applied sciences

MDPI

Article

CRBF: Cross-Referencing Bloom-Filter-Based Data Integrity Verification Framework for Object-Based Big Data Transfer Systems

Preethika Kasu [1], Prince Hamandawana [2] and Tae-Sun Chung [1,*]

[1] Department of Artificial Intelligence, Ajou University, Suwon 16499, Republic of Korea; kasu@ajou.ac.kr
[2] Department of Software, Ajou University, Suwon 16499, Republic of Korea; phamandawana@ajou.ac.kr
* Correspondence: tschung@ajou.ac.kr

Abstract: Various components are involved in the end-to-end path of data transfer. Protecting data integrity from failures in these intermediate components is a key feature of big data transfer tools. Although most of these components provide some degree of data integrity, they are either too expensive or inefficient in recovering corrupted data. This problem highlights the need for application-level end-to-end integrity verification during data transfer. However, the computational, memory, and storage overhead of big data transfer tools can be a significant bottleneck for ensuring data integrity due to the large size of the data. This paper proposes a novel framework for data integrity verification in big data transfer systems using a cross-referencing Bloom filter. This framework has three advantages over state-of-the-art data integrity techniques: lower computation and memory overhead and zero false-positive errors for a limited number of elements. This study evaluates the computation, memory, recovery time, and false-positive overhead for the proposed framework and compares them with state-of-the-art solutions. The evaluation results indicate that the proposed framework is efficient in detecting and recovering from integrity errors while eliminating false positives in the Bloom filter data structure. In addition, we observe negligible computation, memory, and recovery overheads for all workloads.

Keywords: data integrity; Bloom filters; probabilistic structures; false-positive errors; distributed systems; high-performance computing

Citation: Kasu, P.; Hamandawana, P.; Chung, T.-S. CRBF: Cross-Referencing Bloom-Filter-Based Data Integrity Verification Framework for Object-Based Big Data Transfer System. *Appl. Sci.* **2023**, *13*, 7830. https://doi.org/10.3390/app13137830

Academic Editors: Konstantinos E. Psannis and Christos L. Stergiou

Received: 3 June 2023
Revised: 19 June 2023
Accepted: 25 June 2023
Published: 3 July 2023

1. Introduction

Large-scale scientific simulations and experiments [1–3] generate petabytes of data analyzed by scientists worldwide. Although locating the analysis resources close to the data source may appear logical and efficient, this is not always possible. Therefore, distributed solutions are far more prevalent at this scale, and distributed architectures facilitate these collaborations. However, efficient data transfer tools are needed to move vast volumes of data across geographically distributed data centers to support such an ecosystem.

Data transfer involves a number of components, including source and destination host machines, storage systems, networks, and file systems. Fine-tuning individual components to improve performance and reduce latencies between components is vital for achieving higher data transfer rates. However, a mismatch in network and storage impedance [4] can impact the performance of end-to-end data transfer. Therefore, to improve scalability, performance, and reduce the storage and network impedance mismatch, data centers often use parallel file systems (PFSs). PFSs use distinct servers to handle metadata and I/O operations [5,6].

In a PFS (parallel file system), users share resources, which can lead to competition for those resources. This can cause a significant difference between the estimated and actual I/O performance [7]. Therefore, resources must be prevented from being congested

during I/O operations to achieve the expected I/O performance. To achieve this objective, researchers proposed a layout-aware object-based bulk data movement framework (layout-aware data scheduling or LADS), optimizing I/O load imbalance issues in PFSs using layout-aware I/O scheduling and object storage target (OST) congestion-aware I/O scheduling algorithms [8–10]. Object-based big data transfer systems (OBDTS) help achieve higher end-to-end data transfer rates than state-of-the-art data transfer frameworks owing to these enhanced scheduling algorithms.

Traditional data transfer frameworks such as GridFTP [11,12] and BBCP [13] ignore the physical distribution of files within the PFS and transfer file data sequentially. Conversely, the object-based bulk data movement framework views files from a physical perspective rather than a logical one. As a result, object based data transfer tools transfer workload as objects rather than files. Owing to the object-based characteristic of the data transfer and parallel processing, objects belonging to the same file can be transferred in a non-sequential order.

Data corruption during a data transfer is a major challenge for big data transfer frameworks. Corruption is possible during the data transfer (while in "transit") or when the data are stored on the disk (at "rest"). In transit, data corruption can happen due to hardware, network, storage, memory, and software failures. Moreover, when files are being transferred between the network and the disk, or vice versa, faulty memory can cause file corruption errors [14]. Although various in-transit (e.g., the checksum calculation in TCP) [15] data integrity mechanisms can help to prevent data corruption, explicit end-to-end data integrity verification is essential for protecting data from subtle errors that may go unnoticed [16]. Therefore, big data transfer frameworks must implement end-to-end data integrity checks to ensure the accuracy and completeness of transferred data.

End-to-end integrity is a fundamental aspect of big data transfer systems that ensures the accuracy and reliability of data during the entire transfer process. State-of-the-art data transfer frameworks, such as GridFTP [11,12] and BBCP [13], transfer the data in a sequential manner. Therefore, simple checksum aggregation-based data integrity verification schemes can be used to verify data integrity in these frameworks. Conversely, the object-based transfer frameworks transfers the workload as objects. Due to parallel processing and object-oriented nature of data transfer, objects are transferred without any specific order. As a result, object signatures necessary to verify data integrity, are also computed in a non-sequential manner. Hence, complex sorting algorithms are necessary to generate file-level signatures, which has a significant impact on the total computational and memory requirements [17]. Table 1 summarizes the key differences between sequential and object based big data transfer systems.

Table 1. Differences between sequential and object-based data transfer.

Feature	Sequential Transfer	Object-Based Transfer
Data transfer order	Sequential	Out of order
Data integrity verification	Simple checksum aggregation	Complex sorting algorithms
Memory requirements	Low	High
Computational requirements	Low	High

To address the aforementioned challenges with object-based big data transfer systems (OBDTS), researchers proposed a TPBF (two-phase Bloom filter)-based [17] framework for integrity verification. TPBF leverages the space efficiency and insertion order independence characteristics of the Bloom filters to facilitate efficient data integrity verification. Although TPBF successfully reduces the computational, storage, and memory overhead typically associated with data integrity verification, it is still susceptible to false-positive errors. These errors can lead to the possibility of incorrectly assuming that specific files or objects have been successfully transferred to the destination. This can result in data corruption, which makes the data unsuitable for subsequent processing. To overcome this challenge, this paper proposes an alternative framework called the cross-referencing Bloom filter (CRBF)-

based data integrity verification framework. The primary objective of this framework is to effectively handle the data integrity requirements of OBDTS by eliminating false-positive errors. The following are the key findings and/or contributions of this work:

- The proposed framework employs object-, file-, and dataset-level integrity verification, minimizing the data to be retransmitted due to integrity errors.
- We analyzed the memory and false-positive overhead of the proposed framework. We concluded that the proposed framework is effective in eliminating false-positive errors while maintaining a similar memory footprint to the TPBF-based integrity framework.
- In order to evaluate the efficacy of the proposed framework in detecting and recovering from integrity errors, we simulated integrity errors at the object, file, and dataset levels. The experimental results showed that the proposed framework is very accurate at detecting and re-transmitting erroneous objects, files, and datasets.
- We simulated faults at different points in the data transmission process to analyze the recovery overhead and false-positive rate of the proposed framework. Faults were simulated after transmitting 20%, 40%, 60%, and 80% of the total data. The experimental findings demonstrated negligible recovery time and zero false positives for all workloads at all fault points.

The structure of the remaining sections in this paper is as follows: In Section 2, background information on data integrity verification and the motivation for this research is provided. Section 3 explores related work pertaining to data integrity verification. Section 4 introduces the Data- and Layout-Aware Bloom filter (DLBF). The design and implementation details of the proposed end-to-end data integrity verification framework are presented in Section 5. The evaluation results of the framework are discussed in Section 6, and Section 7 concludes the work.

2. Background and Motivation

2.1. Background

2.1.1. Object-Based Big Data Transfer Systems (OBDTS) and Data Integrity

Figure 1 depicts the file distribution and data transfer mechanism for the state-of-the-art and OBDTS. As shown in Figure 1, we considered two files, $File_a$ and $File_b$, distributed across four OSTs, OST_1 to OST_4, and two I/O threads, T_1 and T_2, for transferring the objects of these files. On initiating the data transfer, as illustrated in Figure 1b, both T_1 and T_2 threads contend for the OST_1 resource due to the sequential transfer mechanism. As a result, one of T_1 or T_2 must yield to the other thread until it finishes its task. Similarly, later in the data transfer, the T_1 and T_2 threads again contend for the resource OST_3. This kind of resource competition problem results in lower data transfer rates. In contrast, as illustrated in Figure 1c, on initiating the data transfer, OBDTS identify and avoid temporarily congested storage servers by exploiting storage architecture and employing layout- and congestion-aware scheduling algorithms. Thus, as depicted in Figure 1c, the T_1 thread is assigned to OST_3, and the T_2 thread is assigned to OST_4 to avoid OST_1 congestion. As a result, Figure 1c reveals that objects belonging to a single logical file are transferred in a non-sequential manner. Object-based big data transfer systems (OBDTS) can achieve higher data throughput than traditional big data transfer frameworks due to their enhanced scheduling algorithms [8–10].

Transfer efficiency and reliability are two primary requirements of big data transfer frameworks. As described, OBDTS improve data transfer rates by exploiting the storage layout architecture. However, this results in out of order object transmission of the same logical file. Therefore, simple checksum-based data integrity mechanisms result in space and computational overhead. In this study, we propose a memory-efficient Bloom-filter-based data integrity mechanism. This mechanism enables the verification of data integrity at multiple levels, including object, file, and dataset.

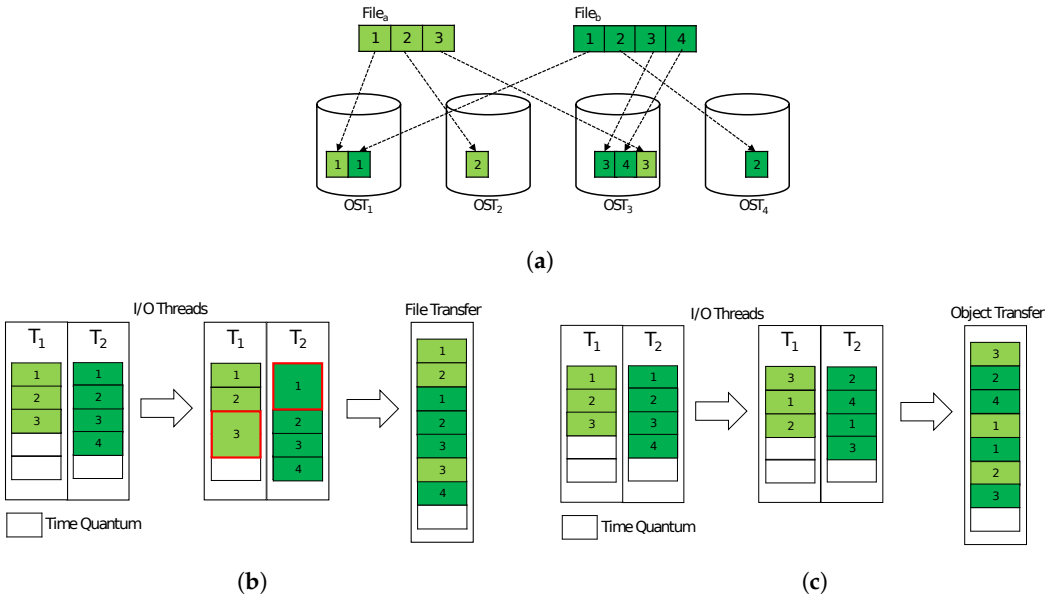


Figure 1. File distribution and transfer mechanisms. (**a**) File distribution. (**b**) State-of-the-art big data transfer mechanism. (**c**) Object-based big data transfer mechanism.

2.1.2. Bloom Filter Data Structure

A Bloom filter (B) is a space-efficient probabilistic data structure [18–20] that is used to test whether an element is a member of a set. Bloom filters utilize k distinct hash functions ($h_1, \ldots, h_k$) to map elements in the set ($S = \{s_1, \ldots, s_n\}$) to a bit array of m bits. When an element is inserted into the filter, it is hashed by each of the hash functions, and the corresponding bits in the array are set to 1. To check if an element is a member of the set, the element is hashed by the same hash functions, and the corresponding bits are checked. If any of the bits are not set, then the element is definitely not in the set. If all bits are set, it is probable that the element is in the set, but there is a chance of false positives.

Figure 2 depicts the insert and query operations of the standard Bloom filter.

- **Insert:** To insert each object s_i in the set S,
  - Compute the hash values $h_1(s_i), \ldots, h_k(s_i)$ for each object s_i. The hash functions can be any hash functions that are collision-resistant.
  - Set the corresponding bits in the Bloom filter B such that $B[h_1(s_i)] = B[h_2(s_i)] = \ldots = B[h_k(s_i)] = 1$.

  Figure 2a depicts the insert operation of object$_A$, object$_C$, and object$_F$.
- **Query:** To determine whether an object, s_i, belongs to set S,
  - Compute the hash values $h_1(s_i), \ldots, h_k(s_i)$ for each object s_i.
  - If all corresponding bits in the Bloom filter B are set, then membership query is successful; otherwise, the element is not in the set. However, if collisions occur during the hashing process and cause $h_1(s_i), \ldots, h_k(s_i)$ in the bit vector B to be set to 1, this leads to false positives.

  Figure 2b depicts the query operation of object$_A$ and object$_D$. As object$_A$ was originally inserted into the Bloom filter, its membership query results are positive. In contrast, object$_D$ was not inserted into the Bloom filter. However, the membership query returns a positive result due to the hash collisions. This approach results in false-positive errors.

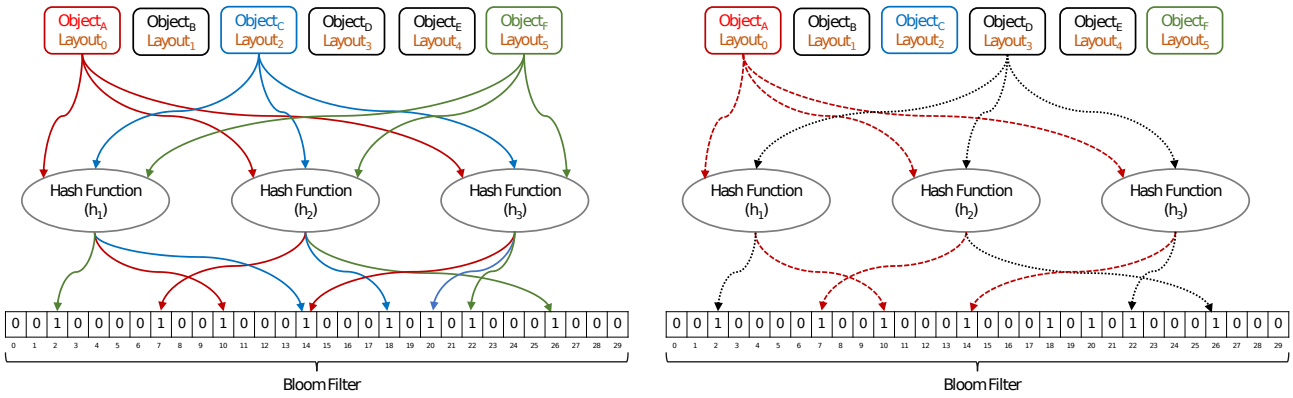


Figure 2. Standard Bloom filter. (**a**) Bloom filter insert operation. (**b**) Bloom filter query operation.

As described above, if m is the size of the Bloom filter and k is the number of hash functions, then the probability that a certain bit is still zero after inserting one element and after insertng n elements is represented as follows [21]:

$$p = (1 - \frac{1}{m})^k \tag{1}$$

$$p = (1 - \frac{1}{m})^{kn} \approx (e^{-kn/m}). \tag{2}$$

Therefore, the probability of false positive errors can be expressed as

$$\epsilon = (1 - (1 - \frac{1}{m})^{kn})^k \approx (1 - e^{-kn/m})^k. \tag{3}$$

From Equation (3), we observe that a higher number of hash functions results in fewer false-positive errors. However, increasing the number of hash functions directly affects the computational overhead. Therefore, for a given m and n, the required number of hash functions can be represented as

$$k = \frac{m}{n} ln2. \tag{4}$$

By substituting Equation (4) into Equation (3), we can express the false-positive error probability as

$$\epsilon = (1 - e^{(\frac{m}{n} ln2)\frac{n}{m}})^{\frac{m}{n} ln2}. \tag{5}$$

By simplifying Equation (5), we can represent it as

$$ln\epsilon = -\frac{m}{n}(ln2)^2 \tag{6}$$

Thus, required number of bits is

$$m = -\frac{nln\epsilon}{ln2^2} \tag{7}$$

2.2. Motivation

Object based big data transfer systems (OBDTS) leverage the layout architecture and employ layout- and congestion-aware object scheduling to improve data transfer rates. However, these enhanced scheduling algorithms transfer file objects and dataset files out of order. This out-of-order nature of the data transfer necessitates higher memory and complex sorting algorithms to generate consistent file, and dataset level signatures for data integrity verification. Consequently, one of the main objectives of end-to-end data integrity verification frameworks is to minimize the computational and memory overheads

associated with data integrity verification, thereby optimizing the overall data transfer performance. The TPBF method of data integrity [17] addressed these issues by exploiting Bloom filter's memory efficiency and insertion-order-independent features.

The Bloom filter is a simple, space-efficient, and insertion-order-independent probabilistic data structure that can be used for both insertion and lookup operations in constant time [18,21]. This data structure is based on hashing and is similar to a hash table that uses hash functions to map an element in the set to a bucket. However, unlike a hash table, a Bloom filter does not store the element in that bucket; instead, it simply marks the bucket as filled. Hence, multiple elements in the set may map to the same filled bucket, resulting in false-positive errors. In this work, we aim to avoid the false-positive errors by exploiting the lower false-positive characteristics of the cross-referencing Bloom filter data structure [22].

3. Related Work

3.1. Performance Optimization of the Bloom Filter

Space efficiency, insertion order independence, and zero false-negative lookups are salient features of the Bloom filter data structure. Exploiting the capabilities of the Bloom filter makes it possible to design an efficient integrity framework for object based bulk data movement systems. However, as Bloom filters are probabilistic data structures, they can produce false-positive results. Thus, the Bloom filter marks an element as a member, although it is not a member of the set. The false-positive rate can be reduced by increasing the Bloom filter size and the total hash functions used for generating the Bloom filter. However, increasing the Bloom filter size is not an optimal solution considering the memory and computational efficiency. Therefore, many researchers have proposed solutions to optimize the Bloom filter performance [23–26] while maintaining space and computational efficiency.

Lim et al. [22] proposed a method to enhance Bloom filter performance by reducing the false positives while still maintaining processing simplicity and storage efficiency. This method employs cross-checking Bloom filters to query and cross-check the results when a primary Bloom filter returns a positive value for a membership query. If all cross-checking Bloom filters return negative results, the primary Bloom filter's positive value can be classified as a false positive. The primary Bloom filter may not be large enough to reduce the false-positives, because most false positives are identified by cross-checking with other Bloom filters. This method programs and queries the cross-checking Bloom filters by grouping the input set based on their characteristics. Cross-checking Bloom filters effectively reduce false positives while minimizing the memory footprint. However, grouping the input in large-scale data transfer systems based on their characteristics is complex and is not always possible.

Tabataba et al. [27] proposed a dual-Bloom-filter structure to reduce the false positive rate. This approach uses two Bloom filters that are connected in series. Both Bloom filters are programmed with the same set of input elements, but they use different hash functions. When querying for membership, if the first Bloom filter returns a positive result, the second Bloom filter is queried. If the second Bloom filter returns a negative result, then the positive result of the first Bloom filter is considered a false positive. The use of different hash functions results in memory overhead, so the same hash functions are used to generate both Bloom filters. However, to ensure the randomness of the hash functions, a second Bloom filter is generated by complementing the input string. That is, all 0 bits of the input string are converted to 1 and vice versa. This method effectively reduces false positives, but it uses twice as much memory as a standard Bloom filter.

Kasu et al. [28] proposed a DLBF data structure to avoid false-positive errors with object based big data transfer systems. To avoid false positives, object layout information (n-bits) is appended to the standard Bloom filter as an additional information about the object. On successful object transfer, the object layout bit and the k hashed positions are set to 1. Upon a fault, while retrieving the successful objects, k hash positions and layout

bits are used for the membership query. Although this method effectively avoids false-positive matches, maintaining an individual Bloom filter for every file in the dataset leads to memory and storage overhead. A TPBF [17] data structure was adopted to address this. Although the TPBF effectively reduces the memory and storage footprint with OBDTS, this data structure is prone to false-positive errors.

3.2. Data Integrity

The primary objective of big data transfer frameworks is to ensure the accuracy of transferred data. To accomplish this, extensive research has been conducted on the design and optimization of data integrity verification systems in different contexts, including storage systems [29,30], cloud-based storage [31,32], file systems [33,34], and data transfer systems [35–38].

GridFTP [11,12] and BBCP [13] are the two most popular and widely used data transfer tools in the big data community. These tools are file-based and transfer data sequentially. Simple checksum aggregation methods are commonly used to create file-level signatures and verify the integrity of data at the file level. This is possible because data transfer is a sequential process. However, this sequential, file-based method of verifying data integrity is not suitable for large datasets. If a dataset contains large files, the checksum computation may fail due to the size of the data, or it may take an extremely long time to complete. For this reason, a parallel checksum approach is necessary and preferable. Many researchers have proposed methods for transferring multiple files in parallel by exploiting pipelining and parallelism [35–37,39] techniques. Unlike serial and file-based data transfer tools, this work considers transferring the workload as objects rather than files. Due to the object-based transfer and parallel processing, objects of the same logical file may be transferred out of order from the source to the destination. This out-of-order nature of the data transfer demands complex sorting and checksum aggregation methods to generate file- and dataset-level signatures used for data integrity verification.

Xiong et al. [38] introduced *fSum*, a scalable parallel dataset checksumming tool designed specifically for long-term storage or archival. This method divides the workload into multiple chunks and distributes them to multiple processes, which calculate the chunk-level checksums in parallel. As a result of the parallel execution, the checksums are generated in a random order. Therefore, all the checksums must be sorted at the root process to generate a consistent dataset-level signature. However, this method is not scalable for large datasets as the root process must collect and store checksums of all processes in memory. This process requires a lot of memory and computation, which can become a bottleneck as the number of processes increases. A Bloom-filter-based data structure aggregates the chunk-level checksums to address these challenges. The insertion-order-independent nature of the Bloom filter data structure helps generate a single and consistent dataset-level signature without sorting. Therefore, this data integrity method for long-term storage or archival is more efficient in memory and computation than other approaches. However, the probabilistic nature of the Bloom filter can lead to false positives [18,20,40].

The proposed data integrity framework is built based on the properties of space efficiency and insertion order independence of the Bloom filter. The proposed CRBF-based integrity verification framework complements the TPBF data integrity method in minimizing the memory and computational overhead while ensuring zero false-positive detections for the given workloads.

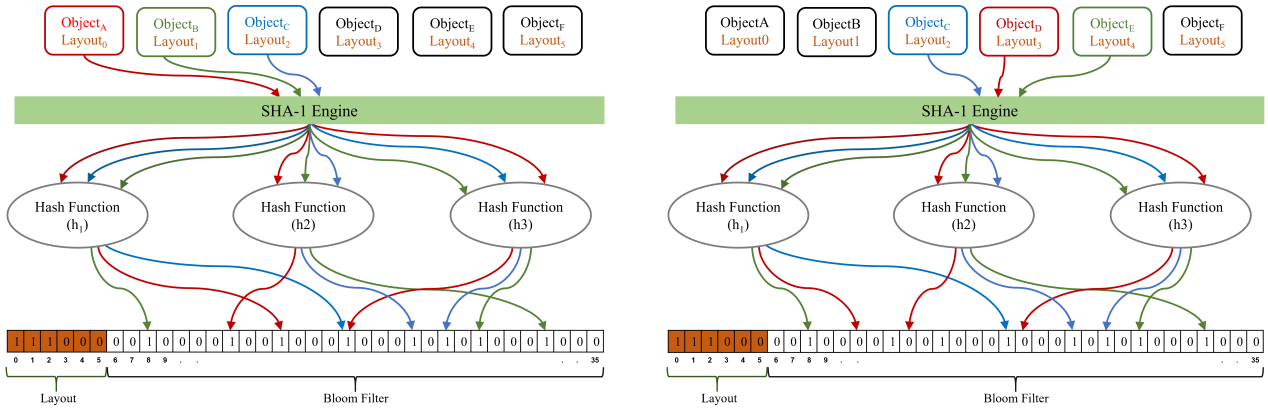
4. Data- and Layout-Aware Bloom Filter (DLBF)

The two principal design considerations for using Bloom filters to support data integrity verification in object-based big data transfer systems are space efficiency and insertion order independence. However, as Bloom filters are probabilistic data structures, membership query result in false positives. This section presents a modified version of the Bloom filter, the DLBF [28], to handle false-positive errors more efficiently than a standard Bloom filter.

As depicted in Figure 3, the DLBF filter employs the SHA-1 block hash engine to transform objects of varying sizes to a fixed size. The resulting deterministic block hash is used as input to the hash functions. Additionally, to prevent false positives in membership queries, the object layout (n-bits) is appended as additional information about the objects. Consequently, the overall size of the Bloom filter is increased to $(n + m)$ bits. Initially, all these $(n + m)$ bits are set to zero.

- **Insert Operation:** Calculate the hash values $h_1(s_i)$, ..., $h_k(s_i)$ for each object s_i and set the corresponding hashed bits, along with the object layout bit, in the Bloom filter B, such that $B[i] = B[h_1(s_i)] = B[h_2(s_i)] = ... = B[h_k(s_i)] = 1$.
- **Query Operation:** To test object membership, compute the hash values $h_1(s_i)$, ..., $h_k(s_i)$ for each object s_i and check the corresponding hashed bits along with the object layout bit in the Bloom filter B are set to 1 or not. If all of $B[i] = B[h_1(s_i)] = B[h_2(s_i)] = ... = B[h_k(s_i)] = 1$ are 1, then membership query is successful; otherwise, the element is not in the set.

Figure 3a represents the Bloom filter state after inserting object$_A$, object$_B$, and object$_C$. Figure 3b illustrates the query operation of object$_C$, object$_D$, and object$_E$. The membership query of object$_C$ is successful as hashed bit positions $B[20]$, $B[24]$, and $B[26]$ along with the layout bit $B[3]$ is set to 1. Membership query of object$_D$ fails as one or more of the hashed positions $B[11]$, $B[14]$, and $B[21]$ are not set to 1. However, the query for object$_E$ returns a false positive in the absence of layout bit, $B[4]$, as all the hashed bit positions $B[8]$, $B[28]$, and $B[32]$ are set to 1 by hash collisions. Therefore, by incorporating the object layout information, we can effectively mitigate the occurrence of false positives associated with the Bloom filter data structure.



(**a**) (**b**)

Figure 3. Data- and layout-aware Bloom filter. (**a**) DLBF insert operation. (**b**) DLBF query operation.

5. Cross-Referencing Bloom-Filter-Based Data Integrity Framework

This section discusses the design and implementation aspects of the proposed CRBF-based data integrity framework, followed by a memory overhead and false-positive error probability analysis.

5.1. System Architecture

The system architecture of the CRBF-based data integrity verification framework for object-based big data transfer systems is presented in Figure 4. This framework consists of two core components: the *file handler* and *object handler*.

The *file handler* is responsible for managing file-level activities such as scheduling the transfer of file objects and performing file level integrity verification. Conversely, the *object handler* is responsible for transferring objects and verifying the integrity of object data.

As depicted in Figure 4, the DLBF [28], dataset Bloom filter (DSBF), and metadata Bloom filter (MDBF) are the core data structures of the proposed framework.

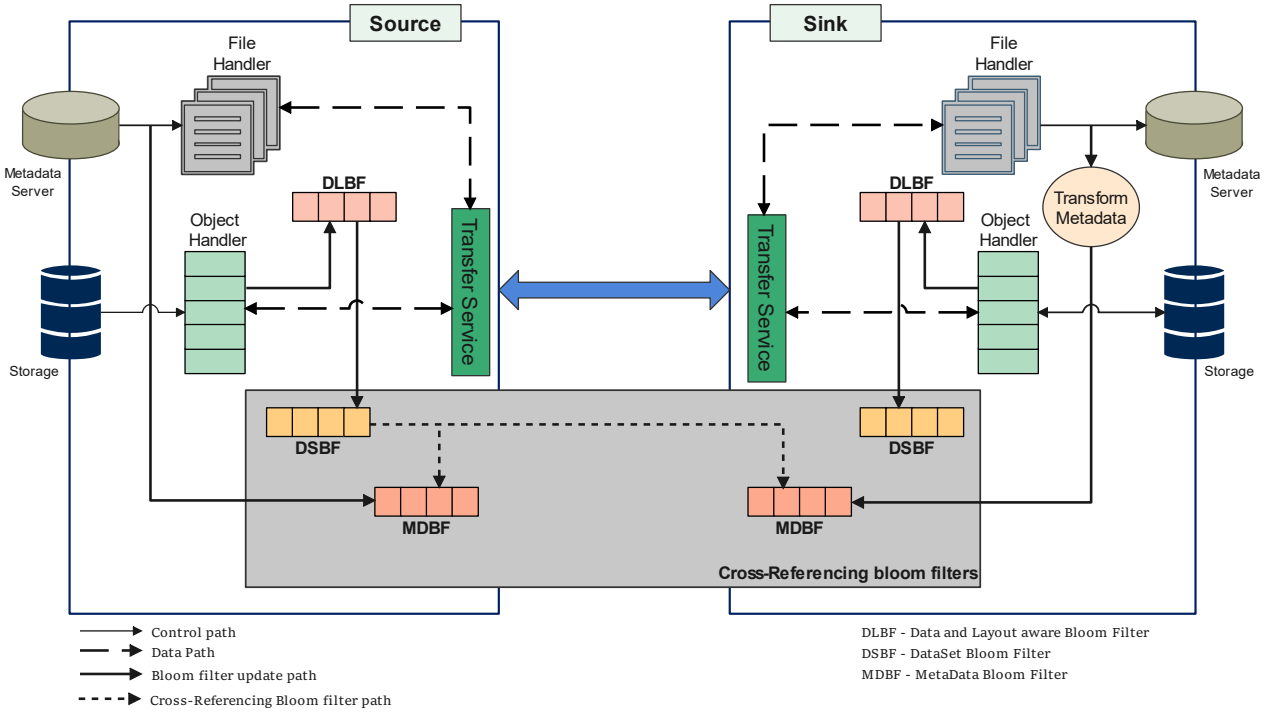


Figure 4. Cross-referencing Bloom-filter-based data integrity verification system.

- **Data- and Layout-Aware Bloom filter (DLBF):** This Bloom filter [28] data structure stores the successfully transferred object information of an active file. The number of active files and active DLBFs depends on the number of I/O threads. As described in Section 4, DLBF is a modified data structure designed to prevent false positives of the standard Bloom filter. It consists of two segments: the layout and the Bloom filter, as shown in Figure 5. After a successful object transfer and subsequent integrity verification, the object layout bit (located in the layout segment) and k hashed bit positions, computed based on the object signature, are set to 1 in the DLBF. However, if the object fails the integrity verification, the *file handler* reschedules the object for retransfer. This process will be continued until all file objects have been successfully transferred without any errors.
  During data transfer, if there is a fault and if the data transfer is resumed from a fault point, the proposed data integrity framework retrieves successfully transferred objects by querying the DLBF. Owing to the layout-aware nature of the data structure, the DLBF produces no false positives. Furthermore, dynamic changes to the objects and files can be detected using this data structure.
- **Dataset Bloom Filter (DSBF):** This Bloom filter data structure maintains information regarding successfully transferred files. Upon transferring all file objects, a file signature is computed using the DLBF. If the file signature at both ends of the data transfer is the same, file-level integrity verification succeeds, and the k hashed bit positions computed on the file signature are set to 1 in the DSBF. If the file-level integrity verification fails, then the *file handler* schedules all objects of that file for retransfer. The process of transferring files is continued until all files in the dataset have been transferred successfully and without any errors.
  If a fault occurs and the transfer resumes from the fault point, the DSBF is used to retrieve successfully transferred files. However, because Bloom filters are probabilistic in nature, there is a chance of false positives. To efficiently detect the false-positive errors of the DSBF, metadata Bloom filters (MDBF) at both ends of the data transfer are used as cross-referencing Bloom filters. If both the cross-referencing Bloom filters produce positive results, then the positive result of DSBF is considered as positive.

Conversely, if any of the cross-referencing Bloom filters produces a negative result, then the positive result of DSBF is considered as negative.

- **Metadata Bloom Filter (MDBF):** This Bloom filter stores the metadata of files that have been successfully transferred. File metadata in the following format is used at both ends of the data transfer.

[File_Name, File_Size, TotalObjects, CreateTime, ModifyTime]

Upon successful file-level integrity verification, both ends of the data transfer update MDBF with the k hashed bit positions computed on the file metadata. This Bloom filter acts as a cross-referencing Bloom filter for the DSBF to minimize false-positive errors while recovering from a fault.

Figure 5 depicts the MDBF aggregation mechanism. As indicated in the figure, on a successful file transfer, both the source and sink endpoints use file metadata to aggregate the MDBF. At the sink end, the file metadata are transformed using 1's complement and are hashed using a different set of hash functions. Thus, the resulting MDBF at both ends has an entirely different bit vector for the same input data. A lower false-positive rate than that of the standard Bloom filter can be achieved using these independent MDBFs at the source and sink endpoints. Therefore, in the proposed design, we used MDBFs as cross-referencing Bloom filters to validate the positive results of the DSBF.

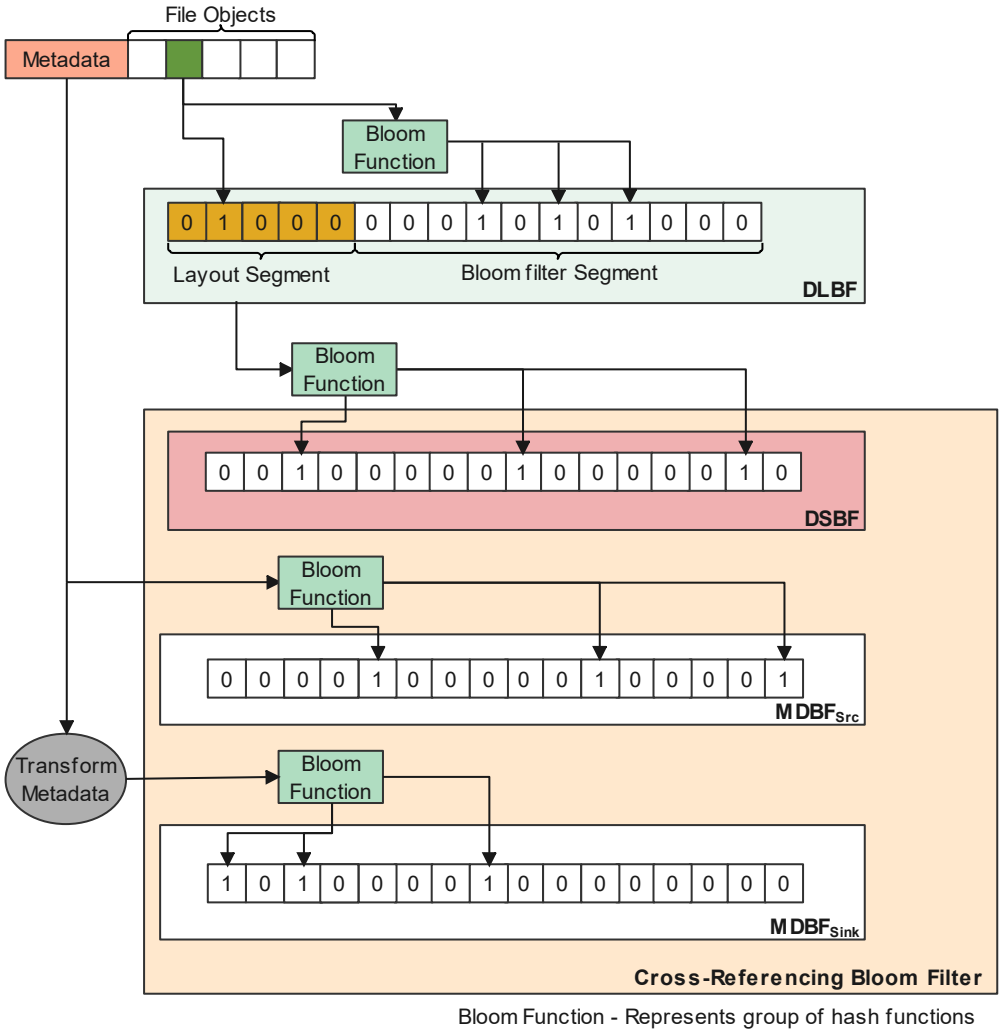


Figure 5. Cross-referencing Bloom filter data structure.

5.2. Implementation Details

The communication sequence and the messages that are exchanged between data transfer endpoints are illustrated in Figure 6 and Listing 1, respectively.

Listing 1. Communication message type.

```
enum message {
    MSG_CONNECT = 0,     //Initial connect request
    MSG_SUCCESS,         //Successful connection
    MSG_NEWFILE,         //Send new file
    MSG_FILEID,          //Create fileid at sink.
    MSG_NEWOBJ,          //Send new object
    MSG_OBJSYNC,         //Receive object sync
    MSG_FILECLOSE,       //Send file close
    MSG_FILECHECK,       //Cross-check MDBF
    MSG_CHECKRSP,        //MDBF response
    MSG_DISCONNECT       //Ready to disconnect
} msg_type_t;
```

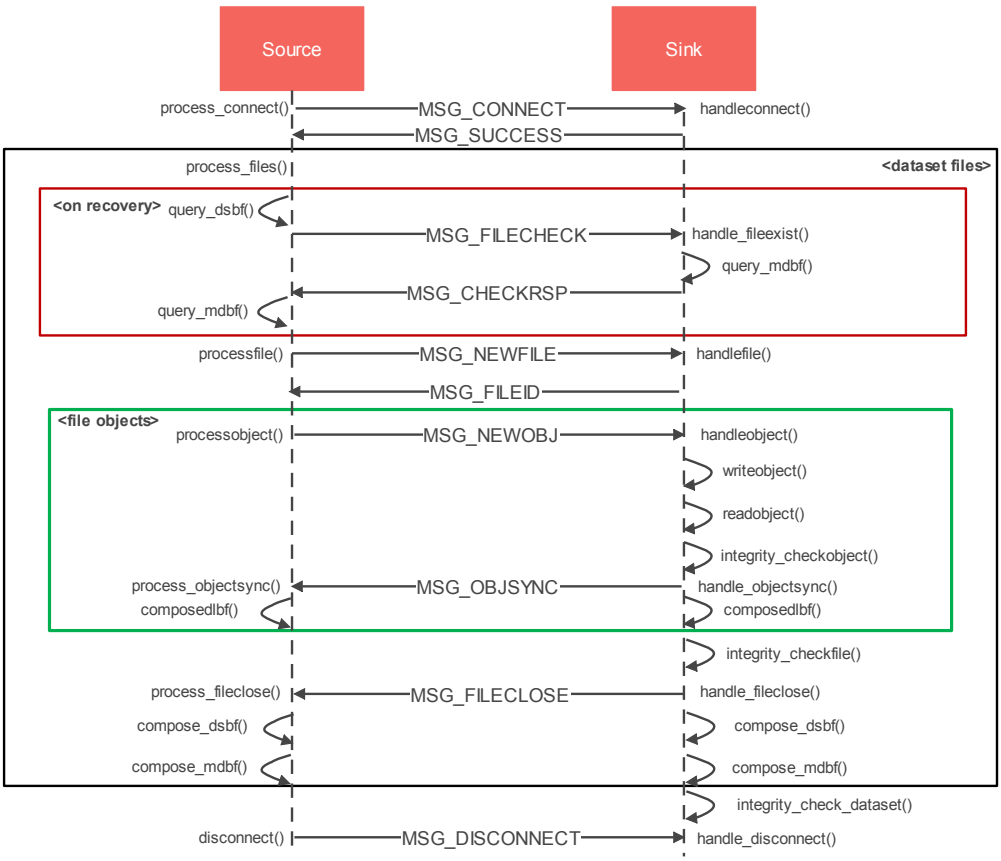


Figure 6. Communication sequence of the CRBF-based integrity framework.

When data transfer is initiated, the following steps occur:

1. The source initiates a connection to the sink by sending a MSG_CONNECT request. Upon successful connection, the sink responds with a MSG_SUCCESS message.
2. The source creates a list of files that need to be transferred.
3. If the transfer is resumed from a failed transfer, then go to Step 12.
4. For each file, the source end sends a MSG_NEWFILE request to the sink end. The request message contains file metadata information. The sink end opens the file

specified in the request message and responds with a MSG_FILEID message. The response message includes the file descriptor used in the sink end.

5. The source queries the DLBF to see if the object has already been transferred. If it has not, the source initiates the object transfer by sending a MSG_NEWOBJ request to the sink. Upon receiving the data, the sink writes the object data to the PFS. Once the object has been successfully written, the sink computes the block hash and compares it to the hash that was received in the MSG_NEWOBJ request. If the hashes match, the sink responds with a MSG_OBJSYNC message.
6. If the object integrity verification is successful, the file-based DLBF is aggregated at both the source and sink endpoints. However, if the verification fails, the source endpoint marks the object as failed and schedules it for re-transfer.
7. Steps 5 and 6 are repeated for all file objects.
8. Upon receiving all objects of a file, the sink endpoint validates the integrity of the file by comparing the computed file hash with the file hash received from the source endpoint. If the hashes match, the sink endpoint acknowledges the completion of the file transfer by sending a MSG_FILECLOSE message.
9. If the file integrity verification is successful, both the dataset Bloom filter (DSBF) and the metadata Bloom filter (MDBF) at both ends of the data transfer are aggregated with the file information. However, if the file integrity verification fails, the source endpoint schedules a re-transfer of the file.
10. For each file in the dataset, Steps 3 to 9 are repeated.
11. After transferring all files in the dataset, the source endpoint performs a dataset integrity verification. If the verification is successful, it sends a MSG_DISCONNECT message to the sink endpoint. However, if the verification fails, the source endpoint repeats Steps 2 to 11.
12. For each file, the source endpoint queries the DSBF to check whether the file is already transferred and whether the file integrity is maintained.
13. If the DSBF query result is negative, then go to Step 4.
14. If the DSBF query result is positive, the source endpoint issues a FILE_CHECK request with the file metadata. The sink endpoint queries the sink-end MDBF and responds with CHECK_RSP by including the MDBF query result.
15. If the sink-end MDBF query result is positive, the source endpoint queries the source-end MDBF. If the result is positive, the file is considered successfully transferred and skipped from the transfer. Otherwise, go to Step 4.

5.3. Memory Overhead Analysis

5.3.1. Memory Requirements of a Standard Bloom-Filter-Based Data Integrity Solution

This section analyzes the memory requirements of a standard Bloom-filter-based data integrity solution [38].

We consider the dataset as shown in Table 2.

Table 2. Dataset specification.

Files in the dataset	N
Objects per file	S
Total objects in the dataset	$(N \times S)$

Given a dataset and a false-positive probability, the total number of bits required for a data integrity solution can be calculated by substituting n in Equation (7).

$$m = -\frac{(N \times S) ln\epsilon}{ln2^2} \tag{8}$$

If we assume that the total number of files in the dataset (S) is equal to the number of objects per file (N), and we ignore any constant values, then the total bits required can be approximated as

$$m \approx (N^2) \tag{9}$$

5.3.2. Memory Requirements of CRBF-Based Data Integrity Solution

The memory requirements of the proposed data integrity framework are analyzed in this section.

Considering the same dataset as that described in Section 5.3.1, the total number of entries per Bloom filter in the cross-referencing Bloom-filter-based data structure is as shown in Table 3.

Table 3. Entries per Bloom filter in CRBF.

Total elements in DLBF	$(C \times S)$
Total elements in DSBF	N
Total elements in MDBF	N

Where C = number of active file transfers.

Therefore, the total memory required by the proposed data integrity solution is the sum of the memory overhead for the DLBF, DSBF, and MDBF, and the same can be represented as follows:

$$m = -\frac{(C \times S) ln\epsilon}{ln2^2} + -\frac{(N) ln\epsilon}{ln2^2} + -\frac{(N) ln\epsilon}{ln2^2} \tag{10}$$

Assuming that the total number of files in the dataset (S) is equal to the number of objects per file (N) and that the constant value C is much smaller than N, then the total bits required can be approximated as

$$m = -\frac{(3N) ln\epsilon}{ln2^2} \approx N \tag{11}$$

From Equation (11), we can observe that CRBF-based integrity framework has a linear memory requirement. This means that the amount of memory needed increases linearly with the number of elements in the dataset (i.e., $O(N)$). Conversely, Equation (9) suggests that the standard Bloom-filter-based integrity solution has a quadratic memory requirement. This means that memory usage grows quadratically with the number of elements in the dataset (i.e., $O(N^2)$). Therefore, the proposed data integrity framework efficiently reduces the memory and storage footprint requirements.

5.4. False-Positive Rate Analysis

5.4.1. False-Positive Rate Analysis of Data Integrity Solution Based on the Standard Bloom Filter

Simplifying Equation (5), the false positives of the standard Bloom filter-based integrity solution can be represented as

$$\epsilon = (\frac{1}{e^{(ln2)^2}})^{\frac{m}{n}} \tag{12}$$

$$\epsilon \approx (0.61850)^{\frac{m}{n}} \tag{13}$$

5.4.2. False-Positive Rate Analysis of Data Integrity Solution Based on the CRBF

This section analyzes the false-positive rate of the data integrity solution based on the CRBF. As described in Section 5.1, the DLBF and DSBF store file and dataset information, respectively. False-positives of the proposed framework only apply when recovering from failed transfers. While DLBF is used to recover already transferred objects of an active file, DSBF is used to recover transferred files. Due to the data and layout aware characteristic of

the DLBF, it is possible to avoid false positives [28]. However, DSBF results in false-positive errors. To efficiently handle the false positives of DSBF, MDBFs at both ends of the data transfer are used as cross-referencing Bloom filters.

As displayed in Figure 5, file metadata are used to generate the source-end MDBF, and 1's complemented the file metadata to generate the sink-end MDBF. In addition, a different hash functions are used for generating these Bloom filters. Hence, the sink-end bit vector is different from the source-end MDBF. Therefore, the resulting false-positive rate by cross-referencing these Bloom filters is represented as

$$ln\epsilon = -\frac{m}{n}(ln2)^2 + -\frac{m}{n}(ln2)^2 \tag{14}$$

By simplifying Equation (14),

$$\epsilon = (\frac{1}{e^{(ln2)^2}})^{\frac{m}{n}} \times (\frac{1}{e^{(ln2)^2}})^{\frac{m}{n}} \tag{15}$$

$$\epsilon \approx (0.38254)^{\frac{m}{n}} \tag{16}$$

Based on Equations (13) and (16), we can conclude that the proposed integrity framework has a lower false-positive rate than the standard Bloom-filter-based integrity solution.

6. Evaluation

The testbed and workload specifications are described in this section. We also present the results of our experiments and discuss their implications. We evaluate the impact of CRBF- based integrity verification system on overall performance of data transfer. We then assess the effectiveness of the CRBF-based data integrity verification system by analyzing factors such as memory overhead, recovery overhead, and false positive error probability. All experiments employed a large, fixed dynamic random-access memory (DRAM) as random-access memory (RMA) buffers at both the source and sink ends, with a maximum utilization of 256 MB at each end. The results were obtained through multiple iterations and presented as average bar graphs. Additionally, 99% confidence intervals are displayed in the error bar wherever needed.

6.1. Test Environment

6.1.1. Testbed Specification

We evaluated our framework on a testbed with two nodes: a source and a sink. The nodes were connected by an InfiniBand interface. Each node was equipped with Intel Xeon E5-2650 v4 CPU with 32 cores and 16 GB of DRAM. The operating system used was Linux kernel v. 3.10.0-1160. Both nodes used the Lustre file system v. 2.13.0 [41] with a single object storage server (OSS) and four object storage targets (OSTs) mounted over a 1 TB drive. The Lustre file system at both endpoints was configured with a stripe count of 4 and a stripe size of 1 MB [42].

6.1.2. Workload Specification

We analyzed the file size distribution using the Lustre file system data from Atlas 1 and 2 [43], which were hosted by the OLCF (Oak Ridge Leadership Computing Facility) [3]. The relationship between the file size and the number of files is depicted in Figure 7. The distribution data revealed that 91.55% of the files are under 4 MB, and 84.17% are under 1 MB. Only 10% of the files are larger than 4 MB despite occupying a significant portion of the file system's capacity. Thus, we used three datasets with varying file sizes to test the framework, as listed in Table 4. We populated the source file system with $D1$, $D2$, and $D3$ workloads by striping the data across all source-end OSTs. As described in Section 6.1.1, a stripe count of 4 and a stripe size of 1 MB are used to stripe the data across the OSTs.

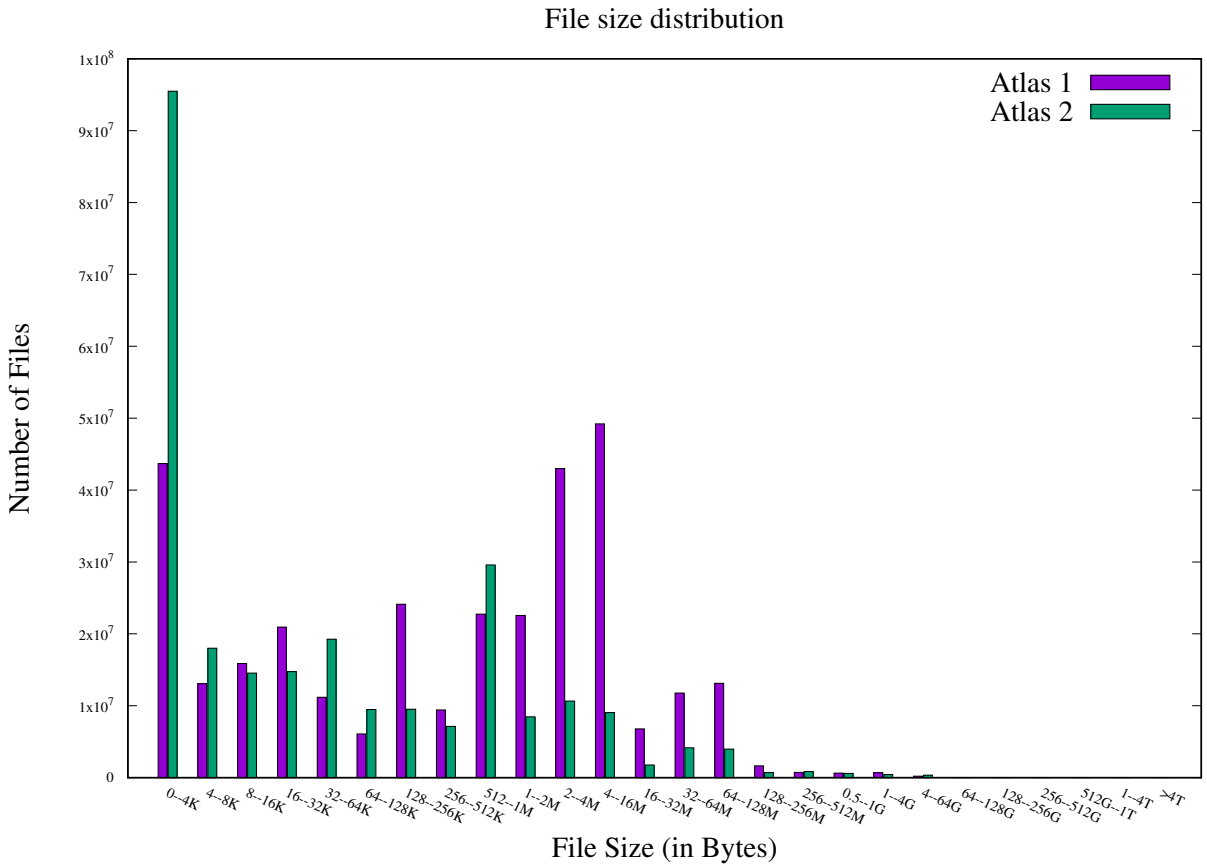


Figure 7. File size distribution.

Table 4. Workload specification.

Dataset	No. of Files	File Size Range
D1 (Big workload)	100	1 GB∼2 GB
D2 (Small workload)	100,000	1 KB∼1 MB
D3 (Mixed workload)	20,050 *	1 KB∼2 GB

* D3 workload contains files of different sizes.

6.2. Performance Evaluation

6.2.1. Data Transfer Rate

The primary objective of this work is to reduce the impact of the proposed framework on the overall speed of data transfers. To address this objective, we evaluated and compared the total data transfer time and throughput of the proposed CRBF-based integrity framework with that of the baseline (without integrity), TPBF, and GridFTP data transfer frameworks. To evaluate the framework fairly, we used four I/O threads for the CRBF and TPBF frameworks and four parallel TCP streams for GridFTP.

The data transfer time for all workloads (see Section 6.1.2) is presented in Figure 8a, indicating that the proposed framework has a minimal impact on the total data transfer time compared to the baseline, TPBF, and GridFTP. Although computing the Bloom filter in the proposed data integrity framework is computationally expensive, we minimize this overhead using hash optimization [44,45] and pipelining techniques. The pipeline is designed to perform read, write, hashing, Bloom-filter-generating, and data transfer operations as efficiently as possible. Each function overlaps with the activities of another block to maximize efficiency. Additionally, Figure 8b illustrates the data transfer rate comparison for D3 workloads with varying numbers of I/O threads, revealing that the GridFTP delivers higher data transfer rates than the proposed framework when using a

single I/O thread. However, as the number of I/O threads increases, the TPBF and the suggested framework outperform the GridFTP regarding the data transfer rate. Furthermore, Figure 8b demonstrates that the proposed CRBF-based data integrity framework imposes a minimal overhead, about 5%, on the overall data transfer rate compared to the baseline. In addition, this figure presents negligible overhead compared with the TPBF data integrity method. Consequently, we conclude that the proposed CRBF-based data integrity framework has a negligible influence on data transfer speed.

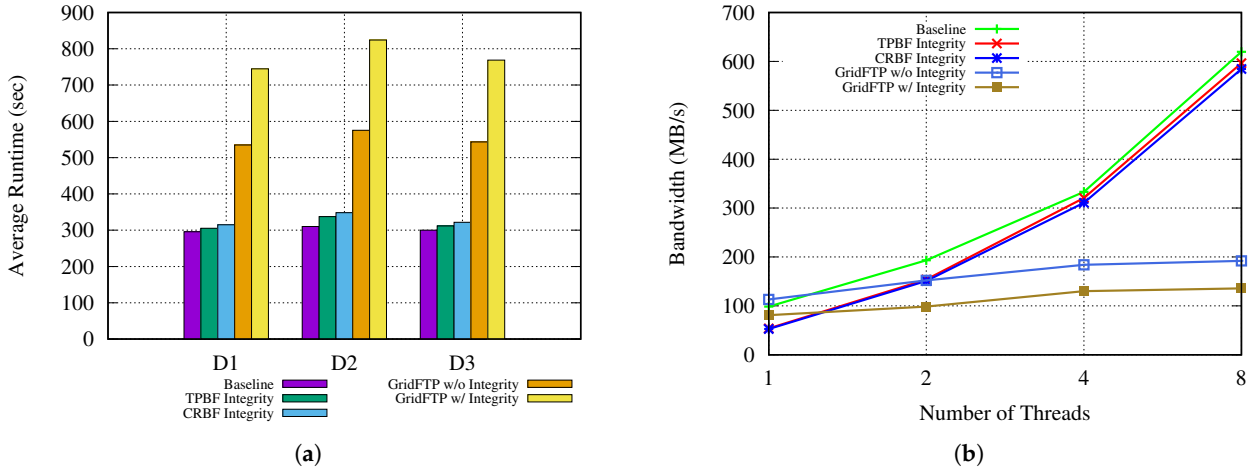


Figure 8. Data transfer performance analysis. (**a**) Data transfer time comparison. (**b**) Data transfer rate comparison for D3 workloads.

6.2.2. CPU Load Analysis

The total CPU load during the data transfer and integrity verification is another critical design aspect of the proposed framework. Figure 9 depicts the average CPU load of the proposed CRBF data integrity framework compared to the TPBF and baseline (without integrity). A bar graph represents the average CPU load, and the error bars denote the 99% confidence intervals. These graphs illustrate that the proposed CRBF-based data integrity demands 10% to 15% higher CPU resources than the TPBF-based data integrity. However, the experimental results demonstrate that this overhead does not affect the overall data transfer rate.

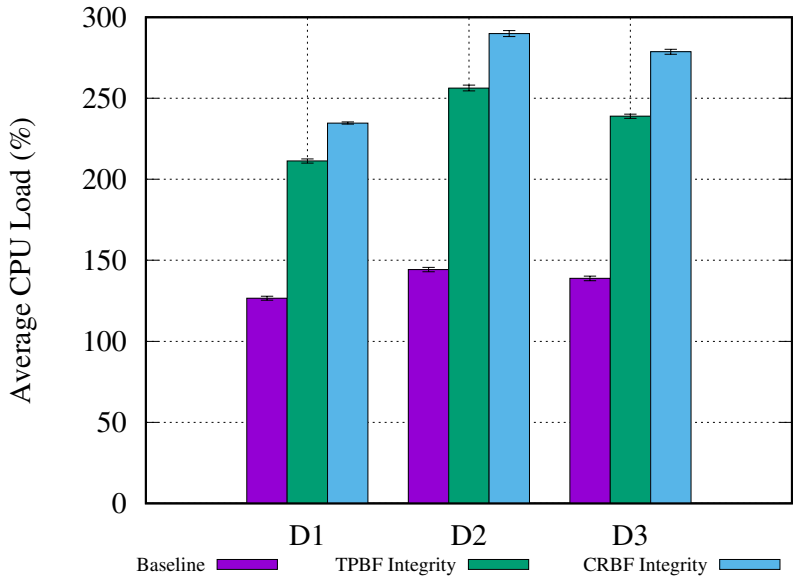


Figure 9. CPU load analysis.

6.2.3. Memory Load Analysis

Bloom filters are known for their memory efficiency. This section analyzes the memory overhead of the proposed framework compared to the baseline, TPBF, and standard Bloom-filter-based data integrity solution, $fSum$ [38]. Figure 10a depicts the memory load (i.e., the total memory consumed by the data transfer application while data transfer is in progress), whereas Figure 10b depicts the Bloom filter memory overhead of the proposed framework.

Figure 10a compares the memory load of baseline (without integrity), TPBF data integrity method, and CRBF- intgrity methods for all workloads described in Section 6.1.2. Figure 10a reveals that the CRBF data integrity method verification exhibits an overhead of 5%, 13%, and 5%, for the D1, D2, and D3 workloads, respectively, compared to baseline. However, compared with the TPBF data integrity method, the overhead is minimal to negligible.

Figure 10b depicts the Bloom filter memory overhead comparison of the proposed data integrity framework to that of $fSum$ and the TPBF method of the data integrity framework. Figure 10b indicates that the CRBF method of data integrity exhibits lower memory requirements than $fSum$ for D1 and D3 workloads. However, we observe slightly higher memory requirements for D2 workloads than $fSum$. Furthermore, we observe negligible overhead for D1 and D2 workloads compared to the TPBF data integrity method. However, similar to $fSum$, D2 workloads have slightly higher memory requirements than the TPBF data integrity method, as the file size and stripe size are the same (i.e., 1 MB). Therefore, based on (7), the CRBF data integrity method exhibits a similar or higher memory overhead than that of the TPBF and $fSum$ solutions. Overall, Figure 10 illustrates that, similar to the TPBF, the proposed CRBF data integrity method is significant regarding memory savings when the dataset contains large or mixed-size files.

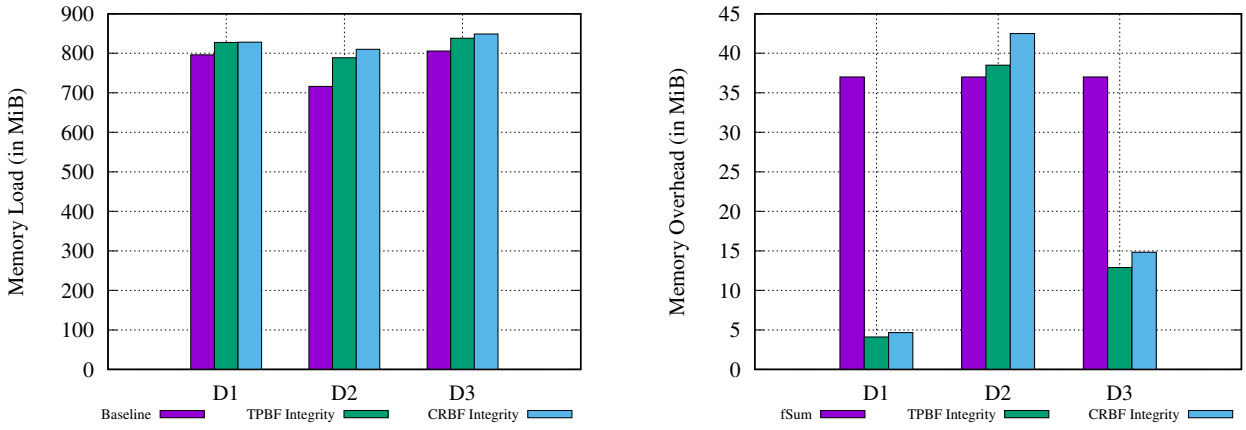


Figure 10. Memory load analysis. (**a**) Memory load comparison. (**b**) Bloom filter memory overhead comparison.

6.2.4. Recovery Overhead and False-Positive Error Analysis

Although Bloom filters are efficient in terms of memory and space, their probabilistic nature can lead to false positives. This means that it is possible for a file to be marked as sent even if it was not actually transferred. As a result, the transferred dataset may be incomplete and not suitable for further analysis. Therefore, avoiding false-positive errors while minimizing the recovery time is another critical design aspect of the proposed data integrity framework. This section presents the recovery time overhead and false-positive error analysis.

To analyze the overhead of recovery and the number of false positives, we artificially introduced errors in the data after transmitting 20%, 40%, 60%, and 80% of the total data. Furthermore, to evaluate the efficacy of the proposed CRBF-based data integrity framework, we randomly modified 10% [16] of the files successfully transferred at the time of the fault.

Figure 11 depicts the recovery overhead of the proposed framework for all workloads, as defined in Section 6.1.2. We represented the time taken before the fault, after the fault, and the total recovery time as stacked histograms. From Figure 11, we can observe that, recovery time is higher at higher fault points. This is because the later the fault point, the higher the number of objects to be processed. For example, the 80% fault point has a recovery time of 2%, 9%, and 3% for the D1, D2, and D3 workloads, respectively. Furthermore, Figure 12 depicts the recovery overhead comparison of the TPBF and CRBF frameworks for all workloads. Figure 12 reveals that, for D1 and D3 workloads, CRBF exhibits slightly a lower overhead than TPBF. However, for D2 workloads, both frameworks exhibit a similar overhead. Thus, the recovery overhead of the CRBF framework is negligible compared to the total data transfer time. In addition, the experimental results show that the proposed framework can successfully detect integrity errors and transfer all modified files during fault recovery.

Figure 13 depicts the false positives of the proposed CRBF framework along with the state-of-the-art *fSum* solution and TPBF framework for D3 workloads. This graph illustrates that the proposed CRBF framework outperforms both *fSum* and the TPBF in eliminating false positives at all simulated fault points.

We ran multiple iterations of the experiments with all workloads, and in all cases, we observed no false positives for any workload at any fault point. Based on these results, we conclude that the CRBF method of data integrity effectively eliminates false-positive errors while minimizing recovery time.

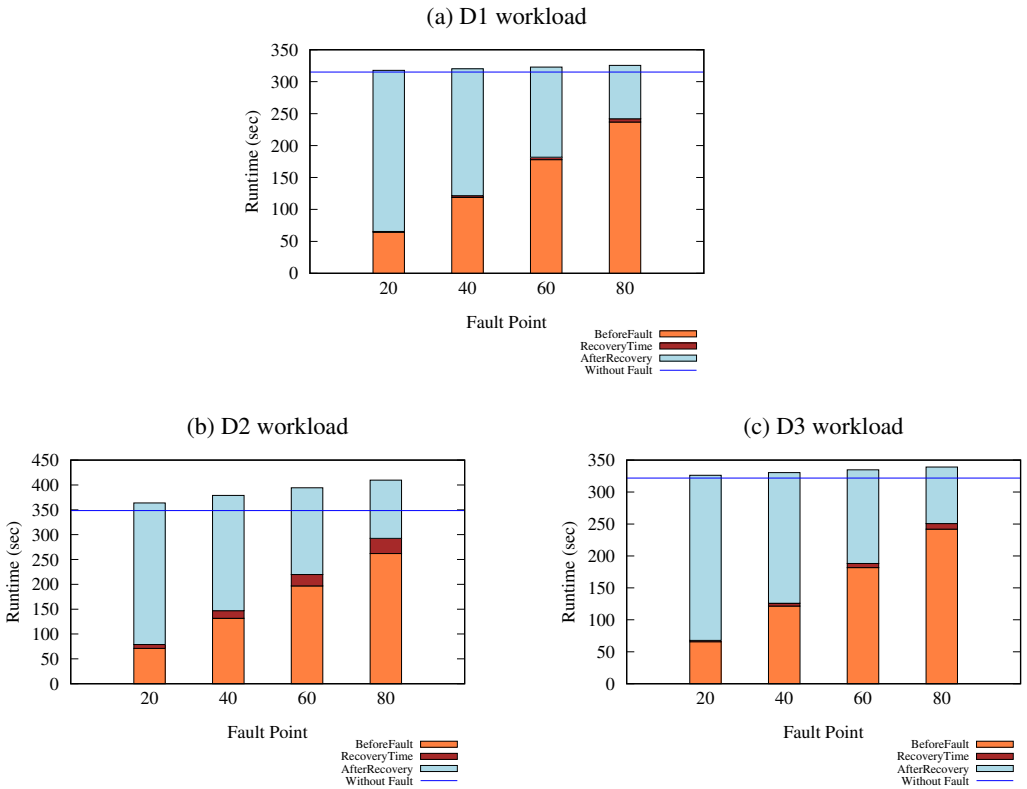


Figure 11. Recovery overhead analysis.

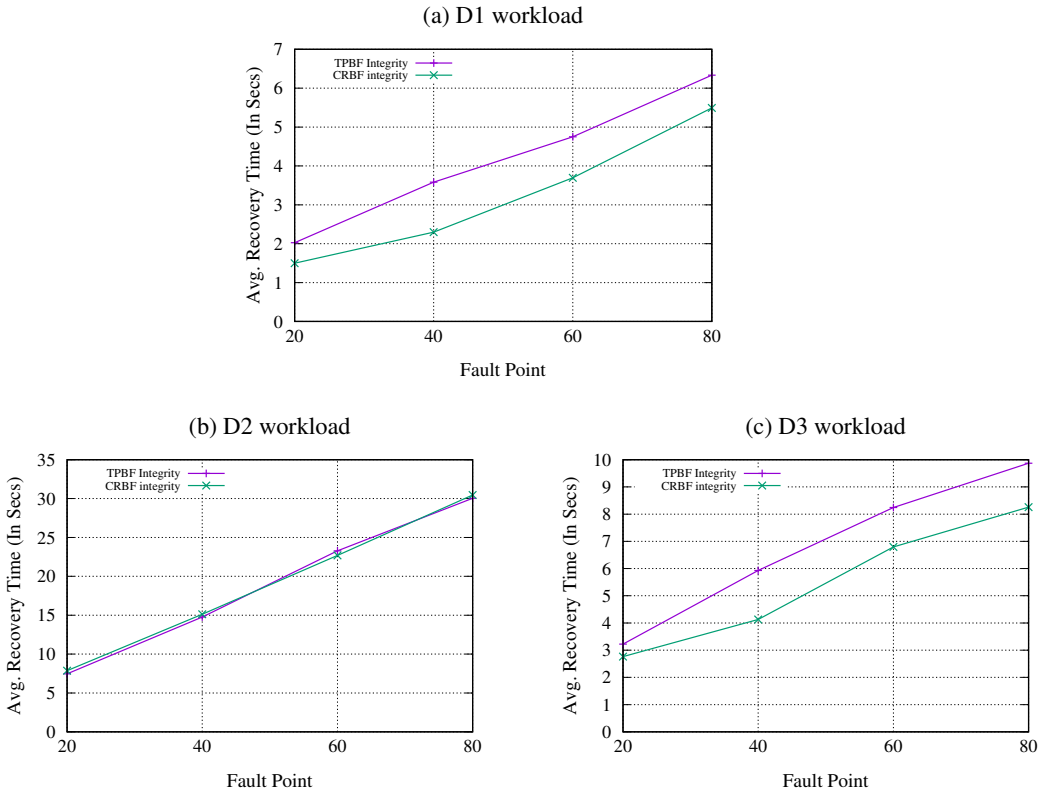


Figure 12. TPBF vs. CRBF recovery overhead analysis.

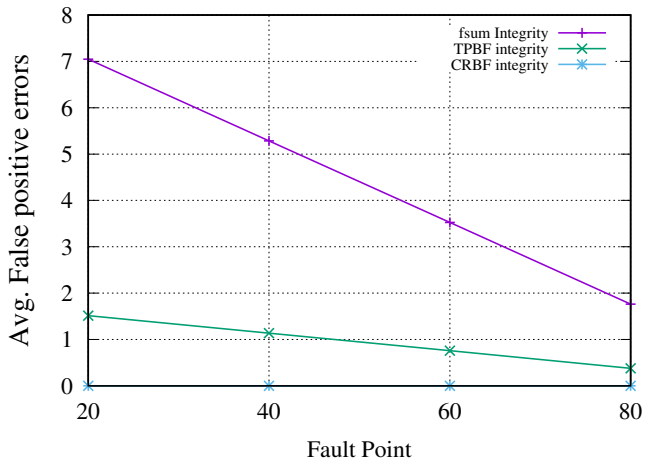


Figure 13. False-positive error analysis.

7. Conclusions

Object-based data transfer systems (OBDTS) are becoming increasingly popular due to their ability to achieve higher data transfer rates than traditional file-based systems. However, OBDTS also introduce new challenges, particularly in ensuring the integrity of transferred data. In this paper, we present a cross-referencing Bloom filter (CRBF)-based integrity verification framework that effectively addresses this challenge. Our framework has comparable computational, storage, and memory requirements to the existing two-phase Bloom filter (TPBF) method. Extensive performance evaluations across various workloads demonstrate the significant performance enhancements achieved by our proposed framework when compared to existing solutions. However, the probabilistic characteristics of the Bloom filter introduces the possibility of false positives. To assess the effectiveness of our framework in mitigating false positives and minimizing recovery time,

we conducted evaluations by simulating faults during data transfer. We also randomly modified 10% of the files that were successfully transferred at the time of the fault to evaluate the reliability of the proposed framework in detecting data integrity errors. The experimental results showed that the proposed framework was able to successfully detect integrity errors and re-transfer all modified files promptly upon fault recovery. Furthermore, the experimental results showed no false positives for any workloads at any fault point.

In summary, the proposed CRBF-based integrity verification framework is highly effective in eliminating false positives and maintaining a similar memory footprint to the TPBF framework. CRBF also enables a lightweight, end-to-end integrity verification framework for object-based big data transfer systems with a minimal impact on overall data transfer performance, typically $< 5\%$.

8. Future Work

The integrity verification framework proposed in this research primarily focuses on the integrity verification of datasets composed of files. However, object-based data transfer systems (OBDTS) can also be used to transfer other types of data, such as streams and databases. Extending the framework to support these other data types would make it more versatile and applicable to a wider range of use cases. Furthermore, it would be beneficial to explore further optimizations to improve the efficiency and scalability of the CRBF-based integrity verification framework. This could be done by exploring techniques such as distributed computing or hardware acceleration to handle larger datasets and higher data transfer rates effectively.

The current study only evaluates the performance of the framework on a small number of workloads. It would be interesting to evaluate the performance of the framework on a wider range of workloads, including both synthetic and real-world workloads. By addressing these future research directions, the proposed CRBF-based integrity verification framework can be further enhanced, extended, and adapted to meet the evolving needs of object-based data transfer systems, ultimately contributing to the advancement of data integrity and reliability in modern computing environments.

Author Contributions: Conceptualization, P.K. and P.H.; Data curation, P.K.; Formal analysis, P.K., P.H. and T.-S.C.; Funding acquisition, T.-S.C.; Investigation, P.K. and P.H.; Methodology, P.K.; Project administration, T.-S.C.; Resources, T.-S.C.; Software, P.K.; Supervision, T.-S.C.; Validation, P.K.; Writing—original draft, P.K.; Writing—review & editing, P.K., P.H. and T.-S.C. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the Institute of Information & communications Technology Planning & Evaluation (IITP) under the Artificial Intelligence Convergence Innovation Human Resources Development (IITP-2023-RS-2023-00255968) grant and the ITRC (Information Technology Research Center) support program (IITP-2021-0-02051), funded by the Korea government (MSIT).

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

Abbreviations

The following abbreviations are used in this manuscript:

CRBF	Cross-Referencing Bloom Filter
PFS	Parallel File System
LADS	Layout-Aware Data Scheduling
OST	Object Storage Target
OSS	Object Storage Server
OBDTS	Object-based Big Data Transfer Systems
TPBF	Two-Phase Bloom Filter
DLBF	Data- and Layout-Aware Bloom Filter
DSBF	Dataset Bloom Filter
MDBF	Metadata Bloom Filter
OLCF	Oak Ridge Leadership Computing Facility

References

1. CERN. Available online: https://home.cern/ (accessed on 20 March 2023).
2. LIGO. Available online: https://www.ligo.caltech.edu/ (accessed on 20 April 2023).
3. ORNL. Available online: https://www.ornl.gov/ (accessed on 20 April 2023).
4. Matsunaga, H.; Isobe, T.; Mashimo, T.; Sakamoto, H.; Ueda, I. Data transfer over the wide area network with a large round trip time. *J. Phys. Conf. Ser.* **2010**, *219*, 062056. [CrossRef]
5. Carns, P.H.; Ligon, W.B., III; Ross, R.B.; Thakur, R. PVFS: A Parallel File System for Linux Clusters. In Proceedings of the 4th Annual Linux Showcase & Conference (ALS 2000), Atlanta, GA, USA, 10–14 October 2000.
6. Enhancing Scalability and Performance of Parallel File Systems. Available online: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/enhancing-scalability-and-performance-white-paper.pdf (accessed on 25 February 2022).
7. Kim, Y.; Atchley, S.; Vallée, G.R.; Shipman, G.M. LADS: Optimizing Data Transfers using Layout-Aware Data Scheduling. In Proceedings of the 13th USENIX Conference on File and Storage Technologies, FAST '15, Berkeley, CA, USA, 16–19 February 2015.
8. Kim, Y.; Atchley, S.; Vallee, G.R.; Shipman, G.M. Layout-aware I/O Scheduling for terabits data movement. In Proceedings of the 2013 IEEE International Conference on Big Data, Santa Clara, CA, USA, 6–9 October 2013; pp. 44–51. [CrossRef]
9. Kim, Y.; Atchley, S.; Vallee, G.R.; Lee, S.; Shipman, G.M. Optimizing End-to-End Big Data Transfers over Terabits Network Infrastructure. *IEEE Trans. Parallel Distrib. Syst.* **2017**, *28*, 188–201. [CrossRef]
10. Kasu, P.; Kim, T.; Um, J.; Park, K.; Atchley, S.; Kim, Y. FTLADS: Object-Logging Based Fault-Tolerant Big Data Transfer System Using Layout Aware Data Scheduling. *IEEE Access* **2019**, *7*, 37448–37462. [CrossRef]
11. Alliance, G. The Globus Toolkit. Available online: http://http://www.globus.org/toolkit/ (accessed on 23 April 2022).
12. Allen, B.; Bresnahan, J.; Childers, L.; Foster, I.; Kandaswamy, G.; Kettimuthu, R.; Kordas, J.; Link, M.; Martin, S.; Pickett, K.; et al. Software as a Service for Data Scientists. *Commun. ACM* **2012**, *55*, 81–88. [CrossRef]
13. Hanushevsky, A. BBCP. Available online: http://www.slac.stanford.edu/~abh/bbcp/ (accessed on 23 April 2022).
14. File Integrity Testing. Available online: https://www.gridpp.ac.uk/wiki/File_Integrity_Testing/ (accessed on 23 April 2022).
15. Stone, J.; Partridge, C. When the CRC and TCP checksum disagree. *ACM SIGCOMM Comput. Commun. Rev.* **2001**, *30*, 309–319. [CrossRef]
16. Kettimuthu, R.; Liu, Z.; Wheeler, D.; Foster, I.; Heitmann, K.; Cappello, F. Transferring a petabyte in a day. *Future Gener. Comput. Syst.* **2018**, *88*, 191–198. [CrossRef]
17. Kasu, P.; Hamandawana, P.; Chung, T.S. TPBF: Two-Phase Bloom-Filter-Based End-to-End Data Integrity Verification Framework for Object-Based Big Data Transfer Systems. *Mathematics* **2022**, *10*, 1591. [CrossRef]
18. Bloom, B.H. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Commun. ACM* **1970**, *13*, 422–426. [CrossRef]
19. Putze, F.; Sanders, P.; Singler, J. Cache-, Hash- and Space-Efficient Bloom Filters. In *Experimental Algorithms, Proceedings of the 6th International Workshop, WEA 2007, Rome, Italy, 6–8 June 2007*; Springer: Berlin/Heidelberg, Germany, 2007; pp. 108–121.
20. Broder, A.; Mitzenmacher, M. Survey: Network Applications of Bloom Filters: A Survey. *Internet Math.* **2003**, *1*, 485–509. [CrossRef]
21. Bloom Filter. Available online: https://en.wikipedia.org/wiki/Bloom_filter (accessed on 20 April 2023).
22. Lim, H.; Lee, N.; Lee, J.; Yim, C. Reducing False Positives of a Bloom Filter using Cross-Checking Bloom Filters. *Appl. Math. Inf. Sci.* **2014**, *8*, 1865–1877. [CrossRef]
23. Luo, L.; Guo, D.; Ma, R.T.B.; Rottenstreich, O.; Luo, X. Optimizing Bloom Filter: Challenges, Solutions, and Comparisons. *IEEE Commun. Surv. Tutor.* **2019**, *21*, 1912–1949. [CrossRef]
24. Kiss, S.Z.; Hosszu, E.; Tapolcai, J.; Ronyai, L.; Rottenstreich, O. Bloom Filter With a False Positive Free Zone. *IEEE Trans. Netw. Serv. Manag.* **2021**, *18*, 2334–2349. [CrossRef]
25. Rottenstreich, O.; Reviriego, P.; Porat, E.; Muthukrishnan, S. Constructions and Applications for Accurate Counting of the Bloom Filter False Positive Free Zone. In Proceedings of the SOSR '20, New York, NY, USA, 3 March 2020; pp. 135–145. [CrossRef]
26. Nayak, S.; Patgiri, R. countBF: A General-purpose High Accuracy and Space Efficient Counting Bloom Filter. In Proceedings of the 2021 17th International Conference on Network and Service Management (CNSM), Virtual, 25–29 October 2021; pp. 355–359. [CrossRef]
27. Tabataba, F.S.; Hashemi, M.R. Improving false positive in Bloom filter. In Proceedings of the 2011 19th Iranian Conference on Electrical Engineering, Tehran, Iran, 17–19 May 2011; pp. 1–5.
28. Kasu, P.; Hamandawana, P.; Chung, T.S. DLFT: Data and Layout Aware Fault Tolerance Framework for Big Data Transfer Systems. *IEEE Access* **2021**, *9*, 22939–22954. [CrossRef]
29. Sivathanu, G.; Wright, C.P.; Zadok, E. Ensuring Data Integrity in Storage: Techniques and Applications. In Proceedings of the 2005 ACM Workshop on Storage Security and Survivability, StorageSS '05, Fairfax, VA, USA, 11 November 2005; pp. 26–36. [CrossRef]
30. Zhang, Y.; Myers, D.S.; Arpaci-Dusseau, A.C.; Arpaci-Dusseau, R.H. Zettabyte reliability with flexible end-to-end data integrity. In Proceedings of the 2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST), Long Beach, CA, USA, 6–10 May 2013; pp. 1–14. [CrossRef]

31. Reyes-Anastacio, H.G.; Gonzalez-Compean, J.; Morales-Sandoval, M.; Carretero, J. A data integrity verification service for cloud storage based on building blocks. In Proceedings of the 2018 8th International Conference on Computer Science and Information Technology (CSIT), Amman, Jordan, 11–12 July 2018; pp. 201–206. [CrossRef]
32. Luo, W.; Bai, G. Ensuring the data integrity in cloud data storage. In Proceedings of the 2011 IEEE International Conference on Cloud Computing and Intelligence Systems, Beijing, China, 15–17 September 2011; pp. 240–243. [CrossRef]
33. Zhang, Y.; Rajimwale, A.; Arpaci-Dusseau, A.C.; Arpaci-Dusseau, R.H. End-to-end Data Integrity for File Systems: A ZFS Case Study. In Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST 10), San Jose, CA, USA, 23–26 February 2010.
34. Lustre, ZFS, and Data Integrity. Available online: https://wiki.lustre.org/images/0/00/Tuesday_shpc-2009-zfs.pdf (accessed on 25 February 2022).
35. Jung, E.S.; LIU, S.; Kettimuthu, R.; CHUNG, S. High-Performance End-to-End Integrity Verification on Big Data Transfer. *IEICE Trans. Inf. Syst.* **2019**, *E102.D*, 1478–1488. [CrossRef]
36. Liu, S.; Jung, E.S.; Kettimuthu, R.; Sun, X.H.; Papka, M. Towards optimizing large-scale data transfers with end-to-end integrity verification. In Proceedings of the 2016 IEEE International Conference on Big Data (Big Data), Washington, DC, USA, 5–8 December 2016; pp. 3002–3007. [CrossRef]
37. Arslan, E.; Alhussen, A. A Low-Overhead Integrity Verification for Big Data Transfers. In Proceedings of the 2018 IEEE International Conference on Big Data (Big Data), Seattle, WA, USA, 10–13 December 2018; pp. 4227–4236. [CrossRef]
38. Xiong, S.; Wang, F.; Cao, Q. A Bloom Filter Based Scalable Data Integrity Check Tool for Large-Scale Dataset. In Proceedings of the 1st Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems, PDSW-DISCS '16, Salt Lake, UT, USA, 14 November 2016; pp. 55–60.
39. Yildirim, E.; Arslan, E.; Kim, J.; Kosar, T. Application-Level Optimization of Big Data Transfers through Pipelining, Parallelism and Concurrency. *IEEE Trans. Cloud Comput.* **2016**, *4*, 63–75. [CrossRef]
40. Mitzenmacher, M. Compressed Bloom filters. *IEEE/ACM Trans. Netw.* **2002**, *10*, 604–612. [CrossRef]
41. George, A.; Mohr, R.; Simmons, J.; Oral, S. *Understanding Lustre Internals*, 2nd ed.; Oak Ridge National Lab. (ORNL): Oak Ridge, TN, USA, 2021. [CrossRef]
42. Lustre File System. Available online: https://docs.csc.fi/computing/lustre/ (accessed on 20 December 2022).
43. Atlas. Available online: https://github.com/ORNL-TechInt/Atlas_File_Size_Data (accessed on 20 April 2023).
44. Kirsch, A.; Mitzenmacher, M. Less Hashing, Same Performance: Building a Better Bloom Filter. *Random Struct. Algorithms* **2008**, *33*, 187–218. [CrossRef]
45. Lu, J.; Yang, T.; Wang, Y.; Dai, H.; Jin, L.; Song, H.; Liu, B. One-hashing bloom filter. In Proceedings of the 2015 IEEE 23rd International Symposium on Quality of Service (IWQoS), Portland, OR, USA, 15–16 June 2015; pp. 289–298.