in the pre-recovery pipeline had been restarted.

7. Pipeline Set Up

7.1. Causes of pipeline set up

There are five cases that a pipeline needs to be set up:

1. Create: When a new block is created, a pipeline needs to be constructed before any bytes are streamed to any DataNode.
2. Append: When a file is to be appended and the last block of the file is not full. A pipeline of all DataNodes that have a replica of the last block needs to be set up before any new bytes are streamed to any DataNode.
3. Append recovery: When case 2 fails, a pipeline containing the remaining DataNodes needs to be set up.
4. Data streaming recovery: If data streaming fails, a pipeline of the remaining DataNodes needs to be set up before the data streaming resumes.
5. Close recovery: If pipeline close fails, a pipeline of the remaining DataNodes needs to be set up in order to finalize the block.

7.2. Pipeline set up steps

1. Cases 2, 3, 4, and 5 build a pipeline on an existing block, so the block's generation stamp needs to be bumped along with pipeline construction. The dfs client asks NN for a new generation stamp.
2. The dfs client sends a write block request to the DataNodes in the new pipeline with the parameters (not inclusive): block id with old generation stamp, block length (number of bytes a replica must have or at least have), max replica length, flags, and/or a new generation stamp.

Cases	Block id /generation stamp	Block len	flags	New generation stamp	Max block len
1 (create)	yes	0	No flag is set	no	no
2 (Append)	yes	Pre-append block len	Append flag is set	yes	no

3 (Append Recovery)	yes	The same as 2	Append and recovery flags are set	yes	no
4 (Data streaming recovery)	yes	BA^c	Recovery flag is set	yes	BS^c
5 (Close recovery)	yes	$BA^c=BS^c$	Close flag is set	yes	no

3. The following table shows the behavior when a DataNode receives a pipeline set up request. Note that rwr replicas do not participate in the pipeline recovery. We can relax this restriction with some special handling of the rwr replicas. But since this is a very rare case, we choose not to do it in this round of design.

Cases	Sanity Check	Replica state change	GS change
1 (Create)	A replica with the same block id should not exist	Create a rbw replica with (BA, BR)=(0,0)	no
2 (Append)	Finalized replica; its on-disk len should match the pre-append len	Open the replica for write; set the write stream offset at the end of file; the replica becomes rbw: (BA,BR)=(preAppendLen, preAppendLen)	Set to new GS
3 (Append Recovery)	Finalized or rbw; replica length (on-disk or BR) should match pre-append len; rbw replica GS could be the same or newer	If finalized, do the same as above; If rbw, wait for writer to exit; open block for write; set write stream offset at the end of file.	Set to new GS
4 (Data Streaming Recovery)	rbw replica; the same or newer GS; $BA^c <= BA^i <= BR^i <= BS^c$	Wait for writer to exit; Open block for write; set write stream offset at the end of file.	Set to new GS
5 (Close Recovery)	rbw or finalized; the same or newer GS; replica length should be the same as BA^c	If rbw, wait for writer to exit and then finalize the replica; Close pipeline when ack is sent back.	Set to new GS

4. In cases 2, 3, and 4, on a successful pipeline setup, the dfs client notifies NN of the new GS, min length, and the DataNodes in the new pipeline. NN

then updates the under construction block's generation stamp, len and locations.

5. If the pipeline set up fails, if at least one DataNode remains, go to step 1 with a recovery flag set. Otherwise, since the pipeline has no more DataNode, mark this pipeline as failed. If a user application is blocked in hflush/write, it will be unblocked and get an EmptyPipelineException. Otherwise the next write/hflush/close will get an EmptyPipelineException immediately.

8. Report Replica/Block State/Meta Information Change to NN

8.1. Client Reports

A client informs NN of an under construction block's meta information change or state change.

As discussed in the pipeline set up section, in cases 2(append), 3(append recovery), and 4(data streaming recovery), after a new pipeline is set up, a client reports NN of the block's new GS and the DataNodes in the pipeline. NN then updates the under construction block's GS, length, and locations.

Note that in this design after a pipeline for creating a block is set up, a client does not report NN of the newly created block and its locations. Instead when the client issues addBlock/append to ask for a new block, NN puts the new block and its locations into the blocksMap before NN returns the block and locations back to client. This design has a minor flaw. If a new reader happens to read the last block between the time the block is added to blocksMap at NameNode and the time a replica of the block is created on a DataNode, the reader may get a "block does not exit" error. But since the chance of having a reader during this short time frame is very slim, we consciously make this design decision to trade for performance. A client does not need to send a notification to NN after a pipeline is set up for every block create.

When a client issues addBlock or close (a file), NN will finalize the last block's GS and length. The last block may move to the Complete state if the last block already has a GS/len matched finalized replica; otherwise the last block is moved to the Committed state. In addition, if the number of the replicas of the last block is less than its replication factor, NN explicitly replicates the block to reach its replication factor.

8.2. DataNode Reports

A DataNode reports a replica's meta information or state change by periodically sending NN a block report or sending NN a blockReceived message when a rbw replica is finalized.

8.3. Block Reports

- Each block report contains two lists: one for finalized replicas, one for rbw. The finalized replicas list include finalized replicas and under recovery replicas whose old state are finalized. The rbw replica list includes rbw replicas, rwr replicas, and rur replicas whose old state are not finalized. An rbw replica's length is its bytes received (BR). The length of an rwr is a negative number.
- Now each reported replica's state is a quadruple <DataNode, blck_id, blck_GS, blck_len, isRbw>.
- After NN receives a block report, compare it with what's in the memory and generate 3 lists: (do we need a updateStateList?)
    - deleteList if blck_id is not valid, i.e. no entry in blocksMap or belongs to no file.
    - addStoredBlockList if NN does not have the replica <DataNode, blck_id> but the block report has it.
    - rmStoredBlockList if NN has the replica <DataNode, blck_id> but the block report does not have it.
    - updateStateList if the replica's state in NN is rbw but the block report says it is finalized.

  To avoid race condition between client reports and DataNode reports, rbw replicas are added to only deleteList or addStoredBlockList.
- Add a new replica
    1. Block in NN is Complete
        - If the reported replica is finalized,
            - If its GS and length are different from NN recorded value, add it to the blocksMap but mark it as corrupt.
            - Otherwise, add the replica.
        - If the reported replica is rbw,
            - If the file is closed, if the replica's GS/len is different from NN–recorded value or the block has reached its replication factor, instruct its DataNode to delete it.
            - Otherwise, do nothing.
    2. Block in NN is Committed
       The handling is very similar to the above case except that if the reported replica is finalized and matches the block's GS and length, NN changes the block to the complete state.
    3. Block in NN is UnderConstruction or UnderRecovery
        - If the reported replica is finalized and the replica's GS is equal to or newer than NN recorded GS, add the replica. Also mark the replica as finalized and keeps track of its length and GS.
        - If the reported replica is rbw and the replica is valid (GS not older and length not shorter), add the replica. If the reported replica is rbw, mark it as rbw; otherwise, mark it as rwr.
        - Otherwise ignore it.

- Update a replica's state
  When a block report shows a replica is changed from rbw to finalized, if the block is under construction, NN marks the NN stored replica as finalized and keeps track of the finalized replica's GS and length. If the block is Committed, if the finalized replica matches the block's GS and length, NN changes the block to the Complete state; otherwise remove this replica from NN.

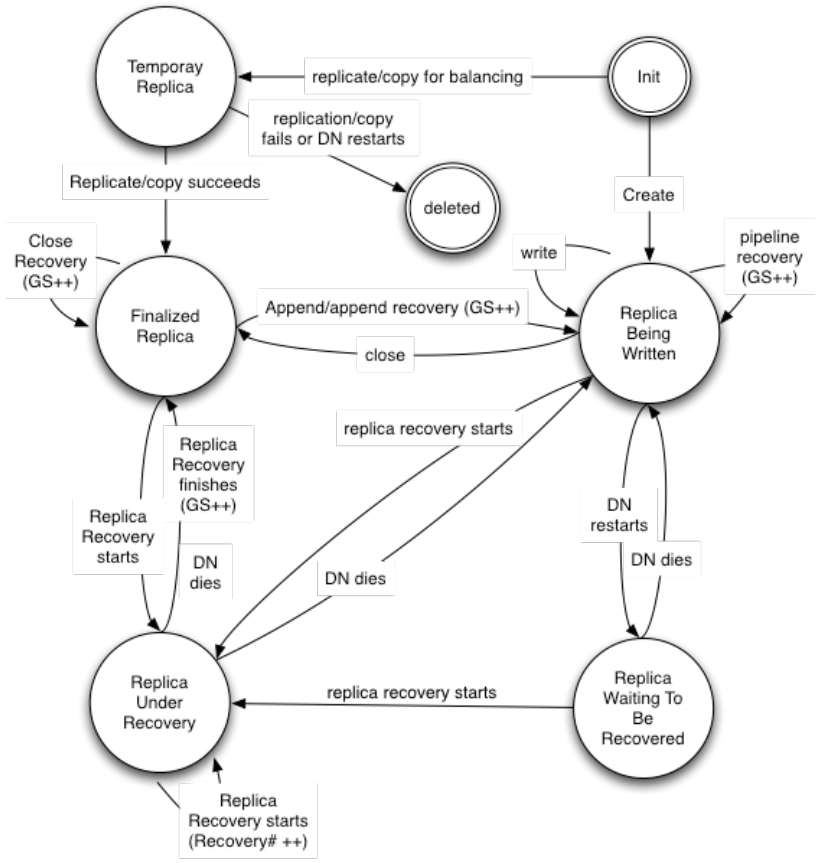
8.4. blockReceived

DataNodes send blockReceived to NN to notify that a replica is finalized. When NN receives a blockReceived notification, it either add a new replica if (DataNode, block_id) does not exist in NN, update the replica's state if it is recorded at rbw, or ask a DataNode to delete the replica if the block is invalid.

9. Replica/Block State Transition

9.1. Replica State Transition

The following diagram summarizes all possible replica state transitions at a DataNode.



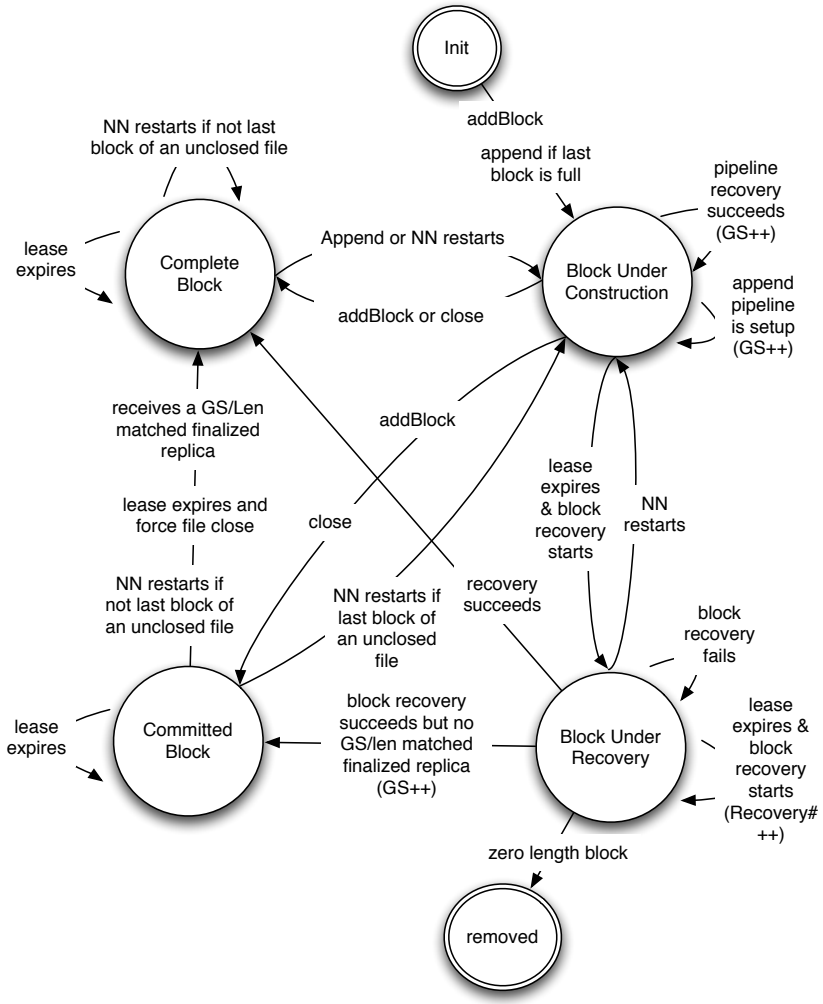
- A new replica is created

- Either by a client. The new replica start with a replica being written (rbw) state.
- Or upon an instruction from NN to replicate or copy a replica for the purpose of balancing. The new replica is in temporary state.

- An rbw replica changes to be a replica waiting to be recovered (rwr) when its DataNode restarts.
- A replica changes to be a under recovery replica when a replica recovery starts in response to lease expiration.
- A replica is finalized when a client issues a close, replica recovery succeeds, or replication/copy succeeds.
- Error recovery always causes a replica's GS to be bumped.

9.2. Block State Transition

The following diagram summarizes all possible block state transitions at the NameNode.



- A block is created
  - Either when a client issues addBlock to add a new block to a file.
  - Or when a client issues append and the last block of the file is full.

  The newly created block is a block under construction.

- Append may also cause a Complete block to be changed to a block UnderConstruction if the last block is partial.
- When addBlock or close is issued,
    - the last block becomes either Complete if the block already has a GS/len matched finalized replica or Committed otherwise.
    - addBlock waits until the penultimate block to become Complete.
    - A file won't be closed until the last two blocks of the file are Complete.
- When lease expires, a lease recovery changes a block under construction to be a block under recovery.
    - A block recovery may change a block under recovery to be
        - Removed if all its replicas are of length 0;
        - Committed if the recovery succeeds and the block has no GS/len matched finalized replica;
        - Complete if the recovery succeeds and the block has a GS/len matched finalized replica.
    - A lease recovery may force a Committed block to be Complete.
- Block states do not persist on disk. When a NameNode restarts, the last block of an unclosed file becomes under construction and the rest become Complete.
    - Note a Complete or Committed block may change to be an UnderConstruction block after NN restarts if it is the last block of a file. If the client is still alive, the client will finalize it again. Otherwise when lease expires, a block recovery will finalize it again.
- Note that once a block becomes Committed or Complete, all its replicas should have the same GS and are finalized. When a block is UnderConstruction, it may have multiple generations of the block coexisting in the cluster.