Modeling of distributed file System in big data storage by event-B

Ammar Alhaj Ali[1], *Pavel Varacha*[1, *], *Said Krayem*[1], *Roman Jasek*[1], *Petr Zacek*[1], and *Bronislav Chramcov*[1]

[1]Faculty of Applied Informatics, Tomas Bata University in Zlin, Czech Republic

Abstract. Nowadays, a wide set of systems and application, especially in high performance computing, depends on distributed environments to process and analyses huge amounts of data. As we know, the amount of data increases enormously, and the goal to provide and develop efficient, scalable and reliable storage solutions has become one of the major issue for scientific computing. The storage solution used by big data systems is Distributed File Systems (DFSs), where DFS is used to build a hierarchical and unified view of multiple file servers and shares on the network. In this paper we will offer Hadoop Distributed File System (HDFS) as DFS in big data systems and we will present an Event-B as formal method that can be used in modeling, where Event-B is a mature formal method which has been widely used in a number of industry projects in a number of domains, such as automotive, transportation, space, business information, medical device and so on, And will propose using the Rodin as modeling tool for Event-B, which integrates modeling and proving as well as the Rodin platform is open source, so it supports a large number of plug-in tools.

1 Introduction

The ability to process and manage large volumes of data such as, search engines, databank centers and data mining systems…etc. require an infrastructure for storing and retrieving data, where the distributed file systems are essential component for storing data infrastructure [1]. DFS provides permanent storage for sharing multiple files and build a hierarchical and unified view of these files by federating storage resources dispersed in a network. So high performance computing applications heavily rely on these DFSs [2]. A DFS is a file system that supports the sharing of files in the form of persistent storage over a set of the network connected nodes. Many DFS's have been developed over the years and almost two decades of research have not succeeded in producing a fully featured DFS [1]. This paper is divided into two sections; In First Section, we will define DFS and show its properties and mechanism to store and share data; then will show used approaches for big data systems, and as case study we will offer Hadoop and HDFS Architecture, and in second section will offer Event-B and modeling in Event-B by Rodin tool, where will demonstrate benefits and features for using Event-B in analysing and modeling the systems.

2 Distributed File Systems

A distributed file system is a client/server-based application that allows clients to access and process data stored on the server as if it were on their own computer [3], The Distributed File System (DFS) is used to build a hierarchical view of multiple file servers and shares on the network. Instead of having to think of a specific machine name for each set of files, the user will only have to remember one name; which will be the 'key' to a list of shares found on multiple servers on the network [1]. Unlike local files systems, storage resources and clients are dispersed in a network. Files are shared between users in a hierarchical and unified view: files are stored on different storage resources but appear to users as they are put on a single location [2]. Basically, A Distributed File System (DFS) is simply a classical model of a file system distributed across multiple machines. The purpose is to promote sharing of dispersed files the resources on a particular machine [4].

3 Big data Solutions

In general, there are two Approaches for big data solution: Traditional Approach and MapReduce Approach.

3.1 Traditional Approach

In this approach, will have a computer (Server) to store, process and retrieve big data, where data will be stored in an RDBMS like Oracle Database, MS SQL Server or DB2 and sophisticated software can be written to interact with the database, process the required data and present

[*] Corresponding author: varacha@utb.cz

it to the users [5], where the traditional database works on data size in range of Gigabytes. [6]

3.2 MapReduce Approach

Google solved the problem of traditional approach with huge amount of data using MapReduce [5], MapReduce is a programming model and an associated implementation for processing and generating big data sets with a parallel, distributed algorithm on a cluster [7], This algorithm divides the task into small parts and assign those parts to many computers connected over the networks and collects the results to form the final result dataset. [5]

4 Hadoop

Hadoop was created by Doug Cutting, the creator of Apache Lucene, the widely used text search library. Hadoop has its origins in Apache Nutch, an open source web search engine; itself a part of the Lucene project [8]. Where Doug Cutting and his team took the solution provided by Google (MapReduce) and started their project (HADOOP) [5]. The Apache Hadoop software library is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models. It is designed to scale up from single servers to thousands of machines, each offering local computation and storage. Rather than rely on hardware to deliver high-availability, the library itself is designed to detect and handle failures at the application layer, so delivering a highly-available service on top of a cluster of computers, each of which may be prone to failures [9].

5 Hadoop Distributed File System (HDFS)

The Hadoop Distributed File System (HDFS) is designed to store very large data sets reliably, and to stream those data sets at high bandwidth to user applications. In a large cluster, thousands of servers both host directly attached storage and execute user application tasks. By distributing storage and computation across many servers, the resource can grow with demand while remaining economical at every size. We describe the architecture of HDFS and report on experience using HDFS to manage 25 petabytes of enterprise data at Yahoo! [10].

6 HDFS Architecture

HDFS is a file system designed for storing very large files and it doesn't require expensive and highly reliable hardware. It's designed to run on clusters of commodity hardware for which the chance of node failure across the cluster is high, at least for large clusters. HDFS is designed to carry on working without a noticeable interruption to the user in the face of such failure [11], and it has been designed to be easily portable from one platform to another. This facilitates widespread adoption of HDFS as a platform of choice for a large set of applications [12]. An HDFS cluster has two types of node operating in a master-worker pattern: a namenode (the master) and a number of datanodes (Slaves) [8]. HDFS Architecture could is represented by Figure 1.

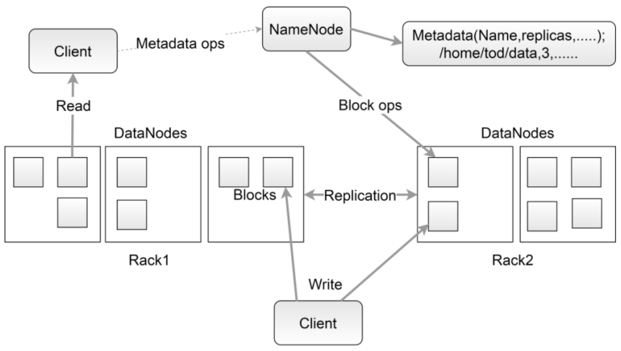

Fig. 1. HDFS Architecture (own source).

Apache Hadoop HDFS Architecture is as following [13]:

- HDFS Master/Slave Topology: Hadoop Distributed File System (HDFS) is a block-structured file system where each file is split into one or more blocks and these blocks of a pre-determined size (typically 128 megabytes) are stored across a cluster of one or more DataNodes. HDFS Architecture follows a Master/Slave Architecture, where a cluster consists of a single NameNode (Master node) and all the other nodes are DataNodes (Slave nodes) [12][13].

- NameNode Name: NameNode can be considered as a master of the system. It maintains the file system tree and the metadata for all the files and directories present in the system. NameNode is a very highly available server that manages the File System Namespace and controls access to files by clients. Namenode has knowledge of all the datanodes containing data blocks for a given file [13].

- DataNode: The DataNodes are responsible for serving read and write requests from the file system's clients. The DataNodes also perform block creation, deletion, and replication upon instruction from the NameNode [12], for every node in a cluster, there will be a datanode. These nodes manage the data storage of their system [5].

- Blocks in HDFS: When you store a file in HDFS, the system breaks it down into a set of individual blocks and stores these blocks in various DataNodes in the Hadoop cluster. This is an entirely normal thing to do, as all file systems break files down into blocks before storing them to disk [14].

If we had a block size of let's say of 4 KB, as in Linux file system, we would be having too many blocks and managing this number of blocks will create huge overhead. That's why we need to have such a huge blocks size i.e. 128 MB. The blocks are of fixed size, so it is very easy to calculate the number of blocks that can be stored on a disk[13]. Blocks in HDFS are represented by Figure 2.

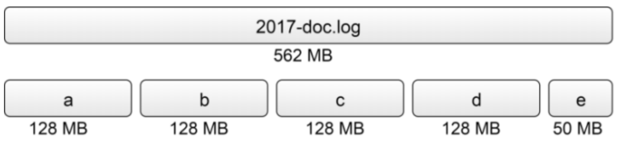

Fig. 2. Blocks in HDFS (own source).

- Replication Management: The blocks of a file are replicated for fault tolerance. The replication factor can be specified at file creation time and can be changed later [12]. As you can see in Figure 3, where each block is replicated three times and stored on different DataNodes. Here, was used default replication factor 3 [13].

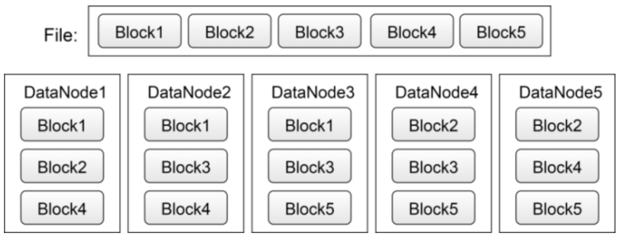

Fig. 3. Replication management of blocks (own source).

All the replicas are not stored on the same rack or on a single rack. It follows an in-built Rack Awareness Algorithm to reduce latency as well as provide fault tolerance [13]. Rack Awareness algorithm is shown in Figure 4.

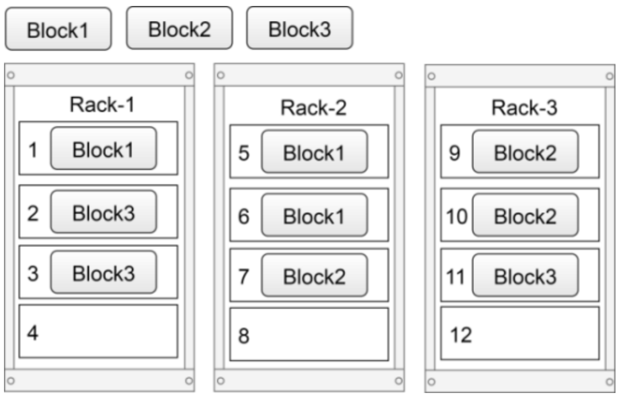

Fig. 4. 6 Rack Awareness algorithm (own source).

- HDFS Read: HDFS follows next steps to achieve Read operation [13] [15, 16]. The HDFS Read operation is illustrated by Figure 5.

1. Client initiates read request by calling Open() method of FileSystem object, which for HDFS is an instance of DistributedFileSystem.

2. This object connects to namenode using RPC (Remote Procedure Call) and gets metadata information such as the locations of the blocks of the file, where these addresses are of first few block of file.

3. For each block the namenode returns the addresses of the datanodes that have a copy of that block and datanodes are sorted according to their proximity to the client.

4. Once addresses of DataNodes are received, an object of type FSDataInputStream is returned to the client. FSDataInputStream contains DFSInputStream which takes care of interactions with DataNode and NameNode, where client invokes Read() method which causes DFSInputStream to establish a connection with the first DataNode with the first block of file.

5. Data is read in the form of streams wherein client invokes Read() method repeatedly. This process of Read() operation continues till it reaches end of block.

6. Once end of block is reached, DFSInputStream closes the connection and moves on to locate the next DataNode for the next block

7. Once client has done with the reading, it calls Close() method.

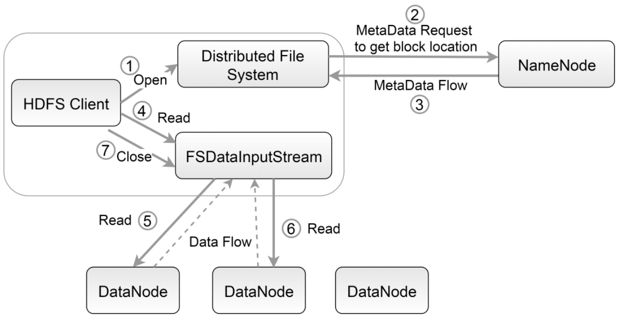

Fig. 5. HDFS Read Operation (own source).

- HDFS Write: HDFS follows next steps to achieve write operation [13] [15, 16], see Figure 6.

1. Client initiates write operation by calling Create() method of DistributedFileSystem object which creates a new file.

2. DistributedFileSystem object do a RPC (Remote Procedure Call) to namenode to create a new file in FileSystem namespace with no blocks associated to it, the client should has permissions to create a file or not.

3. Once new record in NameNode is created, an object of type FSDataOutputStream is returned to the client. Client uses it to write data into the HDFS.

4. FSDataOutputStream contains DFSOutputStream object which looks after communication with DataNodes and NameNode. While client continues writing data, DFSOutputStream continues creating packets with this data. These packets are en-queued into a queue which is called as DataQueue.

5. DataQueue is then consumed by a DataStreamer which also asks NameNode for allocation of new blocks thereby picking desirable DataNodes to be used for replication.

6. Now, the process of replication starts by creating a pipeline using DataNodes. In our case, we have chosen replication level of three and there will be three nodes in the pipeline.

7. The DataStreamer pours packets into the first DataNode in the pipeline.

8. Every DataNode in a pipeline stores packet received by it and forwards the same to the second DataNode in pipeline.

9. Another queue, acknowledgement Queue is maintained by DFSOutputStream to store packets which are waiting for acknowledgement from DataNodes.

10. Once acknowledgement for a packet in queue is received from all DataNodes in the pipeline, it is removed from the acknowledgement Queue. In the event of any DataNode failure, packets from this queue are used to reinitiate the operation.

11. After client is done with the writing data, it calls Close() method, results into flushing remaining data packets to the pipeline followed by waiting for acknowledgement.

12. Once final acknowledgement is received, NameNode is contacted to tell it that the file write operation is complete.

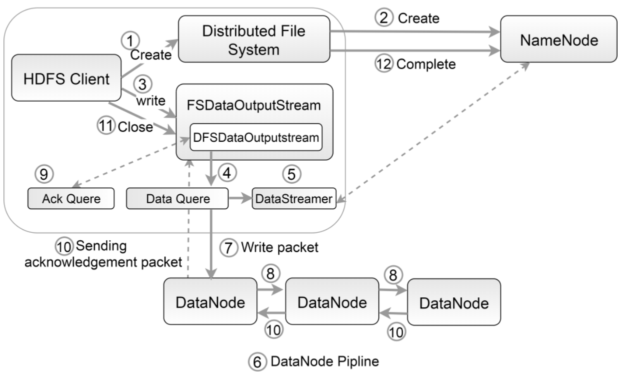

Fig. 6. HDFS Write Operation (own source).

7 Use formal method Event-B to model

Formal methods-based development is a standard and popular approach to deal with the increasing complexity of a system with assurance of correctness in the modern software engineering practices. Formal methods-based techniques increasingly control safety-critical functionality in the development of the highly critical systems. These techniques are also considered as a way to meet the requirements of the standard certificates to evaluate a critical system before use in practice [18]. The primary concept in doing formal developments in Event-B is that of a model [19]. An Event-B model is constructed from a collection of modeling elements. The modeling elements have attributes that can be based on Set Theory and Predicate Logic. Set Theory is used to represent data-types and to manipulate the data. Logic is used to apply conditions to the data. The development of an Event-B model goes through two stages; abstraction and refinement.

- The abstract machine specifies the initial requirements of the system.

- Refinement is carried out in several steps - with each step adding more detail to the system [19].

A model contains the complete mathematical development of a Discrete Transition System. It is made of several components of two kinds: machines and contexts[20][21].

8 Formal modeling of HDFS in Event-B

In this section we will offer modeling for writing operation in HDFS.

8.1 Abstract Model

The cluster in HDFS consists of a single NameNode (Master node) and all the other nodes are DataNodes (Slave nodes),

So we will use following variables: DataNode ⊆ CLUSTER, NameNode ∈ CLUSTER. The Hadoop clients Will send request to create a file and NameNode will perform some checks before create file (has the client permissions?), if all checks pass the NameNode create file and return success to the client see event ClientCreateFile, in Figure 8. After create the file (empty file), it is time to start writing data, HDFS clients will create FSDataOutputStream and start writing data to it, FSDataOutputStream have many tasks, it queues the data locally and divides file into blocks (128 MB), once there is one block of data the DataStreamer connect to NameNode to ask location of block in DataNode, The NameNode can easily assign DataNode to store that block and send back the DataNode name to DataStreamer, see event AddBlockToNameNode in Figure 8. Now, the DataStreamer will start send block to DataNode, if the file is larger than one block the DataStreamer will again connect to NameNode to get location of new block, see event AddBlockToDataNode in Figure 7. Figure 7 is the representation of the source code.

```
variables DataNode NameNode Files
SuccessCreateFile FSDataOutputStream BlockDN
BlockNN
invariants
  @inv1 DataNode ⊆ CLUSTER
  @inv2 NameNode ∈ CLUSTER
  @inv3 {NameNode} ∪ DataNode = CLUSTER
  @inv4 FSDataOutputStream ⊆  BLOCKS
  @inv5 BlockNN⊆FILES×BLOCKS×DataNode
  @inv6 BlockDN⊆ BLOCKS
  @inv7 FSDataOutputStream ∩ BlockDN=∅
  @inv8 Files⊆FILES

  @inv9 SuccessCreateFile ∈ CreateFile
event ClientCreateFile
   any f
   where
    @grd1 f∈FILES
   then
    @act1 Files:=Files ∪ {f}
    @act2 SuccessCreateFile:=True
  End
```

```
event AddBlockToNameNode
   any f bk DN
   where
    @grd1 f∈FILES
    @grd2 bk∈FSDataOutputStream
    @grd3 bk ∉ BlockDN
    @grd4 DN∈DataNode
    @grd5 {f↦bk↦DN} ⊈ BlockNN
   then
    @act1 BlockNN ≔ BlockNN ∪
{f↦bk↦DN}
  endevent AddBlockToDataNode

 any f bk DN
 where
  @grd1 f∈FILES
  @grd2 bk ∈ FSDataOutputStream
  @grd3 bk ∉ BlockDN
  @grd4 DN∈DataNode
  @grd5 {f↦bk↦DN} ⊆ BlockNN
 then
  @act1 FSDataOutputStream ≔
FSDataOutputStream \ {bk}
  @act2 BlockDN ≔ BlockDN ∪ {bk}
 end
```

Fig. 7. A specification of abstract model (own source).

8.2 First refinement

In the first refinement step we extend the abstract model by add new variables and events to improve performance NameNode to response all requests to get location of data blocks, where the block in NameNode will pass in three stages: Buffer, Inprocessing and Processed, so add extra variable and events. In Figure 8 you can see the result of the First refinement as the representation of source code.

```
 event AddBlockToNameNodeBuffer
  any f bk DN
  where
   @grd1 f∈FILES
   @grd2 bk∈FSDataOutputStream
   @grd3 bk ∉ BlockDN
   @grd4 DN∈DataNode
   @grd5 {f↦bk↦DN} ⊈ BlockNN
   @grd6 bk∈BLOCKS
   @grd7 bk∉BlkBuffer
   @grd8 bk∉BlkInprocessing
   @grd9 bk∉BlkrProcessed
  then
   @act2 BlkBuffer:=BlkBuffer ∪ {bk}
 End
event AddBlockToNameNodeProcessed
  any bk
  where
   @grd1 bk∈BlkInprocessing
   @grd2 bk∉BlkrProcessed
  then
   @act1 BlkInprocessing:=BlkInprocessing \
{bk}
   @act2 BlkrProcessed:=BlkrProcessed ∪ {bk}
 end
```

```
event AddBlockToNameNodeInprocessing refines
AddBlockToNameNode
  any f bk DN
  where
   @grd1 f∈FILES
   @grd2 bk∈BLOCKS
   @grd3 bk∈FSDataOutputStream
   @grd4 bk ∉ BlockDN
   @grd5 DN∈DataNode
   @grd6 {f↦bk↦DN} ⊈ BlockNN
   @grd7 bk ∈ BlkBuffer
   @grd8 bk∉BlkInprocessing
  then
   @act3 BlkBuffer:=BlkBuffer \ {bk}
   @act2 BlkInprocessing:=BlkInprocessing ∪
{bk}
   @act1 BlockNN ≔ BlockNN ∪ {f↦bk↦DN}
 end
```

Fig. 8. A specification of First refinement (own source).

8.3 Second refinement

In HDFS, NameNode will receive many requests from DataStreamer from many HDFS clients to assign location of blocks in DataNode, so will add new variables

```
event AddBlockToNameNodeBuffer refines
AddBlockToNameNodeBuffer
  any f bk DN
  where
   @grd1 f∈FILES
   @grd2 bk∈FSDataOutputStream
   @grd3 bk ∉ BlockDN
   @grd4 DN∈DataNode
   @grd5 {f↦bk↦DN} ⊈ BlockNN
   @grd6 bk∈BLOCKS
   @grd7 bk∉SD_BlkBuffer(NameNodeBuffer)
   @grd8
bk∉SD_BlkInprocessing(NameNodeBuffer)
   @grd9
bk∉SD_BlkrProcessed(NameNodeBuffer)
  then
   @act2
SD_BlkBuffer(NameNodeBuffer):=SD_BlkBuffer(
NameNodeBuffer) ∪ {bk}
 End
event AddBlockToNameNodeProcessed refines
AddBlockToNameNodeProcessed
  any bk
  where
   @grd1
bk∈SD_BlkInprocessing(NameNodeBuffer)
   @grd2
bk∉SD_BlkrProcessed(NameNodeBuffer)
  then
   @act1
SD_BlkrProcessed(NameNodeBuffer):=SD_BlkrPr
ocessed(NameNodeBuffer) ∪ {bk}
   @act2
SD_BlkInprocessing(NameNodeBuffer):=SD_BlkI
nprocessing(NameNodeBuffer) \ {bk}
 end
```

```
event AddBlockToNameNodeInprocessing refines
AddBlockToNameNodeInprocessing
  any f bk DN
  where
   @grd1 f∈FILES
   @grd2 bk∈BLOCKS
   @grd3 bk∈FSDataOutputStream
   @grd4 bk ∉ BlockDN
   @grd5 DN∈DataNode
   @grd6 {f↦bk↦DN} ⊈ BlockNN
   @grd7 bk ∈ SD_BlkBuffer(NameNodeBuffer)
   @grd8
bk∉SD_BlkInprocessing(NameNodeBuffer)
  then
   @act2
SD_BlkInprocessing(NameNodeBuffer):=SD_BlkI
nprocessing(NameNodeBuffer) ∪ {bk}
   @act3
SD_BlkBuffer(NameNodeBuffer):=SD_BlkBuffer(
NameNodeBuffer) \ {bk}
   @act1 BlockNN ≔ BlockNN ∪ {f↦bk↦DN}
 End
event AddBlockToDataNode refines
AddBlockToDataNode
  any f bk DN
  where
   @grd1 f∈FILES
   @grd2 bk ∈ FSDataOutputStream
   @grd3 bk ∉ BlockDN
   @grd4 DN∈DataNode
   @grd5 {f↦bk↦DN} ⊆ BlockNN
  then
   @act1 FSDataOutputStream ≔
FSDataOutputStream \ {bk}
   @act2 BlockDN ≔ BlockDN ∪ {bk}  end
```

Fig. 9. A specification of Second refinement (own source).

8.4 Results and proof statistics

In table 1, we can see proof statistics for our model using the Rodin3.2 platform, the statistics give us the proof obligations generated and discharged by the Rodin, The finial development of our model results in 59 POs (Proof obligations), around (95%) of them have been proved automatically by the Rodin platform and the rest have been proved manually in the Rodin interactive proving environment [20] [21].

Table 1. HDFS Architecture HDFS (Our model)

HDFS (Our model)	Total	Auto	Manual
	59(100%)	56 (95%)	3 (0.05%)
Ctx_HDFS	0	0	0
M_HDFS0	6	6	0
M_HDFS1	11	11	0
M_HDFS2	42	39	3

9 Conclusion

In this paper, we have presented some of the basic concepts in formal method using Event-B to model a distributed file system in big data systems, Where Formal method is one of the mechanisms which help us to understanding the complex specification of systems and how to analyse and build its. And we proposed Hadoop distributed file system (HDFS) as study case to proof that Event-B is a formal method allowing a stepwise development of reactive distributed systems, and we proposed using Event-B to helpful the specification and the safe development of distributed systems. So we can say, Event-B is a formal method that is used for specifying and reasoning about complex systems and Rodin tool is a platform where verification of the program is done and offers reactive environment for constructing and analyzing models as do most modern integrated development environments, and provides integration between modeling and proving whereas this is important feature for the developers to focus on the modeling task without switch between different tools to check proving in same time. Our final models are translated into the Event-B notation to verify required properties, so we can say; event-B allows us to define a kind of modeling methodology by write the correct mathematical notions.

In the future, we will conduct more research on the modeling in Event-B and Rodin platform as well as offer techniques and describe code generation approaches and tools (in Event-B) to generate code(Java or C).

This work was supported by the Ministry of Education, Youth and Sports of the Czech Republic within the National Sustainability Programme project No. LO1303 (MSMT-7778/2014) and by the European Regional Development Fund under the project CEBIA-Tech No. CZ.1.05/2.1.00/03.0089. Also supported by grant No. IGA/CebiaTech/2017/007 from IGA (Internal Grant Agency) of Tomas Bata University in Zlin.

References

1. T. D. Thanh, S. Mohan, E. Choi, S. B. Kim, P. Kim, A Taxonomy and Survey on Distributed File Systems (2008)
2. B. D., G. LeMahec, C. Seguin, Analysis of Six Distributed File Systems (2013)
3. http://searchwindowsserver.techtarget.com/definition/distributed-file-system-DFS
4. A. Das, D. Tiwari, N. Rajani, R. Moona, Distributed File Systems:A Case Study (2010)
5. V. K. Jain, Big Data and Hadoop (2017)
6. A. K. Swapnil, S. S. Dangee, A Comparative Analysis of Traditional RDBMS with MapReduce and Hive for E-Governance system (2015)
7. https://en.wikipedia.org/wiki/MapReduce
8. T. White, Hadoop: The Definitive Guide, O'REILLY (2012)
9. http://hadoop.apache.org/
10. K. Shvachko, H. Kuang, S. Radia, R. Chansler, The Hadoop Distributed File System (2010)
11. http://www.corejavaguru.com/bigdata/hadoop/hdfs-architecture
12. https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html
13. http://www.waytoeasylearn.com/2016/11/hdfs-tutorial.html
14. D. deRoos, Hadoop for Dummies (2014)
15. https://www.guru99.com/learn-hdfs-a-beginners-guide.html
16. http://www.hadoopinsight.com/blog/hdfs/anatomy_of_file_write_and_read_in_hadoop
17. http://checkmkblog.blogspot.com/2013/07/hadoop-vs-rdbms.html
18. N. K. Singh, Using Event-B for Critical Device Software Systems, Springer (2013)
19. J.-R. Abrial, Modeling in Event-B System and Software Engineering (2011)
20. Ammar Alhaj Ali, Roman Jasek, Said Krayem, Petr Zacek; Proving the Effectiveness of Negotiation Protocols KQML in Multi-agent Systems Using Event-B(2017);ISBN:978-3-319-57264-2; https://link.springer.com/chapter/10.1007/978-3-319-57264-2_40.
21. Ammar Alhaj Ali, Roman Jasek, Said Krayem, Bronislav Chramcov, Petr Zacek; Improved Adaptive Fault Tolerance Model for Increasing Reliability in Cloud Computing Using Event-B (2018);ISBN:978-3-319-91192-2; https://link.springer.com/chapter/10.1007/978-3-319-91192-2_25.