BlobSeer Scalability: A Multi-version Managers Approach

Baya Chalabi
National Scholl of Computer Science, Oued-Smar -ESI- (Alger's)
Algeria
b_chalabi@esi.dz

Yahya Slimani
ISAMM, Manouba University
Tunisia
yahya.slimani@fst.rnu.tn

ABSTRACT: *With the emergence of Cloud Computing, the amount of data generated in different fields such as physics, medical, social networks, etc. is growing exponentially. This increase in the volume of data and their large scale make the problem of their processing more complex. Actually, the current datasets are very different in nature, ranging from small to very large, from structured to unstructured, and from largely complete to noisy and incomplete. In addition, these datasets evolve over time, often at very rapid rates. If we consider the characteristics of these datasets, traditional data management systems are not adapted to support them. For example, Relational Database Management Systems (RDMS) manage only databases where data conforms to a schema. However, current databases contain a mix of structured and less or no structured data. Furthermore, relational systems lack support for version management that is very important in a data management system. As data management system dedicated to large-scale datasets, we consider the BlobSeer system. It is a concurrency-optimized data management system for data-intensive distributed applications. BlobSeer is adapted for target applications that handle massive unstructured data in the context of large-scale distributed environments. It uses the concept of versioning for concurrent manipulation of large binary objects in order to exploit efficiently access to data. To reach this objective, BlobSeer uses a versioning manager to generate a new snapshot version of a BLOB every time it is written or appended by a client. But if the number of BLOBs created or the primitives (writing, appending or reading) increase and are managed by a single version manager, then we have a performance bottleneck and a version manager overload. To avoid the bottleneck of the version manager, we propose a multi-version managers, such that each version manager maintains a subset of BLOBS.*

Keywords: Cloud Computing, Data Management System, BlobSeer, BLOBs, Scalability, Multi-version Managers

Received: 28 November 2018, Revised 31 January 2019, Accepted 9 February 2019

DOI: 10.6025/jnt/2019/10/2/40-53

1. Introduction

Cloud computing offers real potentialities for organizations to change their Information Technology Infrastructures (ITI). The use of cloud computing has potential benefits to organizations, including increased flexibility and efficiency. Cloud computing describes a computing concept where software services, and the resources they use, operate as a virtualized platform across many different host machines, connected to the Internet or to a local network. However, the use of cloud computing presents significant challenges to the users of clouds (both individuals and organizations). Traditional data management systems have some limitations for data-intensive applications. Among distributed data management systems, BlobSeer targets applications that handle massive unstructured data in the context of large-scale distributed environments. BlobSeer system was proposed to comply with the following properties: massive unstructured data, data striping, distributed metadata management, high throughput under high level of concurrency. An important feature of BlobSeer is versioning that allows to roll back data changes, but also to execute the same request independently on different versions of a Binary Large OBject (BLOB). In traditional data management systems (such as DBMS), the write operations on a given object are done on a single copy of this object. This approach preserves the coherence property. In BlobSeer, the update operation on an object is very different. It is done by generating a new object at each update. Thus, we have several successive versions of the same object that give the content evolution of this object during a given period. To follow this evolution, BlobSeer uses Metadata (changes, location of different versions of the same object, etc.) and thus old and new versions of an object are saved and can be both accessible. All these operations are made by a single version manager. The use of a single version manager has some limitations and drawbacks: (i) overloading the version manager, which can lead to failure or degradation of the manager performance; (ii) if the number of objects and update primitives increase, this leads to a problem of data storage (physical locations and geographic distribution); (iii) preserving data coherence of replicated objects. To avoid synchronism between update operations, the version manager must provide a consistency protocol to ensure that different versions of the same object do not diverge (versions with contradictory values). The version manager ensures the serialization of the concurrent update primitives and the assignment of version numbers for each new update operation; so, this serialization step does not become a bottleneck when a large number of clients concurrently update a specific BLOB. To avoid the limitations caused by a single version manager, we propose in this paper a new approach based on multiple version managers. The rest of this paper is organized as follows: Section 2 reviews some related works on distributed data management systems. In Section 3, we review the BlobSeer system and list some of its limitations. Section 4 describes our proposal model. Section 5 explains, in details, the different functionalities of our proposal. In Section 6, we compare the traditional BlobSeer (with a single version manager) with our proposal (with multiple version managers). Finally, Section 7 concludes and suggests some directions for further research.

2. Overview on Distributed File Systems

In the last years, increasing number of organizations are facing the problem of explosion of data and the size of has been growing at exponential manner. Data is generated through many sources like transactional systems, web servers, social networks, mobile devices, etc. From the structure view point, these data range from structured to unstructured form. New infrastructures have also emerged to support the execution of large-scale applications, such as grid and cloud computing. As the amount of data increases, the need to provide efficient and reliable storage solutions has become a challenge for researchers. Nowadays, the principle storage solution used by cluster infrastructures is a Distributed File System (DFS). A DFS has three main properties: transparency, fault-tolerance and scalability. In the following, we analyze and compare some DFS's. PVFS [11] is an open source parallel file system for clusters. A file is partitioning into stripe units and they are distributed to disks in a round-robin method. PVFS is composed of one metadata server and several data servers. File System (GFS) [8] is a scalable distributed file system that supports the heavy workload at the Google Website and runs on a cluster with inexpensive commodity hardware. A GFS cluster consists of a single master and multiple chunkservers and is accessed by multiple clients; the master maintains all file system metadata and data are splitted into chunks of 64MB and stored in chunkservers. It uses standard copy-on-write technique to implement snapshots. The Hadoop Distributed File System HDFS [2, 4, 12] is a file system designed for large clusters and many clients. Furthermore, it is file system component of Hadoop framework [1]. HDFS splits a file into one or more blocks and set of blocks are stored in DataNodes. HDFS is a highly fault-tolerant system and it is designed to be deployed on low-cost hardware. In HDFS, each block is replicated three times. HDFS kept the hole metadata in the memory of a single NameNode. So, by the increasing number of metadata, the NameNode may become performance bottleneck. Bigtable [3] is a distributed storage system for managing structured data that is designed to scale to a very large size. Bigtable is built on several other pieces of Google infrastructure. It uses the distributed Google File System (GFS) [8] and Bigtable maintains data in lexicographic order by row key. The row range for a table is dynamically partitioned. Each row range is called a tablet, which is the unit of distribution and load balancing. Each cell in Bigtable can contain multiple versions of the same data, and these versions are indexed by timestamp.

Dynamo [17] is a highly available and scalable key-value store used for storing state of many core services of Amazon.com's e-commerce platform. It cannot be publicly accessed, and it is used for storing only internal state information for Amazon's services. Dynamo partitions the keys with consistent hashing [7] to distribute the load across multiple storage hosts. Dynamo implements eventual consistency, supporting low-latency concurrent data reads and updates through replication and versioning. In [6] Varade et al. said that the architecture of HDFS which used a single NameNode where the metadata are stored and data blocks are stored in DataNodes, can't meet the exponentially increased storage demand in cloud computing, as the single master server may become a performance bottleneck.

Most of existing Distributed File Systems verify the main properties of a DFS, namely transparency, fault-tolerance and scalability. On the other hand, they share a same characteristic: they use one master server and multiple clients. This model with a single master is simple and enables the master to make sophisticated chunk placement and replication decisions using global knowledge. However, the master server can lead to a failure of the whole system.

3. BlobSeer

BlobSeer [9] is a data-sharing system that addresses the problem of efficiently storing massive data in large-scale distributed environments. It deals with large, unstructured data blocks called BLOBs. In BlobSeer, a BLOB is splitted into equally-sized chunks. The architecture of BlobSeer is based on five actors. Figure 1 illustrates it's architecture which is a set of distributed processes that communicate through Remote Procedure Call protocol (RPCs) [13, 15].

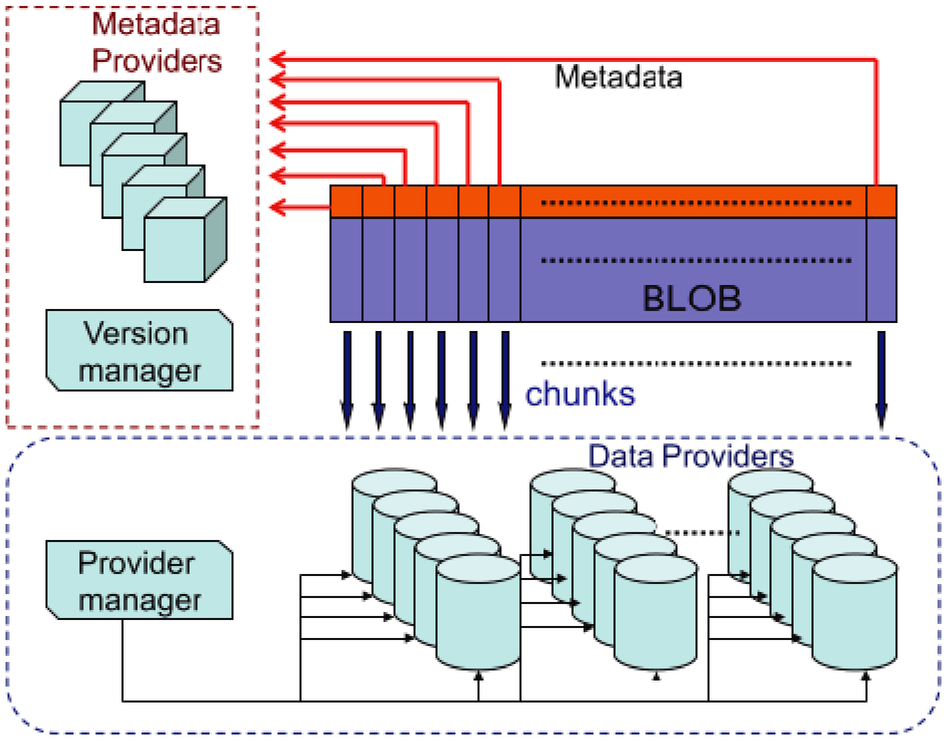

Figure 1. Architecture [9]

• **Clients:** Create, read, write and append data from/to BLOBs. A large number of concurrent clients can simultaneously access the same BLOB.

• **Data Providers:** These actors provide scalable and efficient storage service. Multiple data providers are used to physically store the chunks. Each data provider is simply a local key-value store, and the chunks are accessed from a chunk ID. Data providers can be configured to use different persistent layers such as BerkeleyDB [18], an efficient embedded database, or just keep chunks in main memory. New data providers may dynamically join and leave the system.

• **Provider Manager:** This actor keeps information about the available storage space on data providers. Periodically the data providers send information about their states to the provider manager. To scheduling the placement of newly generated chunks, it employs a configurable chunk distribution strategy to maximize the data distribution benefits with respect to the needs of the application. The default strategy implemented assigns new chunks to available data providers in a round-robin method.

• **Metadata Providers:** The metadata providers physically store the metadata that allow identifying the chunks that make up a snapshot version of a particular BLOB. BlobSeer uses a distributed metadata management organized as a Distributed Hash Table (DHT) to enhance concurrent access to metadata.

• **Version Manager:** This actor is the responsible to assigning new snapshot version numbers to writers and appenders and to reveal these new snapshots to readers. It is done so as to offer the illusion of instant snapshot generation, while guaranteeing total ordering and atomicity. The version manager is the only serialization point in the system. BlobSeer exposes a client interface to make available its data-management service to high-level applications. When linked to BlobSeer's client library, application can perform the following operations: CREATE a BLOB, READ, WRITE and APPEND contiguous ranges of bytes on a specific BLOB.

To understand more precisely the role of each component and specifically the role of the version manager, we explain the functionality of CREATE, WRITE/APPEND and READ primitives. The CREATE primitive exposed by the BlobSeer client library only involves a request for the version manager. This primitive creates a new BLOB. This last will be identified by its ID, and the version manager ensured to be globally unique. When a client wants to update the BLOB, he invokes the corresponding WRITE or APPEND primitives implemented by the BlobSeer client. The WRITE is composed of two main steps [9]:

• **Data-writing Step:** This is the first operation executed when a user calls the WRITE primitive for a specific BLOB and a contiguous range of chunks delimited by the offset (in case of write) of the first chunk to be written, or at the end of the BLOB (in case of append), and the size of the entire sequence. To write such chunk ranges on data providers, the client library performs the following operations:

1. Contact the provider manager and ask him for a number of data providers.

2. Upload the data chunks to the received data providers. In parallel, data-publication operation is executed to make the uploaded data chunks available to the users as a new BLOB version.

3. The client contacts the version manager to notify it about its WRITE primitive. The version manager assigns a version number to the client and adds the WRITE into a queue containing WRITE's in progress.

• The client constructs a metadata tree associated with the new version. Only the leaves for the new chunks are built. After creating the metadata nodes, the client sends them in parallel to the metadata providers. Finally, the version manager notified about the readiness of the metadata associated with the new version.

When the client invokes READ primitive, it contacts the version manager, which stores the root information about all the BLOBS in the system to retrieve the root of the metadata tree corresponding to the specific BLOB identifier and BLOB version requested in the READ primitive. Next, the client scans the metadata tree by issuing metadata read primitives on the metadata providers. It stopped when it find a portion of the tree that covers the needed chunk range. By reaching the leaves, the client can retrieve the location of the data providers that physically store the data chunks. Finally, the client downloads the data chunks in parallel from the data providers.

4. Proposed Model

In the previous section we have explained the BlobSeer architecture and the functionality of the CREATE, WRITE/APPEND and READ primitives. In this architecture, a single version manager is used. As the version manager deals with the serialization of concurrent WRITE primitives and with the assignment of version numbers for each new WRITE/APPEND primitive and to reveal these new versions to readers. The version manager has to generate new snapshots in the case of concurrent WRITE primitives to guarantee the following properties: (i) liveness; (ii) total ordering; and, (iii) atomicity [9]. When the READ and WRITE/APPEND primitives increase, the version manager can become a bottleneck in the BlobSeer standard architecture. To overcome this bad situation, we proposed a new architecture of BlobSeer. Our contribution lies in the use of several version managers since the BLOBs are independent from one another. Each version manager will manage a number of BLOBs and each version manager has a logical number that identifies him in the system. If a client wants to create a new BLOB, it chooses one of the version managers randomly and contacts him. If memory space and CPU usage ratio of the chosen version manager does not exceed a predefined threshold, then it creates the BLOB, otherwise it retransmits the request of creating new BLOB to the underloaded version manager in the system. The version manager will create the BLOB assign him the following identifier (see Figure 2): the first item (Field1) represents the logical number of the version manager, while the second one (Field2) is a number that is incremented locally after each creating of a new BLOB.

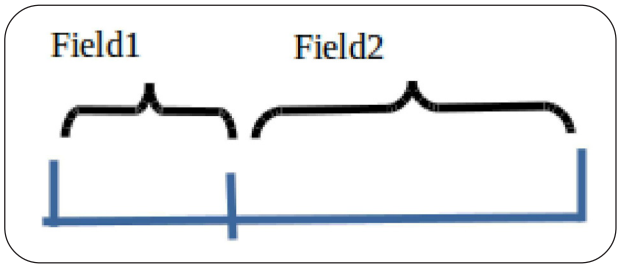

Figure 2. Identifier structure of a new BLOB

Version managers communicate between them by exchanging their memory and CPU workload for load balancing purpose. For example, if a version manager's workload exceeds its predefined threshold, it retransmits the CREATE primitive to the less (memory space and CPU usage ratio) version manager.

4.1 Election of Controller Version Manager

To allow our architecture to be scalable and respond to growing client requests CREATE, READ and WRITE primitives, we

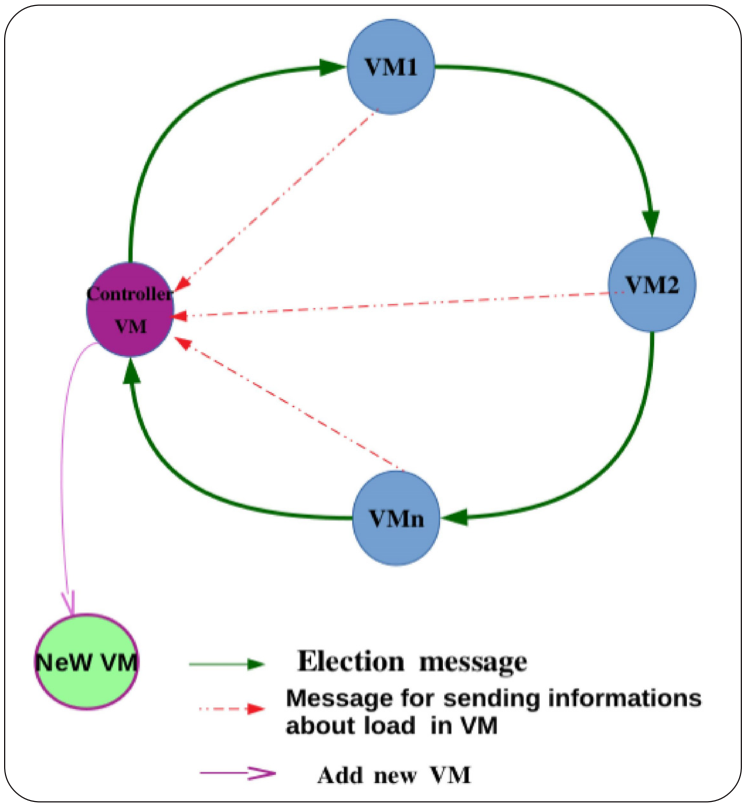

Figure 3. Version managers communication

propose to elect a Controller Version Manager (CVM) using a ring election algorithm [5, 14]. The election starts when one of the version managers wants to contact the current CVM, but the connection fails after two tries. Initially, the CVM is defined in the configuration file of the system. The controller node allows to increase the scalability of our model by adding new version manager, if it is necessary. To ensure this property, all version managers send information about their memory and CPU workloads to the controller version manager. When more than half of the existing version managers exceeds their workload, the controller version manager decides to add new version manager. Figure 3 illustrates this functionality of our model.

5. Illustrative Examples

To explain the behavior of our proposed model, we consider three examples related to three primitives: CREATE, READ and WRITE.

5.1 CREATE Primitive

We suppose that we want to create 20 BLOBs and the system is composed of 4 version managers.

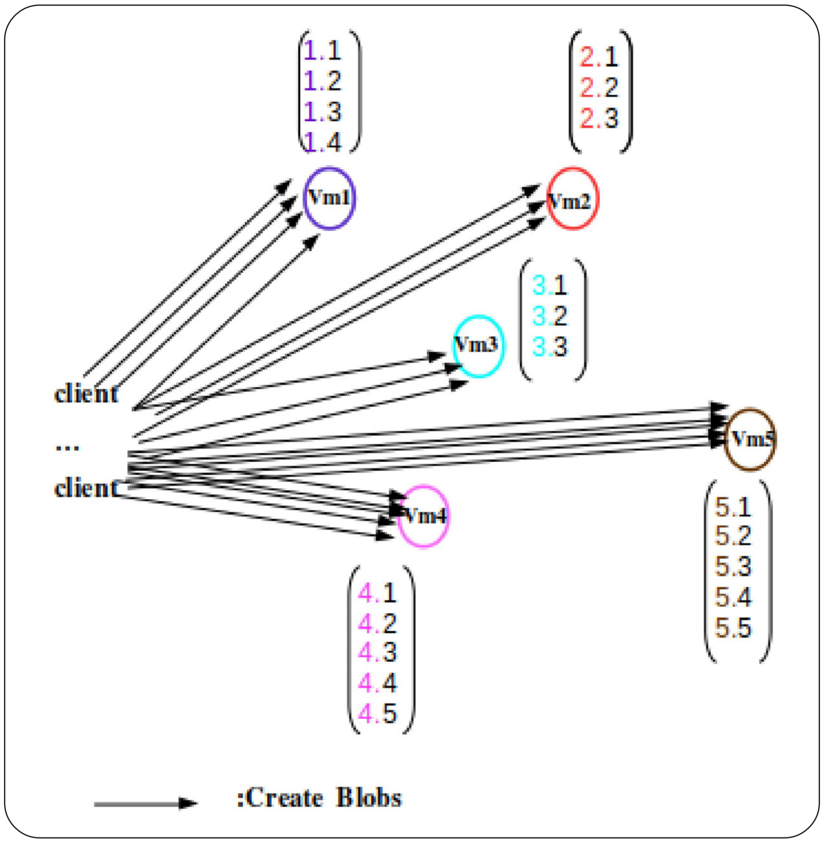

Figure 4. Example of CREATE primitive in modified BlobSeer

As mentioned in Figure 4, to create 20 BLOBs, clients contact the Version manager 1 (VM1) to create 4 BLOBs, the Version Manager 2 (VM2) to create 3 BLOBs, the Version Manager 3 (VM3) to create 3 BLOBs, the Version Manager 4 (VM4) to create 5 BLOBs and Version manager 5 (VM5) to create 5 BLOBs. After each BLOB creation, the corresponding version manager *VMi* (*i* = 1..5), sends the ID-BLOBs of created BLOBs to the client.

Firstly, the READ primitive is a function with 5 parameters: READ (ID, v, buffer, offset, size). A READ primitive accesses a contiguous segment (specified by an offset and a size) from a BLOB (specified by its ID) and copies it into a given buffer. The desired version of the BLOB from which the segment must be taken can be provided in v. In case when parameter v is missing, BlobSeer assumes by default that the latest version of the BLOB is accessed [10, 16].

Following the identifier structure of a BLOB (see Figure 2), it extract the ID of the version manager where the BLOB is created (first field). In the subsequent example, we consider the following assumptions for the first field of version manager ID: (i) number of version manager 1 (blue number); (ii) number of version manager 2 (pink number); (iii) number of version manager 5 (RedViolet number). For the second field, we use a black number.

To read data from BLOB 1.2, we execute the following steps:

1. The client contacts the version manager 1 after extracting the version manager number from the ID (1.2) of the BLOB (the first field). The Version manager 1 sends the latest version of the BLOB 1.2 to the client. The client compares the version requested with the latest version; if it is higher than the current version, the READ primitive fails.

2. The client queries metadata providers for metadata indicating which providers store chunks corresponding to the requested subsequence in the BLOB delimited by offset and size.

3. The client fetches the chunks in parallel from the data providers.

To read data from BLOB 2.40 and from BLOB 5.200 BLOB, steps 2 and 3 are the same as in reading the BLOB 1.2. The only difference is in step 1 which version manager have to be contacted for reading the BLOB 2.40:

1. The client contacts the Version manager 2 after extraction version manager's number from the ID (2.40) of the BLOB (the first field), that is equals to 2. To see if the version of the BLOB are asked to read or not, the Version manager 2 will send the latest version of the BLOB 2.40 to the client. This later compares the version requested with the latest version; if it is higher than the current version, the READ primitive fails.

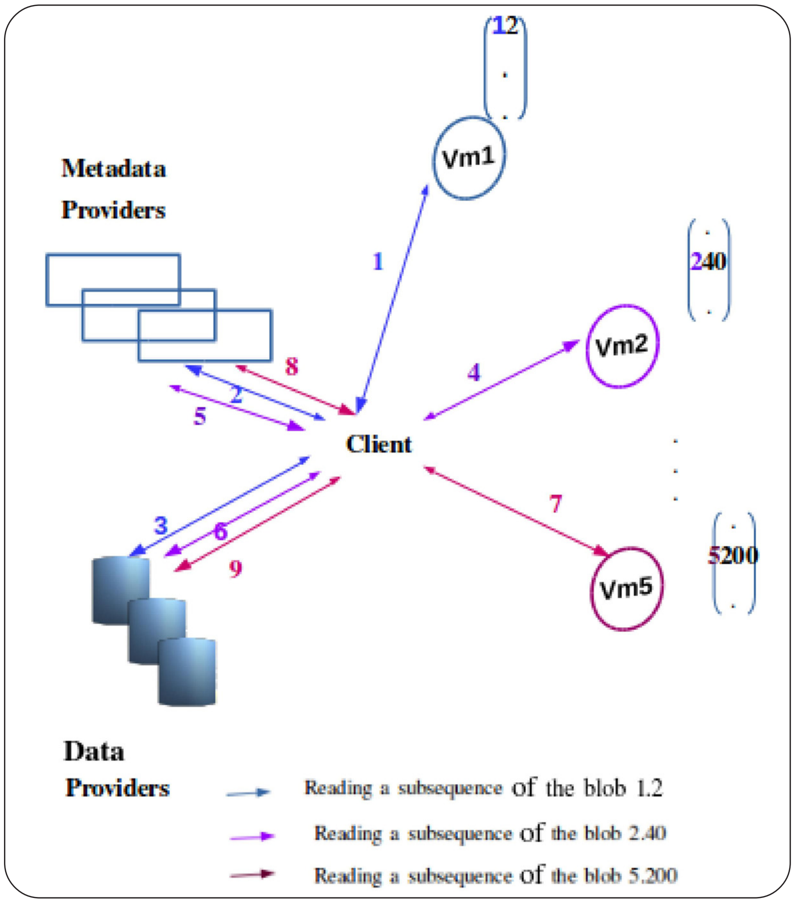
12
Vm1
Metadata Providers
1
240
Vm2
8
4
2
5
Client
7
3
6
9
5200
Vm5
Data Providers
Reading a subsequence of the blob 1.2
Reading a subsequence of the blob 2.40
Reading a subsequence of the blob 5.200

Figure 5. Example of READ primitive in modified BlobSeer

When we want to read data from BLOB 5.200, we execute the following steps :

1. The client contacts the Version manager 5 after extraction of the version manager number from the ID (5.200) of the BLOB (the first field); in this case, the version manager number is equals to 5. To see if the versions of the BLOB are asked to read or not, the version manager 5 will send the latest version of the BLOB 5.200 to the client. This latter compares the requested version with the latest version; if it is higher than the current version, the READ primitive fails.

5.2 WRITE primitive
WRITE (ID, buffer, offset, size) and APPEND (ID, buffer, size) are two primitives that modify a BLOB identified by the parameter ID, by writing content of a buffer of length size at a specified offset or at the end of the specified BLOB. These primitives generate a new version of the BLOB with a new version number generated by the version manager. Remember that WRITE and APPEND primitives only allow updating a contiguous range within a BLOB.

1. To write data in BLOB 1.2, the client first splits the data to be written in chunks.

2. Then, it contacts the provider manager and informs it about the chunks to be written. Using this information, the provider manager selects a data provider for each chunk then builds a list that is returned to the client. Having received this list, the client contacts all providers in parallel and sends the corresponding chunk to each of them.

3. As soon as a data provider receives a chunk, it reports success to the client and caches the chunk which is then asynchronously written to the disk in the background.

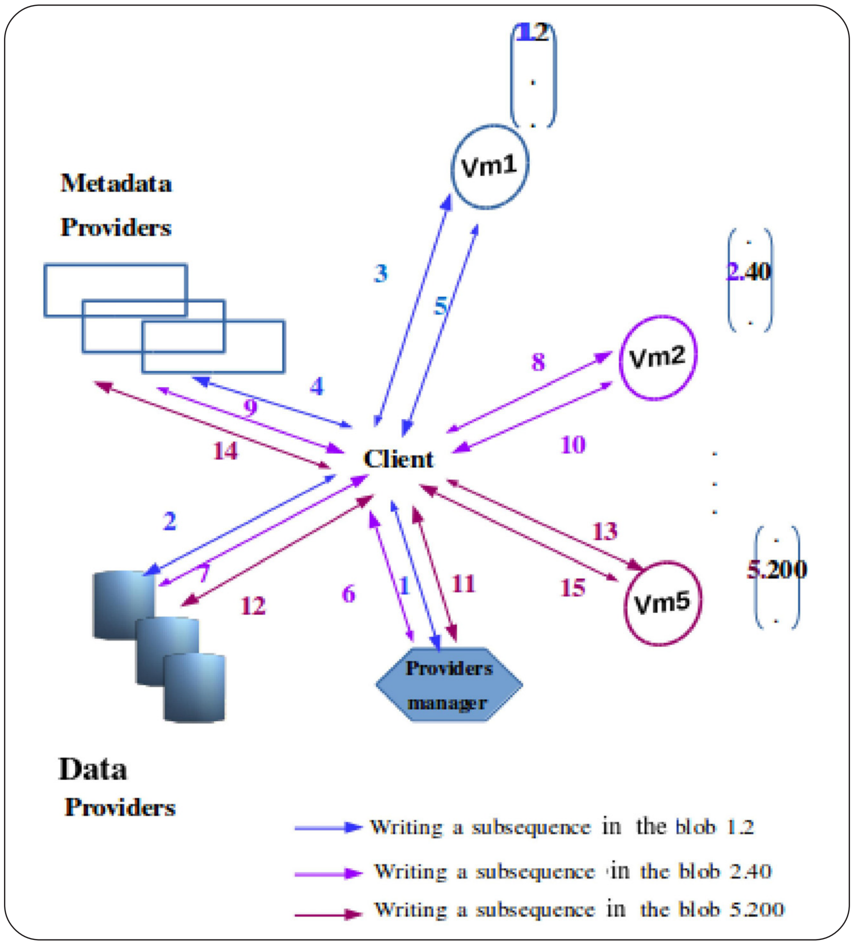
1.2
Vm1
Metadata Providers
3
5
2.40
8
Vm2
4
9
14
Client
10
2
13
5.200
7
6
1
11
15
Vm5
12
Providers manager
Data Providers
Writing a subsequence in the blob 1.2
Writing a subsequence in the blob 2.40
Writing a subsequence in the blob 5.200

Figure 6. Example of Write in modified BlobSeer

4. The client contacts the version manager 1 (the ID of the version manager is extracted from the ID of the BLOB) and registers its update. The version manager 1 assigns to this update a new snapshot version *v* and communicates it to the client.

5. Then, new metadata is generated for the BLOB that is weaved together with the old metadata such that the new snapshot v appears as a standalone entity.

6. Then, it informs the manager version 1 of success, and returns successfully to the users. At this point, the version manager takes responsibility for finally reveal version v of the BLOB to readers.

To write data to BLOB 2.40, the same steps are done with only one difference in steps 4 and 6 instead of contacting the version manager 1, it will contact the version manager 2. Similarly to write data in BLOB 5.200 it will contact the version manager 5.

5.3 Creating BLOBs in standard BlobSeer
In the standard BlobSeer architecture, all CREATE commands are sent to a single version manager. After the creation of a BLOB, the version manager will send the ID of the created BLOB to the client. If we consider the same example presented in the previous section, i.e. the demand for creating 20 BLOBs (see the following figure), 20 BLOBs creation requests are sent to a single Version manager. This latter will send the ID's of the created BLOBs to the client.

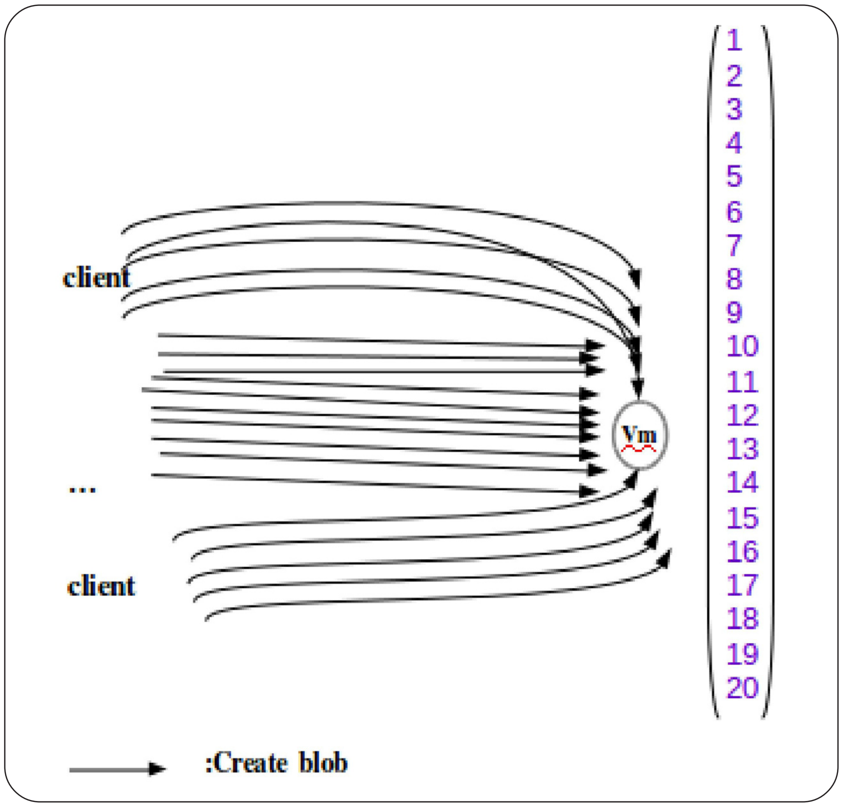

Figure 7. Example of Create BLOBs in standard BlobSeer

5.4 READ Primitive
The following steps are executed to read data from BLOB2, BLOB40, BLOB200 or any BLOB

1. The client sends a message to the single and the only version manager in the architecture, to know if the version of the required BLOBs are asked to read or not. If it is, the version manager will send the latest version of the required BLOBs to the client. At this stage, the client compares the requested version with the latest version. If this latter is higher than the current version, the READ primitive fails. If not, we execute the subsequent steps:

2. The client queries metadata providers for metadata indicating which providers store chunks corresponding to the requested sequence of BLOBs delimited by offset and size parameters.

3. Once the connection to these data providers has been established, the client gets the chunks in parallel from data providers.

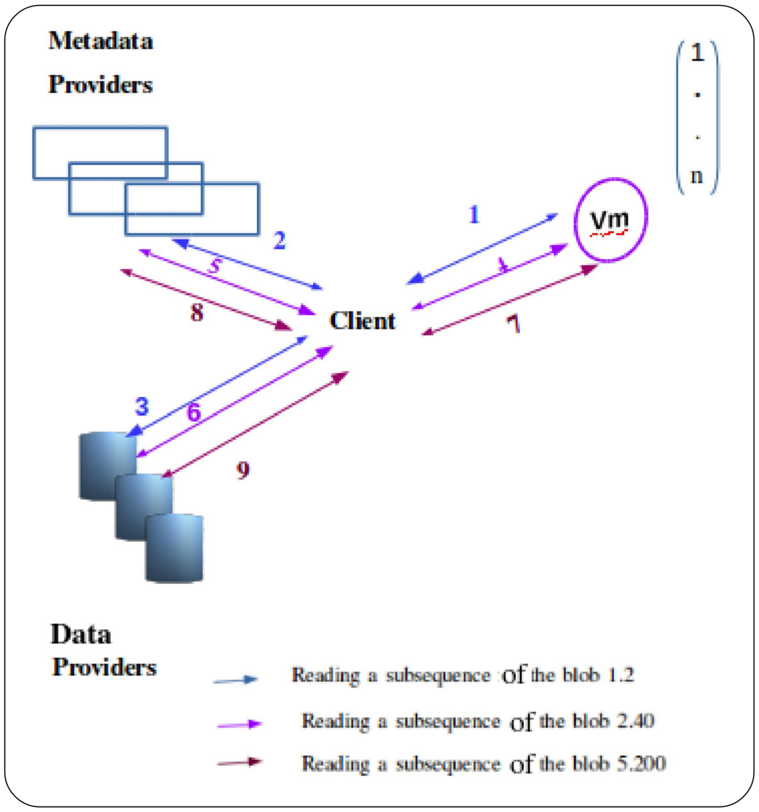
Metadata Providers
1 . . n
Vm
Client
1
2
3
5
6
7
8
9
Data Providers
Reading a subsequence of the blob 1.2
Reading a subsequence of the blob 2.40
Reading a subsequence of the blob 5.200

Figure 8. Example of Read primitive in BlobSeer standard

5.5 WRITE primitive
The behavior of Write primitive is defined as follows:

1. To write data to BLOB2, BLOB40, BLOB200 or any BLOB, creation's steps are the same than read primitive. Firstly, the client splits the data to be written in chunks.

2. Then, it contacts the provider managers and informs it about the chunks to be written. Using these information, the provider manager selects a data provider for each chunk and it builds a list of providers that is returned to the client. Upon receiving this list, the client contacts all providers in the list in parallel and sends the corresponding chunk to each of them.

3. As soon as a data provider receives a chunk, it reports success to the client and caches the chunk which is then asynchronously written to the disk in the background.

4. The client contacts the main Version manager, records and registers its update. Then, it assigns to this update a new snapshot version *v* and communicates it to the client.

5. The client generates new metadata for BLOBs (2, 40 or 200) that is weaved together with the old metadata such that the new snapshot v appears as a standalone entity.

6. Finally, it informs the main version manager of success, and returns successfully to the users. At this point, the version

manager takes responsibility for finally reveal version V of the BLOB to readers.

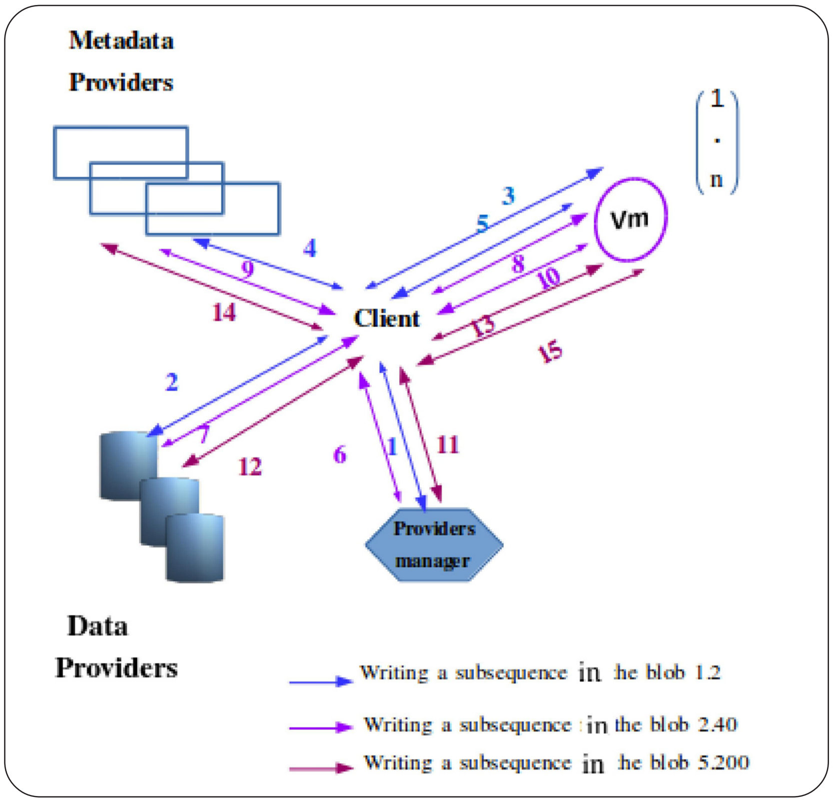
Metadata Providers
1 . n
Vm
Client
Providers manager
Data Providers
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
Writing a subsequence in the blob 1.2
Writing a subsequence in the blob 2.40
Writing a subsequence in the blob 5.200

Figure 9. Example of Write in standard BlobSeer

5.6 HDFS

HDFS (Hadoop Distributed File System) is a distributed file system with a main characteristic: it has been designed to be deployed on low-cost hardware. It has many similarities with existing distributed file systems but it is highly fault tolerant. In addition, HDFS provides high throughput access to applications with large data sets. As architecture, HDFS uses a master/slave model. An HDFS cluster consists of a single NameNode (the master), that manages the file system namespace and orders access to files by clients. In addition to the master server, there are a number of DataNodes, usually one per node. The role of these nodes is to manage storage attached to the nodes that they run on. These nodes are considered as worker nodes in the master/slave model.

In the subsequent sections, we discuss the purpose of three operations on HDFS system: Create, Read and Write.

5.6.1 CREATE File

To create any file, the client makes a Remote Procedure Call (RPC) [13, 15] to the namenode to create a new file in the file system's namespace, without blocks. If the operation is successfully completed, the namenode sends positive acknowledge.

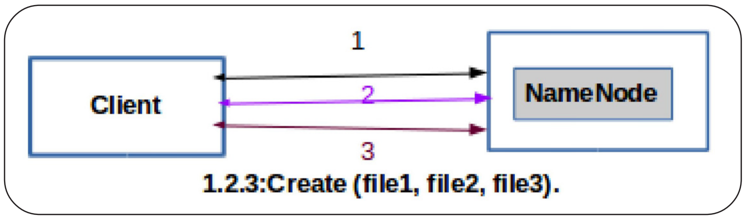

Figure 10. Examples of Create operation in HDFS

5.6.2 READ file

For reading data from a file, the Client communicates with the namenode to obtain metadata information about the locations of data blocks, using RPC protocol [13, 15]. For each block, the namenode returns the addresses of all the datanodes that have a copy of that block. Finally, client will interact with respective datanodes to read the file.

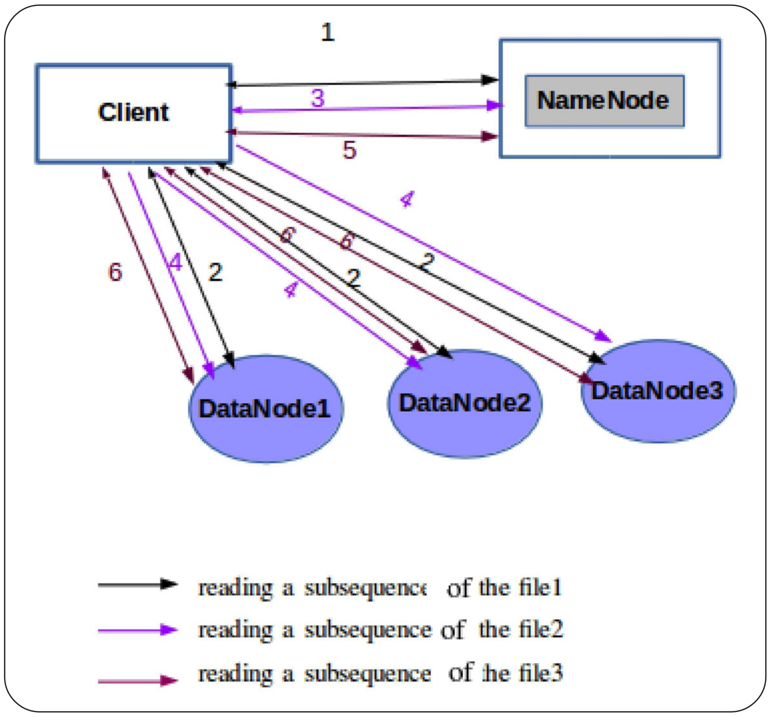
1
Client
3
NameNode
5
4
6
6
6
4
2
4
2
2
DataNode1
DataNode2
DataNode3
reading a subsequence of the file1
reading a subsequence of the file2
reading a subsequence of the file3

Figure 11. Example of Read in HDFS

5.6.3 WRITE file

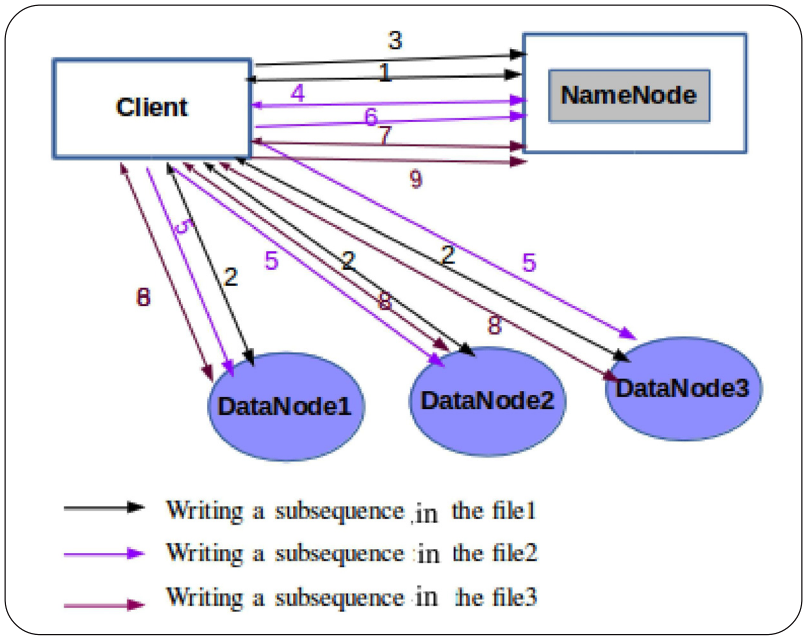
3
1
Client
4
6
7
NameNode
9
5
2
5
2
2
5
8
8
8
DataNode1
DataNode2
DataNode3
Writing a subsequence in the file1
Writing a subsequence in the file2
Writing a subsequence in the file3

Figure 12. Example of Write in HDFS

For writing data in file1, file2 or file3 (see Figure 12), the client splits a file into packets, then it solicits the namenode to allocate new blocks by picking a list of suitable datanodes to store the replicas of the packets. The list of datanodes forms a pipeline. In our example, there are three nodes in the pipeline. The client streams the packets to the first datanode in the pipeline, which stores the packet and forwards it to the second datanode and then to the third datanode. When the client has finished writing data (blocks)in the three replicas, the client sends a signal to the namenode indicating that the file is complete. The namenode already knows the blocks that make up the file.

6. Discussion

If we compare the standard BlobSeer with modified BlobSeer proposed in this paper, we find that the use of multiple version managers instead of a single one allows some flexibility when CREATE/WRITE set operations is large. In addition, our proposal increases the scalability of the data management system because the number of version managers is not fixed in advance and it is not static. We create dynamically new version managers when we rich a given threshold of overload.

BlobSeer supports an efficient fine-grain access to the BLOBs, for a large number of concurrent processes, that is not provided in HDFS. In fact, HDFS provides write-once-read-many access model for files.

By contrast, the combination of a distributed metadata management and multi version managers provides high scalability and increases fault tolerance compared with HDFS where all metadata are stored in one server (NameNode).

7. Conclusion

BlobSeer is a well-known concurrency-optimized data management system for data-intensive distributed applications. It is adapted for applications that handle massive unstructured data in large-scale distributed environments. Its main component is the version manager that create and manage different versions of BLOB objects. The architecture of BlobSeer ensures high-performance and avoids synchronization to achieve better throughput. But its main problem is the existence of a single version manager for the whole system. In fact, when the number of WRITE/APPEND primitives increase, the version manager can become overloaded, and in turn lead to a failure of the system. To overcome this bad situation, we proposed to use multi-version managers instead one single. In fact, the use of a single version manager is a point of congestion that cause a bootleneck, that has a significant impact on the system. In standard BlobSeer, version manager has been built to support only a finite limited number of WRITE/APPEND primitives. In our proposal model, we overcome this limitation by introduction more than one single version-manager.

As future works, we plan to achieve more experimentations our proposal to compare its performance with similar distributed data management systems for large-scale applications.

References

[1] Hadoop. http://hadoop.apache.org.

[2] Hdfs. the hadoop distributed file system. http://hadoop.apache.org/common/docs/r0.20.1/hdfs-design.html.

[3] Chang, F., Dean, J., Ghemawat, S., Wilson, C., Hsieh, D. A., Wallach, Burrows., M., Chandra, T., Fikes, A., Gruber, R. E. (2006). Bigtable: A distributed storage system for structured data. *In*: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation, 7. USENIX Association, 2006.

[4] Shvachko, K. (2010). The hadoop distributed file system. *In*: Mass Storage Systems and Technologies (MSST), IEEE 26th Symposium on IEEE, 2010.

[5] Garcia-Molina, H. (1982). Elections in a distributed computing system. *IEEE Transactions on Computers,* C-310:48–59, 1982.

[6] Varade, M., Jethani, V. (2013). Distributed metadata management scheme in hdfs. *International Journal of Scientific and Research Publications*, 3(5).

[7] Karger, D., Lehman, E., Leighton, T., Tom, Panigrahy, R., Levine, M., Lewin, D. (1997). Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. *In*: Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing (El Paso, Texas, United States,), p. 654–663, Press, New York, NY, May 04 - 06 1997.

[8] Ghemawat, S., Gobioff, H., Leung, S. (2003). The google file system. *In*: ACM Symposium on Operating Systems Principles, 29–43, Lake George, October 2003. NY.

[9] Nicolae, B., Antoniu, G., Bougé, L., Moise, D., Carpen-Amarie, A. (2011). Blobseer: Next-generation data management for large scale infrastructures. *Journal of Parallel and Distributed Computing*, 71.169–184, (February).

[10] Nicolae, B. (2010). BlobSeer: Towards efficient data storage management for large-scale, distributed system. PhD thesis, Université de Rennes 1, Rennes, France.

[11] Ross, R. B., Carns, P. H., Ligon III, W. B., Thakur, R. (2000). Pvfs: A parallel file system for linux clusters. *In*: 4th Annual Linux Showcase and Conference, 317–327.

[12] Shvachko, K., Huang, H., Radia, S., Chansler, R. The hadoop distributed file system. *In*: Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), MSST '10, 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[13] Gomes Soares, P. (1992). On remote procedure call. *In:* IBM Press, editor, Proceedings of the 1992 conference of the Centre for Advanced Studies on Collaborative research, CASCON '92, 215–267, 1992.

[14] Andrew, S., Tanenbaum, Van Steen, M. (2006). Distributed Systems: Principles and Paradigms (2Nd Edition). Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[15] Tay, B. H., Ananda, A. L. (1992). A survey of asynchronous remote procedure calls. *SIGOPS Oper. Syst. Rev.*, 26 (2) 92–109, (April).

[16] Tran, V-T. (2013). Scalable data-management systems for Big Data. PhD thesis, École Normale Supérieure de Cachan - ENS Cachan, 2013.

[17] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W. (2007). Dynamo: Amazon's highly available key-value store. *In*: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles, 205–220, New York, NY, USA, 2007. ACM.

[18] Yadava, H. (2014). The Berkeley DB Book. Apress, Berkely, CA, USA, 1st edition, 2014.