computers

MDPI

Article

Providing Consistent State to Distributed Storage System

Laskhmi Siva Rama Krishna Talluri [1,†], Ragunathan Thirumalaisamy [2,†], Ramgopal Kota [3,†], Ram Prasad Reddy Sadi [4,†], Ujjwal KC [5,†], Ranesh Kumar Naha [5,*] and Aniket Mahanti [6,7]

1 Department of Computer Science and Engineering, Koneru Lakshmaiah Educational Foundation, Vaddeswaram, Andhra Pradesh 522502, India; sivamca.mtech@gmail.com
2 Department of Computer Science and Engineering, SRM University-AP, Andhra Pradesh 522502, India; ragunathan.t@srmap.edu.in
3 Department of Physical Layer Products, Broadcom, Hyderabad 500032, India; ramgopalkota@gmail.com
4 Department of Information Technology, Anil Neerukonda Institute of Technology & Sciences, Visakhapatnam, Andhra Pradesh 531162, India; reddysadi@gmail.com
5 School of Information and Communication Technology, University of Tasmania, Hobart, TAS 7005, Australia; ujjwal.kc@utas.edu.au
6 School of Computer Science, University of Auckland, Auckland 1010, New Zealand; a.mahanti@auckland.ac.nz
7 Department of Computer Science, University of New Brunswick, Saint John, NB E2L 4L5, Canada
* Correspondence: raneshkumar.naha@utas.edu.au
† These authors contributed equally to this work.

Abstract: In cloud storage systems, users must be able to shut down the application when not in use and restart it from the last consistent state when required. BlobSeer is a data storage application, specially designed for distributed systems, that was built as an alternative solution for the existing popular open-source storage system-Hadoop Distributed File System (HDFS). In a cloud model, all the components need to stop and restart from a consistent state when the user requires it. One of the limitations of BlobSeer DFS is the possibility of data loss when the system restarts. As such, it is important to provide a consistent start and stop state to BlobSeer components when used in a Cloud environment to prevent any data loss. In this paper, we investigate the possibility of BlobSeer providing a consistent state distributed data storage system with the integration of checkpointing restart functionality. To demonstrate the availability of a consistent state, we set up a cluster with multiple machines and deploy BlobSeer entities with checkpointing functionality on various machines. We consider uncoordinated checkpoint algorithms for their associated benefits over other alternatives while integrating the functionality to various BlobSeer components such as the Version Manager (VM) and the Data Provider. The experimental results show that with the integration of the checkpointing functionality, a consistent state can be ensured for a distributed storage system even when the system restarts, preventing any possible data loss after the system has encountered various system errors and failures.

Keywords: distributed system; distributed file system; BlobSeer; Hadoop; consistency; check-point; cloud computing

check for updates

Citation: Talluri, L.S.R.K.; Thirumalaisamy, R.; Kota, R.; Sadi, R.P.R.; KC, U.; Naha, R.K.; Mahanti, A. Providing Consistent State to Distributed Storage System. *Computers* **2021**, *10*, 23. https://doi.org/10.3390/computers10020023

Academic Editors: Paolo Bellavista and Grigorios E. Koulouras

Received: 5 December 2020
Accepted: 8 February 2021
Published: 15 February 2021

1. Introduction

For the past two to three decades, large data generated from a variety of sources such as commercial organizations, educational institutes, social networking sites, research organizations, the gaming industry, and web-based applications are increasing with tremendous velocity. Large data cannot be stored in centralized server systems, and thus we require distributed storage and a computing environment for storing, accessing, and processing large data in a scalable manner. In such a case, the natural idea to emerge would be to use the powers of multiple autonomous computers capable of communicating with each other in a network to achieve a common goal. Accordingly, a similar infrastructure referred to as a distributed system (DS) [1,2] was developed. Modern cloud computing systems (CSSs) [3], built on the principles of a distributed system, are capable of providing large storage and computing capacity in a scalable manner [4,5] with high availability and reliability [6] for a wide range of data, computers, and concurrent-access-intensive services [7–12]. Such a capability in a cloud computing system is enabled by distributed file systems (DFS), which is considered to be one of the core components in such distributed systems.

In distributed systems, applications requiring large volumes of data are studied under "data-intensive computing", in which most of the processing time is dedicated to reading/writing or manipulating the data. With the surge in the amount and pace with which the data are generated, the focus of distributed systems such as cloud computing has shifted from compute-intensive to data-intensive domain [13]. The data to be processed in distributed systems can be broadly classified into three categories: structured, semistructured, and unstructured. When it comes to processing a huge amount of data generated at a tremendous rate, it is quite a challenging task to make decisions for those applications even in supercomputers [14]. To address the challenges associated with handling large data sets in distributed systems, several programming models were developed by different internet service providers. Map-Reduce [15–19], a parallel programming model, was developed by big internet service providers to perform computation on massive data sets [15,20] in data-intensive and distributed applications. The model was developed as an open-source project by Apache Hadoop (extension of the original Google File system developed by Google [21]), in which the Hadoop distributed file system (HDFS) [22] was used as a core component in the data storage system. HDFS offers high fault tolerance along with high scalability and is best suited for immutable files and web applications. However, when the files have to be concurrently accessed by multiple applications, HDFS faces difficulties to offer high throughput [23]. Recently, the apache software foundation released a new version 3.2.2 for Hadoop. According to the work in [24], Hadoop cannot guarantee isolated access to data for concurrent file access. BloobSeer [23,25] was developed to overcome this issue as it seamlessly supports concurrent file access, data-intensive applications, and versioning capabilities [26].

In computing systems, the checkpointing technique periodically takes a snapshot of the states in a persistent storage device. Checkpointing has been used as one of the key mechanisms to provide a consistent state to the distributed storage system and ensure fault tolerance. When the system fails at any point during the execution of the applications, the system can recover and restart the application from one of the most recent checkpoints, without having to start the applications from the start. While checkpointing is a topic of interest for both practitioners and researchers, in this paper, we add a checkpointing restart method for BlobSeer DFS as such checkpoints can prevent data loss due to any system failures or restarts, especially in a distributed Cloud environment.

The rest of the paper is organized as follows. Section 2 discusses the background of the Blobseer DFS along with related works on providing consistent state to a distributed storage system. Section 3 explains in detail the proposed technique, while Section 4 presents the experimental results along with discussions. Section 5 concludes the work.

2. Related Works

In this section, we first discuss the background of BloobSeer, and then include the related works on various checkpointing mechanisms.

2.1. BlobSeer Distributed File System

BloobSeer [23,25] was developed to support concurrent file access, data-intensive applications, and versioning capabilities [26]. BlobSeer DFS was developed to store big data, ensuring faster access to smaller parts of huge data objects and high throughput during concurrent data access. In BlobSeer, data are stored as a binary large object, termed as a blob. The blob can then be accessed through an interface. The same interface can be used to create blobs, read from blobs, and write to blobs. Another key feature of BlobSeer is the versioning—the ability to generate a separate identification number (version) for newly created or modified copies of existing blobs. The version number follows an incremental fashion for existing blobs while a random version number is generated for new blobs. BlobSeer ensures high throughput for concurrent access as the clients can access an existing or the most recent version of a blob by specifying the relevant version identification number [27,28]. The framework of BlobSeer DFS is depicted in Figure 1. Data Providers (DPs), Provider Manager (PM), Metadata Providers (MPs), and Version Manager (VM) are the major components of BlobSeer DFS.

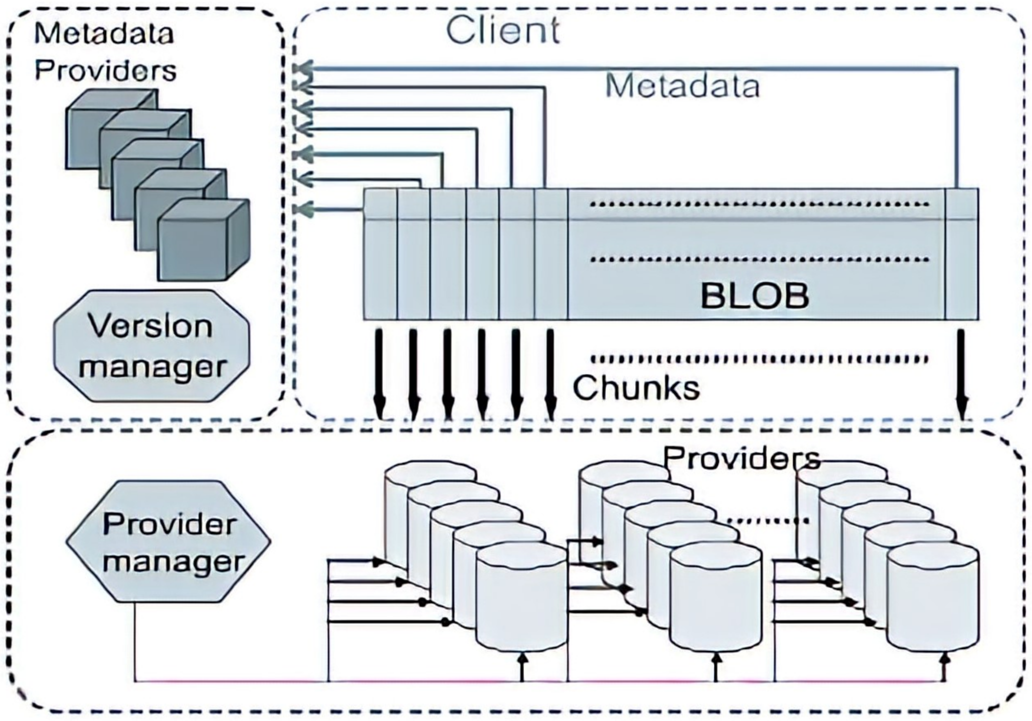


Figure 1. Framework of BlobSeer distributed file system (DFS) [26].

The clients can create, read, write, and append data from/to blobs along with concurrent access to the same data object. DPs store the chunks of data coming out of write or append operations. BlobSeer offers flexibility as new DPs can be added or the existing DPs can be removed from the system when required. PM maintains the information of the available storage space in the system and is responsible for scheduling the location of newly generated data chunks. The metadata information maintained by MP allows the identification of the chunks to make a snapshot version. The distributed metadata management scheme in BlobSeer allows efficient concurrent access to the metadata information. VM in BlobSeer is responsible for allocating version identification values to new write clients and the clients performing append operations on data objects.

The key file operations in BlobSeer DFS are read, write, and append. To read a file or data object, a client initially requests the VM by specifying the version number to be read. The VM then transfers the request to MPs. The MP then sends the requested range of metadata of the corresponding data object to the requested client. Once the client determines the location information of all the pages, the client can read the data from the corresponding providers simultaneously. For the write operation, the client first divides the data into several pages of specified sizes. The client then requests the PM for the list of DPs to write the data pages. After receiving the DP list from the PM, the client performs data write operations to multiple DPs simultaneously. If any of the write operations fail, the entire write operation is considered to be unsuccessful. This arrangement in BlobSeer ensures data consistency. Once the file write operation is successful, the client requests the VM for the assignment of a new version for the data object. The VM assigns the version number based on its version control policy. The file append operation is quite similar to the file write operation with the difference in the range of the offset for the append operation, which is fixed by the VM during the assignment of the version value. In BlobSeer, each data object is divided into fixed smaller parts, which can be specified at the time of data object creation. These fixed smaller parts are evenly distributed among storage space providers based on load balancing strategies, because of which multiple clients can concurrently access the data objects with high throughput. BlobSeer maintains decentralized metadata information to avoid a single point of failure during the concurrent data object access. For metadata information, a distributed segment tree is associated with each version of the data object. A distributed hash table is also used by metadata providers for efficient access to the metadata. Furthermore, BlobSeer implements linearizability semantics to provide concurrent access to data objects. When multiple write clients are accessing different parts of the same data object, the versioning-based control mechanism allows the clients to write all the different parts of the same data object simultaneously until the version-number assignment step. While assigning version numbers, the concurrent clients are serialized. The clients can function simultaneously only after they have been assigned the version number.

To employ BlobSeer for data-intensive applications in distributed systems, such as Cloud infrastructure, we must be able to stop and restart the framework from a consistent state [29]. To offer a consistent state after a system restart, we propose a checkpoint functionality to stop and restart various components of the BlobSeer framework. This checkpoint functionality ensures consistent data operation after any stop and restart events. The proposed technique also facilitates the integration of the BlobSeer framework to virtual machines in Cloud infrastructure to make the framework a storage service in the Cloud by transferring the advantages of the framework. The proposed mechanism is expected to offer a new way, as an alternative to existing methods, for using the BlobSeer framework in a Cloud environment for more efficient data access, storage, and processing.

2.2. Checkpointing Mechanisms

Checkpointing has been used as one of the key mechanisms to provide consistent state to the distributed storage system and ensure a fault-tolerant DS [30]. In this subsection, we describe various approaches adopted in the literature to implement checkpointing in DFS.

In computing systems, the checkpointing technique periodically takes a snapshot of the states in a persistent storage device. When the system fails at any point during the execution of the applications, the system can recover and restart the application from one of the most recent checkpoints, without having to start the applications from the start. In the literature [31,32], three main checkpointing approaches—uncoordinated, coordinated, and communication-induced—have been defined. Further discussion on these approaches is given as follows.

2.3. Uncoordinated Checkpoints

In this approach, while restarting after encountering a failure, the application processes check through an already finalized set of checkpoints and find one in a consistent state. The processes can then be resumed from the chosen consistent state. Usually, processes consider the checkpoints saved in a small space for better efficiency.

The uncoordinated approach has been discussed in the literature [32]. Under this approach, processes identify local checkpoints individually. While restarting, the processes verify for an already finalized set of checkpoints, which are in a consistent state. From this consistent state, execution can be resumed. The advantage of this approach is that each process can consider a checkpoint as per their convenience. Processes can consider checkpoints which are saved in a small amount for efficiency purpose. One of the strong points of this approach is that each process can have the flexibility to take a checkpoint whenever the process requires one. For more efficient operations, the processes are even allowed to consider the checkpointing when the frozen state information size is small.

However, there are some disadvantages to the approach. There is a possibility of the system rolling back to the starting state of the execution that can waste the resources and the work already done after the system has started. Additionally, the processes under this approach may consider checkpoints that are not a part of a globally consistent state. In an uncoordinated approach, initially, all the specified processes figure out the local checkpoints individually. Uncoordinated checkpointing forces each process to maintain multiple checkpoints which can incur a large storage overhead. At the time of a restart, the processes verify the existing list of finalized checkpoints to determine a consistent state from which the execution process can resume.

2.4. Coordinated Checkpoints

The need for multiple processes to coordinate to produce a globally consistent checkpoint state while finalizing the local checkpoints is highlighted in [32]. Accordingly, a coordinated approach was incepted. Due to the coordination among all the processes, the recovery process is simplified and all the processes can start from the most recent available checkpoint state. This prevents the domino effect that was existent with an uncoordinated approach. More importantly, with the coordinated approach, the storage is minimized as only one persistent checkpoint has to be maintained. The limitation of the coordinated approach is the latency while finalizing the checkpoint as the approach requires the global checkpoint to be finalized prior to storing the checkpoints to be written into the permanent storage.

2.5. Communication-Induced Checkpoints

In a communication-induced approach, each process is forced to take checkpoints based on protocol-related information that is "piggybacked" on the application messages received from other processes. The checkpoints are taken in such a manner that the system-wide consistent always exists on the stable storage. This prevents the "domino effect" within the approach. Moreover, the processes are allowed to take some of their checkpoints independently (termed as local checkpoints) as well. However, while deciding on the globally consistent state, the processes are forced to take some additional checkpoints (forced checkpoints). The receiver of each application message uses the piggybacked information to determine if a receiver has a forced checkpoint. The forced checkpoint must be taken before the application can process the contents of the message, which can incur high latency and overhead. In contrast to coordinated checkpointing, no special coordination messages are exchanged in this approach.

In our work, we implement a checkpoint mechanism within a BlobSeer cluster based on uncoordinated checkpoint algorithms for their simplicity and associated benefits. In the future, we expect to explore the possibility of a new checkpoint approach based on speculation.

3. Proposed Approach

So as to ensure a consistent state to a distributed storage system in Cloud infrastructure from where any application processes can resume the operations, we propose an uncoordinated checkpoint-based restart method for the BlobSeer storage layer in a distributed environment. The uncoordinated checkpoint approach has been adopted in our proposed method due to the minimum overhead and efforts required for the approach and its convenient and speedy operation when compared to its counterparts. The uncoordinated checkpointing approach can be easily integrated into each component of BlobSeer without having to expense the overheads for the global state.

Figure 2 shows how an uncoordinated checkpointing can be implemented in the VM within the BlobSeer framework. As can be seen in the figure, checkpointing in the VM starts when a client issues a "CREATE_BLOB" request in the BlobSeer API. When the API is called, the checkpointing function creates the requested blob. The function then stores the blob-id and other relevant parameters. All these values are stored in a file. As there is only one Version Manager (VM) in the BlobSeer framework, there is no requirement for distributing the same information to other nodes. Single file storage is thus sufficient for version checkpointing. Apart from the blob-id, page size and replication count are stored in the file as "space" separated entries. The file is parsed during the restart phase after any failure is encountered in the storage system. In our proposed method, there is a counter that keeps track of the total number of blobs created. During the restart, the counter can start from the last value rather than starting from the zero (0) value. As such, storing the key information in a file as explained in our proposed method, which when parsed during any restart phase provides a consistent state for the VM in a distributed storage system in a distributed environment. The file in our proposed method is stored in ".txt" file format.

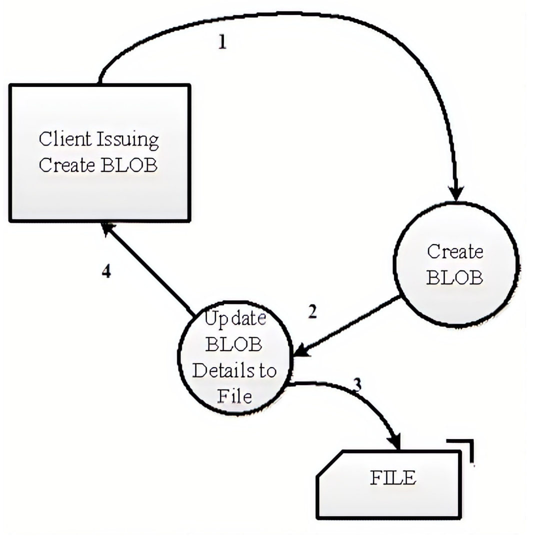


Figure 2. Version manager checkpointing.

Figure 3 represents the processes that follow after the VM restarts after having encountered some failure. Once the VM restarts, the counter value for the total number of BLOBs created is initialized to 0. The VM then looks for any entries in the text file, where each entry in the file represents a new blob. Each entry in the file is read and parsed, based on which a new blob is created and its id is returned.

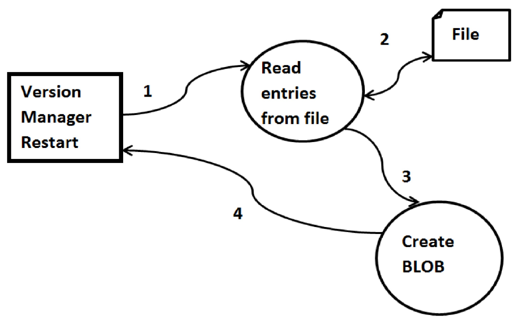


Figure 3. After restart of version manager.

The checkpointing functionality in the metadata provider is shown in Figure 4. The checkpointing starts once the client issues a write or append API. The data required for checkpointing for the data provider and the metadata provider is similar. Both components have <KEY, VALUE> pairs. When checkpoint data is built, the checkpoint data is stored with the key-value pairs and a new version is finally created.

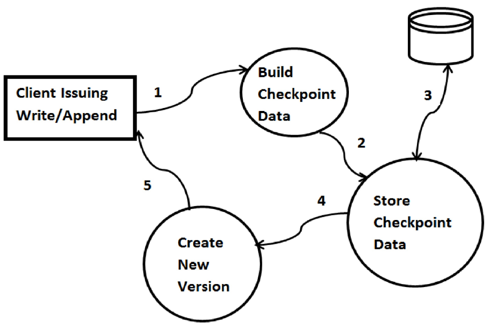


Figure 4. Provider or metadata provider checkpointing.

Figure 5 represents the processes that follow when the data provider or metadata provider restarts. Once the data provider or the metadata provider restarts, the entries in the database are looked for. If there are any entries in the database, those entries are populated to the cache as if they are currently in execution. A segment tree is consequently constructed. The same process is repeated for all the entries in the database. Once the process for all the entries is completed, the data provider or the metadata provider functions usually.

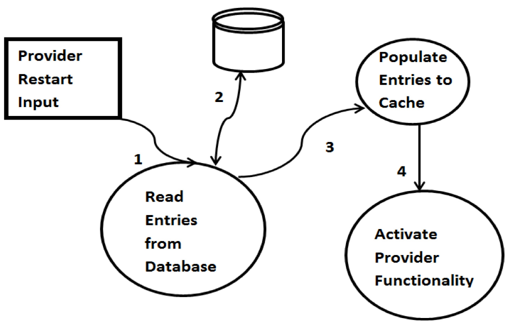

Provider Restart Input
1
Read Entries from Database
2
3
Populate Entries to Cache
4
Activate Provider Functionality

Figure 5. After restarting the provider or metadata provider.

4. Experimental Results

In this section, we demonstrate the implementation of the checkpointing functionality in various components of the BlobSeer framework. We first present the details of the experimental setup and then explain the findings obtained by considering the restart approach for the VM and the data provider.

4.1. Experimental Setup

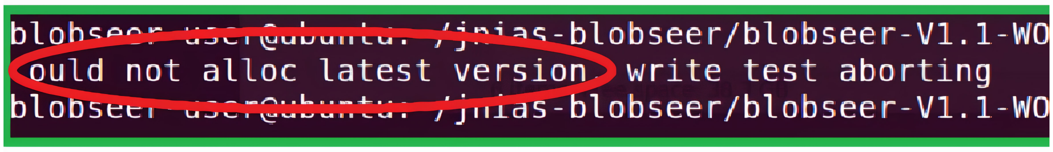
We consider a test environment built on a cluster consisting of five nodes. Each node has an Intel Core 2 Duo processor with 2.9 GHz frequency, 2 GB of RAM, and 500 GB of SATA HDD. The operating system on these nodes is Ubuntu and BlobSeer v1.1 is installed on top of the operating system. Out of the five nodes considered for the test, we install both the metadata provider and data provider in two nodes, the VM and provider manager in one node, and two different data providers in two different nodes. The multiple nodes within the cluster as considered in the experimental setup are taken to replicate the Cloud computing environment with an aim to demonstrate how the checkpointing functionality integrated with various components in the BlobSeer framework can provide a consistent state after restarting the system.

4.2. Check-Point Restart Approach of Version Manager

We demonstrate the checkpoint restart approach of the VM in BlobSeer with two different cases: the first being without the implementation of the restart approach and the second being after the implementation of the restart approach. The demonstration is discussed further as follows.

4.2.1. Without Restart Approach

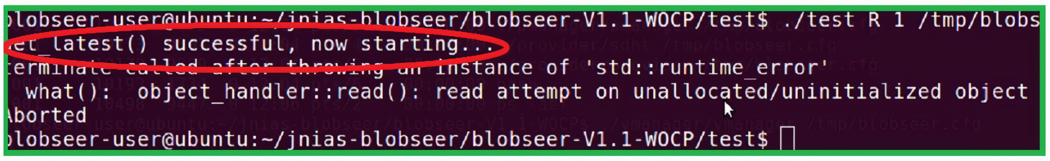
Figure 6 shows the execution of the read operation when the VM is unavailable during the system operation. In this experiment, we perform various normal operations of BlobSeer components: create a new blob, and write to and read from a blob. During the test, the VM was made unavailable by executing a kill operation. Given the role of the VM while executing the usual operations in BlobSeer, while attempting to execute a read operation, the system displayed *"Could not alloc latest version"* message, as can be seen in Figure 6 highlighted by red color.



```
blobseer-user@ubuntu:~/jnias-blobseer/blobseer-V1.1-WO
ould not alloc latest version, write test aborting
blobseer-user@ubuntu:~/jnias-blobseer/blobseer-V1.1-WO
```

Figure 6. Read operation when version manager is unavailable.

In the next step of the demonstration, we restart the VM without implementing the restart (checkpointing) functionality and try to execute the read operation. As can be seen from Figure 7, the VM has restarted, but the read operation was still unsuccessful. This is because the VM did not have any checkpointing functionality to store the version information about the data that was to be read. Consequently, the BlobSeer framework was unable to find a consistent state for the data after the restart and the read operation failed.

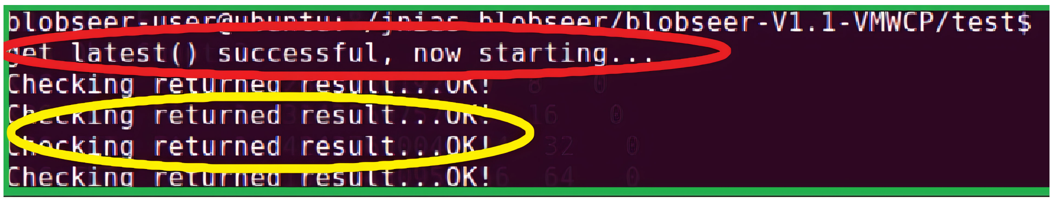


```
blobseer-user@ubuntu:~/jnias-blobseer/blobseer-V1.1-WOCP/test$ ./test R 1 /tmp/blobs
et_latest() successful, now starting...
erminate called after throwing an instance of 'std::runtime_error'
  what():  object_handler::read(): read attempt on unallocated/uninitialized object
Aborted
blobseer-user@ubuntu:~/jnias-blobseer/blobseer-V1.1-WOCP/test$
```

Figure 7. Read operation when version manager is restarted (without restart approach).

4.2.2. With Restart Approach

Figure 8 shows the same test with read operation when the VM is available within the BlobSeer framework. We executed the normal operations of creating a new blob, writing to and reading from a blob by making the VM available. As can be seen from the figure, the text highlighted in the red color confirms the availability of the VM that has the checkpointing functionality enabled. The texts in the yellow color confirm that the test read operation is successful and the blob information is restored. The test read operation was successful when the VM was made available with the checkpointing functionality as the system could retrieve the version information from the VM and get the relevant data object for the requested read operation. As such, checkpointing in the VM provided a consistent state for the data storage after the system restarted after having encountered some problems.



```
blobseer-user@ubuntu:~/jnias-blobseer/blobseer-V1.1-VMWCP/test$
get latest() successful, now starting...
Checking returned result...OK!
Checking returned result...OK!
Checking returned result...OK!
Checking returned result...OK!
```

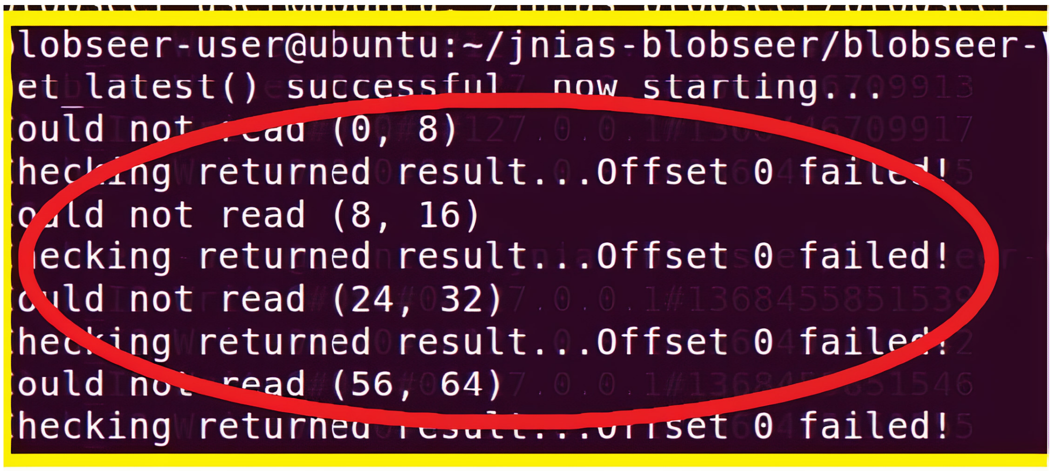
Figure 8. Read operation when version manager is restarted (with restart approach).

4.3. *Checkpoint Restart Approach of Data Provider*

We conduct similar experiments with the data provider within the BlobSeer framework to demonstrate the availability of a consistent state for data storage and operations with checkpointing functionality in data provider, yet another component in the BlobSeer framework. Further discussion on the findings with and without restart approach is made as follows.

4.3.1. Without Restart Approach of Data Provider

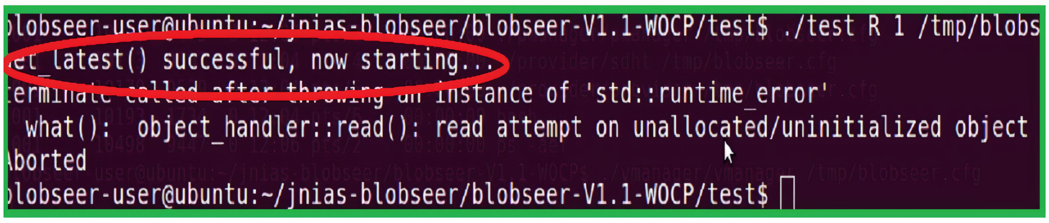
In a similar approach to our experimental tests, we tried to execute various normal operations (creating a new blob, reading from, and writing to a blob) in the system. For the test without the restart approach, we killed the data provider by executing a kill operation. We then ran a read from a blob operation to read the contents of the blob. Figure 9 shows the outputs obtained while attempting to run the read operation when the data provider was unavailable. As can be seen from the figure, it is clear that the read operation failed. The text *"Could not read"* confirms the failure of the read operation, which was all because of the unavailability of the data manager.



```
lobseer-user@ubuntu:~/jnias-blobseer/blobseer-
et_latest() successful, now starting...
ould not read (0, 8)
hecking returned result...Offset 0 failed!
ould not read (8, 16)
ecking returned result...Offset 0 failed!
uld not read (24, 32)
hecking returned result...Offset 0 failed!
ould not read (56, 64)
hecking returned result...Offset 0 failed!
```

Figure 9. Read operation performed when the data provider is unavailable.

In the next step, the data manager was made available and the same read operation was executed. Figure 10 shows the set of outputs that were obtained during the test. As can be seen from the texts highlighted within the red color, the data manager was available to the system during the test. However, the read attempt still failed. This is because the data manager, without the checkpointing functionality, did not have the required information about the blob which the read operation was attempting to access. Thus, the BlobSeer framework in our test with the data manager could not provide a consistent state after the data manager restarted after having encountered some failure.

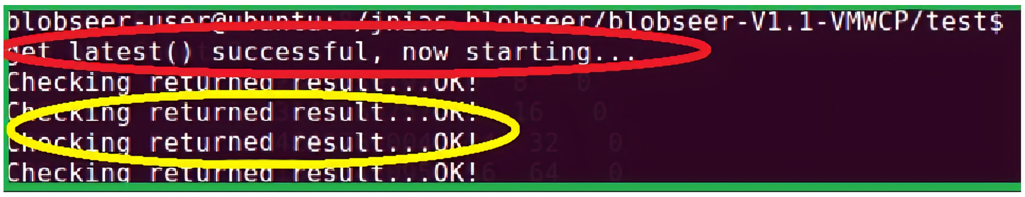


```
lobseer-user@ubuntu:~/jnias-blobseer/blobseer-V1.1-WOCP/test$ ./test R 1 /tmp/blobs
et_latest() successful, now starting...
erminate called after throwing an instance of 'std::runtime_error'
  what():  object_handler::read(): read attempt on unallocated/uninitialized object
borted
lobseer-user@ubuntu:~/jnias-blobseer/blobseer-V1.1-WOCP/test$
```

Figure 10. Read operation is performed when the data provider is restarted (without restart approach).

4.3.2. With Restart Approach of Data Provider

In an attempt to explore and investigate the availability of a consistent state with checkpointing functionality integrated to the data provider in the BlobSeer framework, we re-conducted the same experiment with normal operations by making the data manager available and enabling the checkpointing functionality in the data manager. Figure 11 shows the list of output statements obtained during the test experiments while trying to execute the read operation on a blob. As can be seen from the texts within the red color in the figure, the data manager was available during the read operation. The texts in the yellow color confirm the successful completion of the read operation and the restoration of the blob information. As such, as demonstrated in our test, we can conclude that the checkpointing functionality, as enabled in the data provider within the BlobSeer framework, provided a consistent state for the data storage and operations in a distributed environment after the system restarted.



```
blobseer-user@ubuntu: /jnias blobseer/blobseer-V1.1-VMWCP/test$
get latest() successful, now starting...
Checking returned result...OK!        8     0
Checking returned result...OK!        16    0
Checking returned result...OK!        32    0
Checking returned result...OK!        64    0
```

Figure 11. Read operation is performed when the data provider is restarted (with restart approach).

5. Conclusions

BlobSeer DFS is a data storage system that was built as an alternative solution for existing popular open-source data storage system (such as HDFS developed by Hadoop) in distributed systems. BlobSeer DFS can be used in emerging distributed technologies such as cloud computing but, one of the limitations with BlobSeer is the possibility of losing data on the system restart. For distributed systems, the machines can restart for various reasons as such systems are prone to faults and system errors. For seamless integration of flexible BlobSeer framework in distributed systems to take advantages of its features, it is imperative to ensure the data are not lost on every occasion of the system restart. As such, in this paper, we implemented checkpointing restart functionality in various components of the BlobSeer framework to offer consistent start and stop state for system restarts. The facilitation of the checkpointing functionality within the BlobSeer framework provided a consistent state for the data storage and operations in a distributed system. We conducted different tests in a real-life test environment with a cluster of multiple machines. The experimental results demonstrate the ability of the BlobSeer framework to provide a consistent state to distributed data storage system with the checkpointing functionality enabled. The BlobSeer DFS was able to start from a previous consistent state after system restarts and no data was lost to system restarts as the framework was able to recover from the consistent state provided by the checkpointing functionality. The findings from this study can be carried forward to other components within a distributed system to ensure no data is lost to any system failures or errors that may arise in such systems for various known and unknown reasons.

Author Contributions: Conceptualization, L.S.R.K.T.; investigation, L.S.R.K.T., R.P.R.S. and A.M.; methodology, L.S.R.K.T., R.K. and R.P.R.S.; project administration, L.S.R.K.T. and R.T.; supervision, R.T.; validation, L.S.R.K.T.; writing—original draft, L.S.R.K.T.; writing—review & editing, R.T., R.K., U.K., R.K.N. and A.M. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Krishna, T.L.S.R.; Ragunathan, T.; Battula, S.K. Improving performance of a distributed file system using a speculative semantics-based algorithm. *Tsinghua Sci. Technol.* **2015**, *20*, 583–593. [CrossRef]
2. Krishna, T.L.S.R.; Ragunathan, T.; Battula, S.K. Efficient algorithms for improving the performance of read operations in distributed file system. In Proceedings of the Symposium on High Performance Computing, Alexandria, WV, USA, 2–15 April 2015; pp. 143–149.
3. Krishna, T.L.S.R.; Ragunathan, T. Performance evaluation of speculative semantics-based algorithm for read operations in distributed file system. *Int. J. Commun. Netw. Distrib. Syst.* **2019**, *22*, 275–293.
4. Gavvala, S.K.; Jatoth, C.; Gangadharan, G.; Buyya, R. QoS-aware cloud service composition using eagle strategy. *Future Gener. Comput. Syst.* **2019**, *90*, 273–290. [CrossRef]
5. Praveen, S.P.; Rao, K.T.; Janakiramaiah, B. Effective allocation of resources and task scheduling in cloud environment using social group optimization. *Arab. J. Sci. Eng.* **2018**, *43*, 4265–4272. [CrossRef]
6. Lavanya, K.; Reddy, L.; Reddy, B.E. Distributed based serial regression multiple imputation for high dimensional multivariate data in multicore environment of cloud. *Int. J. Ambient. Comput. Intell. (IJACI)* **2019**, *10*, 63–79. [CrossRef]
7. Ujjwal, K.; Garg, S.; Hilton, J.; Aryal, J.; Forbes-Smith, N. Cloud Computing in natural hazard modeling systems: Current research trends and future directions. *Int. J. Disaster Risk Reduct.* **2019**, *38*, 101188.
8. Ujjwal, K.; Garg, S.; Hilton, J. An efficient framework for ensemble of natural disaster simulations as a service. *Geosci. Front.* **2020**, *11*, 1859–1873.
9. Liu, H.; Orban, D. Gridbatch: Cloud computing for large-scale data-intensive batch applications. In Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID), Lyon, France, 19–22 May 2008; pp. 295–305.
10. Chen, Y.; Deng, S.; Ma, H.; Yin, J. Deploying data-intensive applications with multiple services components on edge. *Mob. Netw. Appl.* **2020**, 426–441. [CrossRef]
11. Baker, T.; Aldawsari, B.; Asim, M.; Tawfik, H.; Maamar, Z.; Buyya, R. Cloud-SEnergy: A bin-packing based multi-cloud service broker for energy efficient composition and execution of data-intensive applications. *Sustain. Comput. Informatics Syst.* **2018**, *19*, 242–252. [CrossRef]
12. Ujjwal, K.; Garg, S.; Hilton, J.; Aryal, J. A cloud-based framework for sensitivity analysis of natural hazard models. *Environ. Model. Softw.* **2020**, *134*, 104800.
13. Dey, N.S.; Gunasekhar, T. A comprehensive survey of load balancing strategies using hadoop queue scheduling and virtual machine migration. *IEEE Access* **2019**, *7*, 92259–92284. [CrossRef]
14. Chen, C.P.; Zhang, C.Y. Data-intensive applications, challenges, techniques and technologies: A survey on Big Data. *Inf. Sci.* **2014**, *275*, 314–347. [CrossRef]
15. Dean, J.; Ghemawat, S. MapReduce: Simplified data processing on large clusters. *Commun. ACM* **2008**, *51*, 107–113. [CrossRef]
16. Condie, T.; Conway, N.; Alvaro, P.; Hellerstein, J.M.; Elmeleegy, K.; Sears, R. MapReduce Online. *Nsdi*, **2010**, *10*, 20. Available online: https://www.usenix.org/legacy/events/nsdi10/tech/full_papers/condie.pdf (accessed on 13 January 2021).
17. Venkateswara Rao, P.; Hussain, M. Mashup service implementation on multi-cloud environment using map reduction approach. *J. Adv. Res. Dyn. Control. Syst.* **2017**, *9*, 758–767.
18. Venkateswara rao, P.; Hussain, M. A novel filtered based grid partitioning multiple reducers skyline computation using hadoop framework. *Int. J. Eng. Technol. (UAE)* **2018**, *7*, 686–690. [CrossRef]
19. Venkateswara Rao, P.; Ali Hussain, M. An efficient pre and post processing skyline computational framework using mapreduce. *Int. J. Recent Technol. Eng.* **2019**, *8*, 5954–5959.
20. Lee, Y.; Lee, Y. Toward Scalable Internet Traffic Measurement and Analysis with Hadoop. *SIGCOMM Comput. Commun. Rev.* **2012**, *43*, 5–13. [CrossRef]
21. Ghemawat, S.; Gobioff, H.; Leung, S.T. The Google file system. In Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, Bolton Landing, NY USA, 19–22 October 2003; pp. 29–43.
22. Shvachko, K.; Kuang, 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), Incline Village, NV, USA, 3–5 May 2010; pp. 1–10.
23. Nicolae, B.; Antoniu, G.; Bougé, L. BlobSeer: How to enable efficient versioning for large object storage under heavy access concurrency. In Proceedings of the 2009 EDBT/ICDT Workshops, Saint-Petersburg, Russia, 22 March 2009; pp. 18–25.
24. Foundation, T.A.S. *Apache Hadoop 3.2.2-Introduction*; The Apache Software Foundation: Wakefield, MA, USA, 2021.
25. Nicolae, B.; Antoniu, G.; Bougé, L.; Moise, D.; Carpen-Amarie, A. BlobSeer: Next-generation data management for large scale infrastructures. *J. Parallel Distrib. Comput.* **2011**, *71*, 169–184. [CrossRef]
26. Nicolae, B. BlobSeer: Towards Efficient Data Storage Management for Large-Scale, Distributed Systems. Ph.D. Thesis, School of Computer Sceience, University of Rennes 1, Rennes, France, 2010.
27. Krishna, T.L.S.R.; Ragunathan, T. A novel technique for improving the performance of read operations in BlobSeer Distributed File System. In Proceedings of the 2014 Conference on IT in Business, Industry and Government (CSIBIG), Indore, India, 8–9 March 2014; pp. 1–7.
28. Krishna, T.L.S.R.; Ragunathan, T.; Battula, S.K. Improving the performance of read operations in distributed file system. In Proceedings of the 2014 International Conference on Computational Intelligence and Communication Networks, Bhopla, India, 27–29 May 2014; pp. 1126–1130.

29. Nicolae, B.; Cappello, F. BlobCR: Virtual disk based checkpoint-restart for HPC applications on IaaS clouds. *J. Parallel Distrib. Comput.* **2013**, *73*, 698–711. [CrossRef]
30. Coti, C.; Herault, T.; Lemarinier, P.; Pilard, L.; Rezmerita, A.; Rodriguezb, E.; Cappello, F. Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant MPI. In Proceedings of the SC'06 2006 ACM/IEEE Conference on Supercomputing, Tampa, FL, USA, 11–17 November 2006; p. 18.
31. Elnozahy, E.N.; Alvisi, L.; Wang, Y.M.; Johnson, D.B. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv. (CSUR)* **2002**, *34*, 375–408. [CrossRef]
32. Sankaran, S.; Squyres, J.M.; Barrett, B.; Lumsdaine, A.; Duell, J.; Hargrove, P.; Roman, E. Parallel checkpoint/restart for MPI applications. *Int. J. High Perform. Comput. Appl.* **2005**, *1*, 479–493. [CrossRef]