# Trinity File System (TFS) Specification V0.8

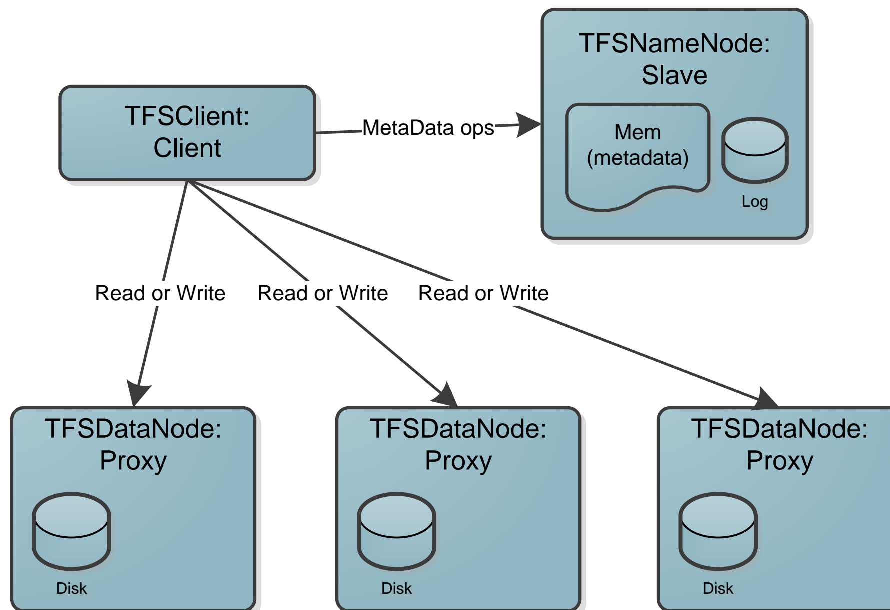
**Jiaran Zhang (v-jiarzh@microsoft.com), Bin Shao (binshao@microsoft.com)**

## 1. Introduction

Trinity File System (TFS) is a distributed file system designed to run on machines inside the local network. TFS is a HDFS[1] clone on Windows (R). It depends on Trinity.Core, especially Trinity's message passing mechanism. TFS provides high throughput access to data and achieves fault tolerance on commodity hardware.

## 2. Deployment

TFS adopts the master/slave architecture. TFSNameNode based on Trinity Slave is the master of TFS while TFSDataNode dependent on Trinity Proxy is the slave of TFS. As Trinity does, we should use the command *edit_config* or modify the file *trinity.xml* to configure the Trinity/TFS cluster.




Apart from that, some TFS parameters could be set in the class *Trinity.TrinityConfig*. We pick out some important members to demonstrate.

a. REPLICA_NUM: the replica number of blocks

[1] HDFS Architecture Guide. http://hadoop.apache.org/docs/stable/hdfs_design.html

b. BLOCK_SIZE: the size of blocks (default 64MB)
c. TRANSFERRED_BLOCK_SIZE: the size of data transmission message (default 1MB)
d. …

After the configuration, the application should be deployed to multiple machines by the command *cluster_deploy*.

## 3. Startup

In the normal startup, we should start all TFSDataNodes firstly by command *start TrinityFileSystem.exe –p*. If TFS are deployed on different nodes but under the same path, we can be benefit from the command *cluster_exec -p XXX\TrinityFileSystem.exe –p*.

When all TFSDataNodes are ready, TFSNameNode starts. We could start TFSNameNode by the command *start TrinityFileSystem.exe –s*.

Some details about the startup are listed as follows:

a. All TFSDataNodes start firstly, but do nothing. (In fact, the proxy would try to connect the slave.)
b. TFSNameNode starts and enters the safe mode. During the safe mode, TFS rejects all client requests.
    a. Trinity heartbeat mechanism works.
    b. Maybe it takes a long time to deal the log.
    c. TFSNameNode begins to count down to exit the safe mode.
    d. TFSNameNode sends the message to each TFSDataNode.
c. When TFSDataNodes receive the message, they send their block reports back.
    a. Regular block report mechanism works.
d. If TFSNameNode receives the reports from all the TFSDatanodes, it would exit the safe mode immediately and TFS could response to client request. Otherwise, it would exit the safe mode after 30s (it can be configured at *Trinity.TrinityConfig*), those TFSDataNodes, whose reports are not received, are regard as the "dead" nodes.

If we want to get a "clean" file system, we can clear the historical data under the directory *TFSData* of TFSNameNode and each TFSDataNode. Or we can just clear TFSNameNode and TFS would delete other unwanted data later.

## 4. Shutdown

We just need kill TFSNameNode. All TFSDataNodes would exit automatically if they cannot connect with TFSNameNode after a time. Never kill any TFSDataNode firstly. TFSNameNode would treat it as a "dead" node and exclude it from the cluster.

## 5. Recovery

If one TFSDataNode has a failure and has been fixed, we can rejoin this node into the cluster by the command *start TrinityFileSystem.exe –n*. This command is different from the previous. It uses "-n" instead of "-p". Under this situation, TFSDataNode would send the report on its own initiative rather than wait for TFSNameNode's message.

If TFSNameNode crashes, all the TFSDataNodes would exit automatically. We just need start the cluster up again to recover the data.

## 6. Interface

All the TFS entities and operations for users are in the namespace *Trinity.IO.Client*. There are two public classes so far. *TFSFile* presents the file stored in TFS while *TFSDirectory* collects some methods about the directory.

```
public class TFSFile
{
    public TFSFile(string path, TFSFileAccess access);
    public void Close();
    public long Position;
    public long Seek(long position, TFSSeekOrigin seekOrigin);
    public int Read(byte[] buffer, int offset, int count);
    public void Write(byte[] buffer, int offset, int count);
    public void ParallelWrite(byte[] buffer, int offset, int count);
    public static bool RemoveFile(string path);
    public static bool Exists(string path);
    public static long GetFileSize(string path);
}
public enum TFSFileAccess : byte
{
    Read, Write, Append
}

public enum TFSSeekOrigin : byte
{
    Begin, Current, End
}
public class TFSDirectory
{
    public static List<string> GetPathList();
    /// <summary>
    /// Test function for developers.
    /// It prints all the file paths, all the blocks and their TFSDataNodes.
    /// </summary>
    public static void Details();
}
```

TFS supports random read and append write. You cannot use *Seek* in the write or append access mode. If you violate this rule, *TFSFile* would throw an exception. TFS allows multiple readers but only one writer on one file at any time.

The difference between *Write* and *ParallelWrite* is illustrated in the chapter High Throughput.

## 6.1 Reading Data

A reading data sample:

```
void read()
{
    TFSFile readFile = new TFSFile(@"\a", TFSFileAccess.Read);
    FileStream writeLocalFile = new FileStream("A - 1", FileMode.OpenOrCreate);
    byte[] buffer = new byte[10 * 1024 * 1024];
    int size = 0;
    while ((size = readFile.Read(buffer, 0, buffer.Length)) != 0)
    {
            writeLocalFile.Write(buffer, 0, size);
    }
    writeLocalFile.Close();
    readFile.Close();
}
```

## 6.2 Writing Data

A writing data sample:

```
void write()
{
    FileStream readLocalFile = new FileStream("A", FileMode.Open);
    TFSFile writefile = new TFSFile(@"\a", TFSFileAccess.Write);
    byte[] buffer = new byte[10 * 1024 * 1024];
    int size = 0;
    while ((size = readLocalFile.Read(buffer, 0, buffer.Length)) != 0)
    {
            writefile.Write(buffer, 0, size);
    }
    readLocalFile.Close();
    writefile.Close();
}
```

# 7. Fault Tolerance

TFS achieves fault tolerance on commodity hardware. We summarize this character as the below list:

a. Any TFSDataNode's failure does not affect data validity and system availability. TFS can handle this failure in the background automatically. If the broken TFSDataNode has been fixed, it can be rejoined to TFS cluster.
b. TFSNameNode can recover from unpredictable breakdown. It will take some time to handle the logs. The recovery time lies on the size of the log file.
c. Write is an atomic operation. Those unsuccessful writes are usually caused by the disconnected connections between TFS and clients or users' wrong usage on TFS API. TFS will roll back them.

## 8. Data Integrity & Replication

Each file stored on TFS is divided into fixed-size block besides the tail. These blocks are distributed stored on different TFSDataNodes. Each block would exist as a single file with a GUID (e.g. 4cc8fccb-863c-4466-ada9-f57c481d0942) as its file name. To validate data, every block has a checksum file of the same name (e.g. 4cc8fccb-863c-4466-ada9-f57c481d0942.checksum). Every 1MB data unit in the block file owns a 16-byte (128-bit) MD5 checksum in the corresponding checksum file. When a client reads data, it also gets the checksum. The corrupted data can be prevented by this transparent verification.

To achieve fault tolerance on commodity hardware, TFS stores multiple replications for each block. TFSNameNode keeps a mapping from the blocks to their TFSDataNodes in the memory. TFSDataNode sends the block report, which contains of a list of all the blocks on this node, to TFSNameNode periodically. On the basis of these block reports, TFSNameNode can be master of the replica number to match the configured figure.

## 9. High Throughput

Since files are divided into blocks and stored on different TFSDataNodes, TFS tries to support parallel operation involved multiple TFSDataNodes. In the reading data, multiple blocks of one file are read parallel from different nodes. About the writing data, multiple replicas of one block are written to different nodes parallel in the function *Write* while multiple replicas of multiple blocks are written to different nodes parallel in the function *ParallelWrite*.

## 10. Persistence of TFS Metadata

TFS metadata are kept under the directory *TFSData* on TFSNameNode. File *TFSImage* stores entire file system namespace. It is the snapshot of TFSNameNode when it starts. File *TFSEditLog* is a transaction log to record every change that occurs to file system metadata.

When TFSNameNode starts, it handles *TFSImage* and *TFSEditLog* as follows:

a. load namespace *TFSImage*
b. apply logs in *TFSEditLog*
c. save new namespace to *TFSImage.new*
d. delete *TFSImage* and *TFSEditLog*
e. rename *TFSImage.new* to *TFSImage*

## 11. TFSNameNode

As the master of TFS cluster, TFSNameNode need to maintain the global information, manage nodes in the cluster and regulate the client access to data.

## 11.1 File System Namespace

Each file stored on TFS is divided into fixed-size blocks. Therefore, the file is only recorded as a sequence of block IDs while the block content is stored on the TFSDataNodes. TFS Namespace mainly maintains the file set and provides the service for data position request. It is constituted by the below four classes.

```
class TFSBlockInfo
{
    int blockSize;
    /// <summary>
    /// stores the tfsdatanode numbers which own this block.
    /// </summary>
    List<int> nodeList;
}
class TFSBlockDictionary
{
    static Dictionary<string, TFSBlockInfo> blocks;
}
class TFSFileInfo
{
    /// <summary>
    /// stores the blockIDs which make up this file.
    /// </summary>
    List<string> blockIDList;
    bool isUnderConstruction;
}
class TFSFileInfoCollection
{
    static Dictionary<string, TFSFileInfo> files;
}
```

TFS File System Namespace includes three logic parts, which consists of the above four classes.

a. File Set. *TFSFileInfoCollection* contains all file information. Compared with the general directory tree, we just have implemented this flat structure so far.
b. File – Block mapping. *TFSFileInfo* stores the IDs (implemented by a GUID string) of blocks which make up this file.
c. Block – TFSDataNode mapping. To get the concrete data, we must acquire blocks' positions. *TFSBlockDictionary* and *TFSBlockInfo* provide this query by the known block ID.

## 11.2 Lease Management

TFS adopts the lease method to regulate the clients' concurrent access to data. No matter read or write, users should apply a lease before opening a file, update the lease during the operation and delete the lease after closing the file. Through the lease, TFS could manage the exclusive read and write. Besides that, if a user is lost during read or write, then his lease would not be updated anymore and eventually overdue. TFS would find those overdue write lease and roll back those write operations.

```
class LeaseManager
{
    private static Dictionary<string, ReadLeaseDictionary> read_lease_table;
    private static Dictionary<string, WriteLease> write_lease_table;
}
class ReadLeaseDictionary
{
    /// <summary>
    /// Dictionary<handleID, ticks>
    /// </summary>
    private Dictionary<long, long> readLease;
}
internal class WriteLease
{
    long handleID; //session ID
    long time; //ticks
}
```

Since one file only allows be either read or written at the same time, the read lease and the write lease on the same file are mutually exclusive. TFS allows multiple readers but only one writer on one file at any time; therefore, multiple read leases, which are distinguished by different handleIDs (session ID), could be added on the same file.

## 11.3 Task Dispatcher

To balance the cluster overhead, TFS uses a class called *TaskDispatcher* to trace the jobs on different TFSDataNodes. With this information, TFSNameNode always assigns the task to the TFSDataNode which has not been assigned a job for the longest time. Therefore, the timestamp is like the penalty.

```
class TaskDispatcher
{
    /// <summary>
    /// Dictionary<tfsdatanodeID, Ticks>
    /// </summary>
    private static Dictionary<int, long> tasks;
}
```

In the writing data, clients need multiple TFSDataNodes to write the same block. TaskDispatcher would choose the top N idlest nodes and update their timestamps. But in the reading data, *TaskDispatcher* would choose the idlest one from those nodes which contains the specific block and update its timestamp.

## 11.4 Block Report Management

Each TFSDataNode would send its block report, including all the block file names and block sizes to TFSNameNode periodically. TFSNameNode has a class named *BlockReportManager* to keep these reports. If the block in the report exists in *TFSBlockDictionary*, and its size in report is no less than the size in *TFSBlockInfo*, *BlockReportManager* would add this block or update its timestamp. We must clearly point out that why the block size in report is must no less than the size in *TFSBlockInfo*. When we write a new block, we first create a 0-byte block and add it into *TFSBlockDictionary*. If the client has written all the replicas of this block, it would send a message to TFSNameNode to update this block size. Similarly, if the client appends some data to

a file, it would append the data to all the replicas of the tail block and then update the block size. In above situations, the block size in the block report may be greater than it in the *TFSBlockInfo*. Once the block size in the block report may is less than it in the *TFSBlockInfo*, we can confirm this block is an old version and will delete it later.

```
class BlockReportManager
{
    /// <summary>
    /// Dictionary<tfsdatanodeID, Dictionary<blockID, Ticks>>
    /// </summary>
    private static Dictionary<int, Dictionary<string, long>> reports;
}
```

*BlockReportManager* has a background thread to check those overdue blocks which have not been updated for a long time. Those overdue blocks would be deleted from *BlockReportManager* and *TFSBlockDictionary* both.

*TFSBlockDictionary* provides an index from block ID to its TFSDataNode ID while *BlockReportManager* offers the reverse one from TFSDataNode ID to those blocks on this node. This reverse index would be used for handling those lost TFSDataNodes.

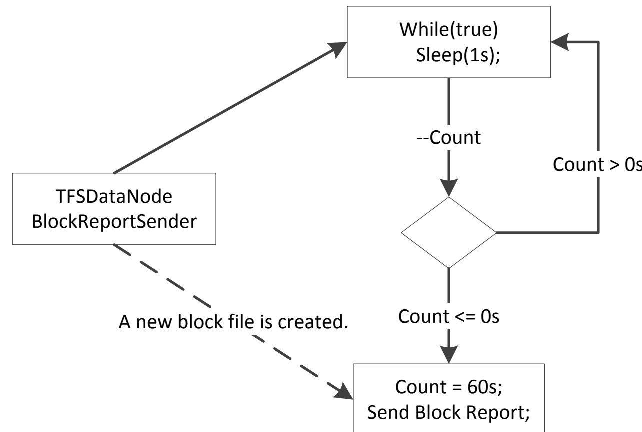
## 11.5 Heart Beat

The heart beat mechanism is a part of Trinity. It provides TFSNameNode an interface to handle the loss of TFSDataNode. TFSNameNode registers two handlers to prevent this harm. If the lost TFSDataNode rejoins the cluster, TFSNameNode would receive a block report whose TFSDataNode ID does not exist in *BlockReportManager*. Then TFSNameNode would clear this TFSDataNode and add it to *BlockReportManager* and *TaskDispatcher* as a “clean” node.

## 11.6 Block Replication Management

Many reasons may lead the number of one block’s replicas cannot match the configured REPLICA_NUM; therefore, the class *ReplicateBlockManager* starts a background thread to check this number. If the number is greater than REPLICA_NUM, *ReplicateBlockManager* would delete the redundant replicas; otherwise, it would copy one replica to more TFSDataNodes to reach REPLICA_NUM. Since this check may need to copy or delete data, it adds the read lease on the file. If one client wants to delete, append or rewrite this file at the same time, client’s operation may fail on adding the write lease even though it cannot see

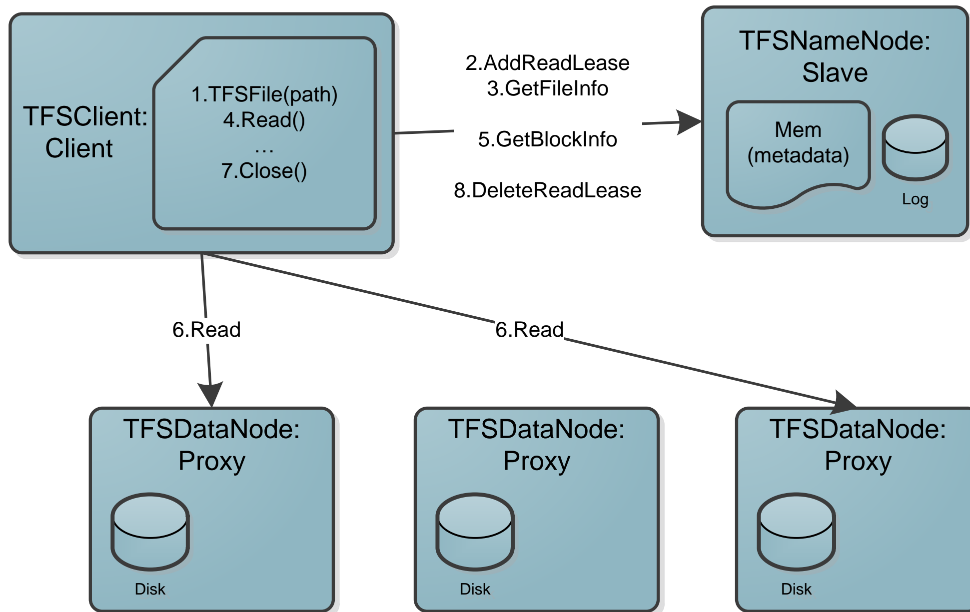
Apart from that, TFSDataNode would send the block report to TFSNameNode. *Trinity.IO.DataNode.BlockReportSender* has a countdown timer to control the block report sending. The timer is always set to *TrinityConfig.BLOCK_REPORT_PERIOD* again after sending the block report. But if a new block file is created on the TFSDataNode, *BlockReportSender* would send the block report immediately and set the timer to *TrinityConfig.BLOCK_REPORT_PERIOD*.



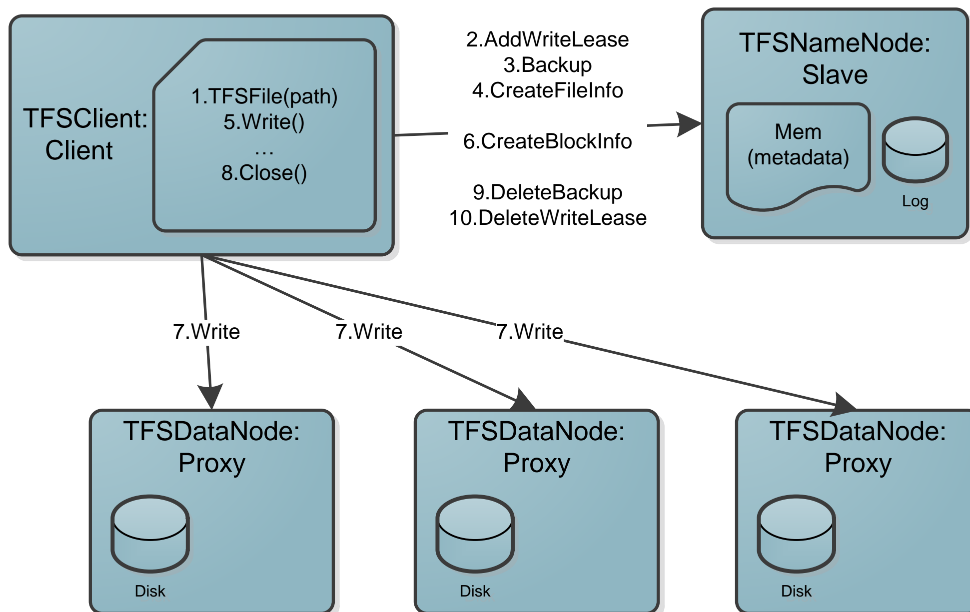

The immediate block report confirms *BlockReportManager* can own the real-time reverse index. Supposing one TFSDataNode is lost just after writing one block, *BlockReportManager* does not keep it but *TFSBlockDictionary* owns. We delete those blocks according to the lost TFSDataNode ID and the reverse index, so this block would not be found. The immediate block report is sent when the new empty block file is created; therefore, *BlockReportManager* could get the real-time information.

## 13. Data Flow

## 13.1 Anatomy of a File Read




## 13.2 Anatomy of a File Write




# 14. Data Renovation

The block files or checksum files on TFSDataNodes may be corrupted due to many reasons, so TFS builds a mechanism to renovate those corrupted blocks.

If a client finds that the data cannot match the corresponding checksum in the reading data, it would try to read data from other TFSDataNodes and submit a data validation request. An object on TFSNameNode called *DataValidator* would add this request to its list and execute

these requests in the background thread. *DataValidator* sends the data validation to the TFSDataNode which owns the “suspected” block. If TFSDataNode confirms the data cannot match the checksum, TFSNameNode would delete this replica. *ReplicateBlockManager* would copy another replica later.

```
struct SuspectedBlock
{
    internal string blockID;
    internal int tfsdatanode;
    internal int blockSize;
}
class BlockValidator
{
    private static List<SuspectedBlock> list;
}
```

Every 1MB data unit in the block file owns a 16-byte (128-bit) MD5 checksum in the corresponding checksum file, but the tail data unit which is less than the fixed size also has a checksum. When clients append data to files, clients should verify the tail unit’s checksum and rewrite its checksum after appending. If clients find the tail unit cannot match its checksum, it would ignore it and write other replicas first. If all the replicas cannot match their checksums, clients would throw exceptions; otherwise, clients would send requests to TFSNameNode to delete wrong replicas and copy right replicas which are appended successfully.

# 15. Transaction

TFS supports the atomic write operation. Write failures could be caused by many reasons, and we classify them into two types:

a. Clients’ failure, like the client is disconnected during the writing; the client crashes and so on.
b. TFSNameNode’ failure, like TFSNameNode loses the power, the TFSNameNode crashes and so on.

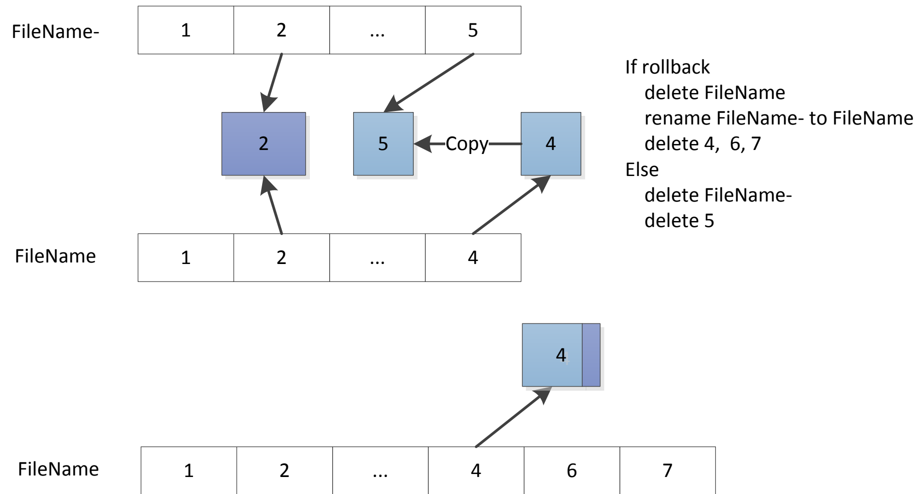
We also work out the below roll back strategy:

a. Failure in writing a new file: Delete the unfinished file.
b. Failure in overwriting a file: Recover to the original file.
c. Failure in appending an existing file: Recover to the original file.

## 15.1 Backup

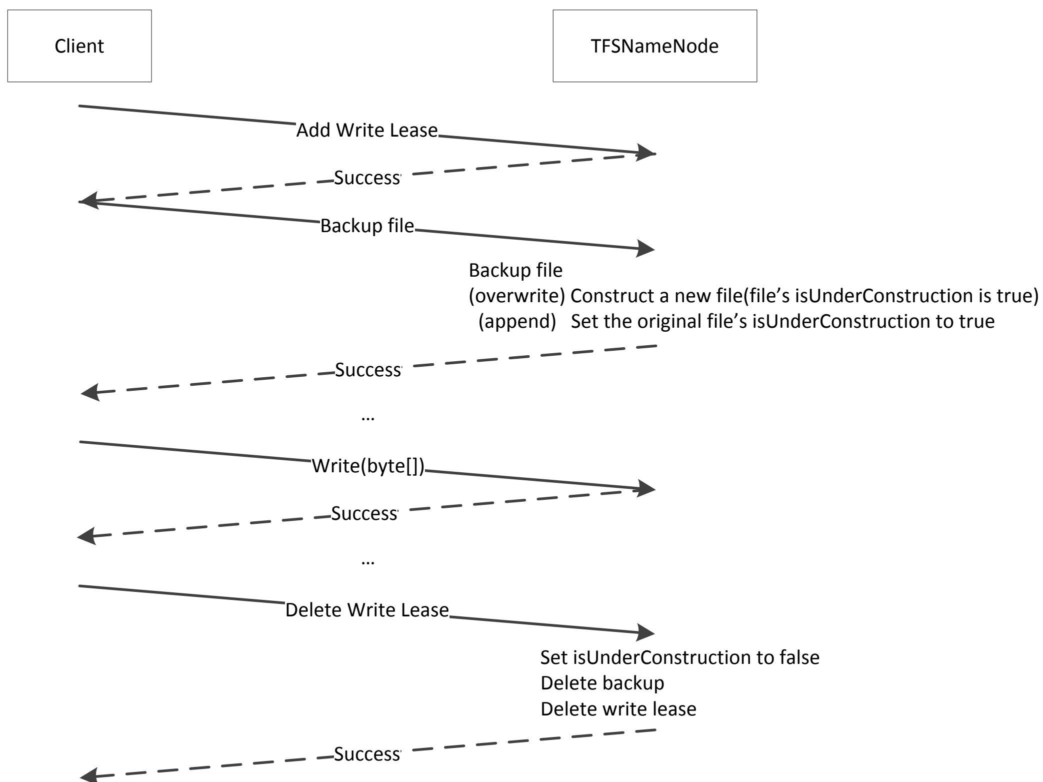
Before the overwriting and appending, TFS needs to store a backup of the original file. The backup is quite easy for overwrite. TFSNameNode just renames the original file FileName to the backup file FileName~. Then overwriting becomes writing a new file. But for append, it is more complex. As shown of the below figure, TFSNameNode firstly requests the TFSDataNodes, which own the tail block 4, to duplicate the tail block. Then TFSNameNode creates a backup file called

FileName-, which shares the blocks with the original file FileName except the tail one. The new data would be appended to the file FileName and we can copy the fewest data to implement the backup.




## 15.2 Normal Procedure

The below figure illustrates a normal procedure for write, overwrite or append. There is a property called *isUnderConstruction* in class *TFSFileInfo* to record whether the file has been written completely. The normal procedure can confirm at least one of backup and original files is available.



## 15.3 Rollback

The rollback happens in two situations.

For the clients' failure, *LeaseManager* would find the file's overdue write leases.

a. If the file's *isUnderConstruction* is true, TFSNameNode would recover it if its backup file exists or just delete it if its backup file does not exist.
b. If the file's *isUnderConstruction* is false, TFSNameNode would try to delete its backup file if its backup file exists. In this case the probability of occurrence is low.

For the TFSNameNode's failure, TFSNameNode would check each file's *isUnderConstruction* property when it restarts.

a. If the file's *isUnderConstruction* is true, TFSNameNode would recover it if its backup file exists or just delete it if its backup file does not exist.
b. If the file's *isUnderConstruction* is false, TFSNameNode would try to delete its backup file if its backup file exists. In this case the probability of occurrence is very low.
c. If a backup file exists but the original file does not exist, TFSNameNode would try to recover the file. In this case the probability of occurrence is very low.

# 16. Failure

Trinity.Core provides two failure handlers: connection failure handler and machine failure handler. The connection failure handler means the real-time handling while the machine failure handler would be invoked after N fail connections.

If one TFSDataNode cannot be connected once, a connection failure handler would be invoked at the client. The client's *ConnectionFailureManager* would record this TFSDataNode and time. When the client reads the data, it would verify whether the data source is in the *ConnectionFailureManager*. If the data source is added a moment ago (*TrinityConfig.OVERDUE_CONNECTION_LIMIT*), this read's data may come from Trinity's buffer rather than TFSDataNode, so the client would read from other TFSDataNode again. If the TFSDataNode is lost eventually, *TaskDispatcher* would remove it and never assign the task to it anymore; at the same time, TFSNameNode would remove its block report from *BlockReportManager* and find all the blocks on this node according to its block report to delete those blocks from *TFSBlockDictionary* later.

To catch the TFSNameNode's failure, TFSDataNode and the client both register the machine failure handlers. *Trinity.IO.DataNode.TFSNameNodeFailureManager* let the TFSDataNode exit automatically while *Trinity.IO.Client.MachineFailureManager* makes the client throw the *Trinity.IO.TFSNameNodeException*.