mathematics

MDPI

Article

TPBF: Two-Phase Bloom-Filter-Based End-to-End 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, Korea; kasu@ajou.ac.kr
[2] Department of Computer Science and Engineering, Soongsil University, Seoul 06978, Korea; princeh@ssu.ac.kr
* Correspondence: tschung@ajou.ac.kr

Abstract: Computational science simulations produce huge volumes of data for scientific research organizations. Often, this data is shared by data centers distributed geographically for storage and analysis. Data corruption in the end-to-end route of data transmission is one of the major challenges in distributing the data geographically. End-to-end integrity verification is therefore critical for transmitting such data across data centers effectively. Although several data integrity techniques currently exist, most have a significant negative influence on the data transmission rate as well as the storage overhead. Therefore, existing data integrity techniques are not viable solutions in high performance computing environments where it is very common to transfer huge volumes of data across data centers. In this study, we propose a two-phase Bloom-filter-based end-to-end data integrity verification framework for object-based big data transfer systems. The proposed solution effectively handles data integrity errors by reducing the memory and storage overhead and minimizing the impact on the overall data transmission rate. We investigated the memory, storage, and data transfer rate overheads of the proposed data integrity verification framework on the overall data transfer performance. The experimental findings showed that the suggested framework had 5% and 10% overhead on the total data transmission rate and on the total memory usage, respectively. However, we observed significant savings in terms of storage requirements, when compared with state-of-the-art solutions.

Keywords: big data; geo-distributed data centers; data integrity; Bloom filter; parallel file system; high-performance computing

MSC: 68M14; 68M15

Citation: 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. https://doi.org/10.3390/math10091591

Academic Editor: Shih-Wei Lin

Received: 9 April 2022
Accepted: 2 May 2022
Published: 7 May 2022

1. Introduction

A terabyte(s) to a petabyte(s) of data is generated every day by modern scientific centers such as ORNL [1], CERN [2], and LIGO [3]. Furthermore, in the modern age, every single entity is associated with some digital components or equivalents that may generate data. Security cameras, mobile phones, smart home devices, and telemetry devices are just a few examples of the many electronic devices that are constantly generating data and digital content. There are more than 2.5 quintillion bytes generated by internet users every day [4], and this pace has been increased by current technologies, such as AI (artificial intelligence), IoT (Internet of Things), and ML (machine learning).

Service providers distribute their data centers geographically all over the globe to give better service to consumers in terms of availability and response time, depending on their location. This leads to a huge growth in the need for high-performance data transmission across the data centers distributed geographically. Although network speeds are approaching petabits per second and storage capacity is approaching a zettabyte, there

is still a significant performance gap between network and storage. As a result, increasing end-to-end data transmission rates has become a major challenge for service providers.

Data centers are equipped with PFSs (parallel file systems) to decrease the impedance mismatch between the network and storage and also to increase the scalability [5,6]. Consequently, the PFSs help to deliver the much required inter-data-center high-performance data transfers. However, resources in such PFSs, are highly shared among users. Owing to this, users might be contending for the same resources. As contention for these resources increases, there can be a significant gap between the actual and estimated I/O performance [7,8]. To avoid such contention issues and to increase the data transfer rates, scholars have suggested object-based big data transfer frameworks [9–12] to prevent temporarily congested servers from transferring data.

Object-based big data transfer frameworks [9–12] utilize the file(s) storage architecture and boost data transmission rates by sending the objects of multiple files in parallel. Due to the object nature of data and concurrent processing, data transfer frameworks might send objects of one logical file from the host to the destination in an out-of-order fashion. If the end-to-end path has any faults, the out-of-order nature of the data transmission will result in data integrity errors. To avoid such data integrity issues, after recovering from the fault, this kind of data transfer framework necessitates the retransmission of the whole file data (objects), causing undesirable congestion.

For the authenticity of big data, the integrity of the data that is transferred must be guaranteed. While several techniques (e.g., TCP checksum) [13] currently maintain point-to-point data integrity in transit, explicit end-to-end data integrity validation protects scenarios that may go unreported by usual in-transit procedures. End-to-end data integrity in big data transfer systems refers to the reliability and trustworthiness of data throughout the data transfer lifecycle. Without end-to-end data integrity support, it is not possible for the data transfer system to ensure whether the data that is transferred to the sink-end PFS is the same as the data that users or applications have created at the source-end PFS. Using end-to-end data integrity verification, it is possible to protect the data against hardware issues (network or servers), software bugs, and storage write errors (misplaced or partial writes) by the hard disks or SSDs, and more.

Checksumming is the most basic implementation of end-to-end data integrity [14]. Both sink and source endpoints retrieve the data from storage after a successful data transfer and then calculate the checksum with a hash technique including SHA1 [15] or MD5 [16]. The transfer is considered successful if the calculated checksums at both ends are the same; otherwise, the file at the sink end is deemed to be damaged, and the transmission is redone. Conventional checksum tools are both serial and file-based. Therefore, data integrity verification frameworks based on the checksum method fall short as far as large-scale datasets are concerned. If the dataset contains large files, then the checksum computation either fails, owing to the size of the data, or takes an extremely long time to process. File-based checksums, on the other hand, are cumbersome and unusable when the dataset comprises millions of small files.

End-to-end data integrity verification in large scale data transfers is not only essential but also very expensive. It increases the amount of disk I/O and processing needed for the data transfer, thereby reducing the overall data transfer performance [17,18]. In this work, our aim is to design a data integrity verification framework for object-based big data transfer systems. Owing to the object nature of the data transfer, it is possible to transfer the objects of a logical file in a random order. Therefore, serial and file-based checksumming-based data integrity verification methods are not suitable.

To address the limitations of the traditional serial and file-based checksumming tools, Xiong et al. [19] proposed a standalone, scalable data integrity check tool for large-scale datasets. This tool breaks the files in the data storage into chunks of reasonable size and calculates chunk-level checksums in parallel. However, the parallel nature of the execution results in checksum ordering issues. To overcome checksum ordering issues, as well as to optimize the memory, a Bloom filter data structure is used. Though this method is

efficient in data storage systems, this method of data integrity falls short in detecting and recovering from silent data corruption errors in the end-to-end path of the data transfer. In this work, our aim is to support an end-to-end data integrity verification framework for object-based big data transfer systems while minimizing the impact on the overall data transfer performance.

In this paper, our proposed two-phase Bloom-filter (TPBF)-based data integrity verification framework aims to reduce the memory and storage footprint without affecting the data transfer performance. To the best of our knowledge, our proposed framework is the first of its kind to address end-to-end data integrity verification for object-based big data transfer systems. The major contributions of our work are listed below.

- A data- and layout-aware Bloom filter (DLBF) mechanism for effectively handling object and file level data integrity verification with object-based big data transfer systems.
- For efficiently handling dataset level integrity verification, we developed a two-phase Bloom-filter (TPBF)-based end-to-end data integrity verification framework for optimizing the memory and storage footprint when compared with state-of-the-art data integrity solutions.
- We utilized a Lustre file system [20–22] that interacts over an InfiniBand (IB) network [23,24] to evaluate the proposed design. Based on the experimental results, we can conclude that the proposed data integrity framework is very effective at detecting and resolving data integrity issues at all of object, file, and dataset levels.
- Data transfer performance, memory, and storage overhead have been evaluated to assess the overhead of the proposed end-to-end integrity verification framework in the context of data transmission. The experimental findings show that the suggested framework had 5% and 10% overhead on the total data transmission rate and on the total memory usage, respectively. Moreover, we observed significant $\geq$50% savings in terms of storage requirements, when compared with state-of-the-art solutions.
- The false-positive error rate was evaluated to assess the effectiveness of the proposed data integrity framework by manually inducing faults after transferring 20%, 40%, 60%, and 80% of the total data. Our experimental results showed that the proposed framework significantly reduced false-positive errors and was up to 80% more effective than current state-of-the-art solutions.

The rest of the paper is structured as follows. The background and motivation for our work is described in Section 2. Section 3 summarizes related work. Section 4 discusses the design and implementation details of the proposed two-phase Bloom-filter-based data integrity verification framework. Section 5 highlights the evaluation results and in Section 6, we conclude our study.

2. Background and Motivation

2.1. Background

2.1.1. Object-Based Big Data Transfer Systems

Traditional big data transfer frameworks ignore the underlying storage system architecture and rely solely on file logical representation [25–27]. Due to this, objects of the same file are transferred in sequence. If a single I/O thread is assigned to transfer the file, it will work on that file sequentially until the entire file is transferred. As only one file is transferred at a time, the big data transfer framework consumes a considerable amount of time to transfer all the files in the dataset. To improve the data transfer performance, it is possible to assign multiple I/O threads to process the data transfer. However, employing multiple I/O threads without knowledge of the physical distribution of the file might result in disk contention issues as multiple threads compete for the same object storage server (OSS) or object storage target (OST) [8,21,28]. Due to this contention, the data transfer performance of the application will be degraded.

In contrast to the traditional big data transfer frameworks, object-based big data transfer systems address storage contention issues by considering the physical distribution of the files across different OSTs [11,29]. Owing to this, the workload is considered as

objects rather than files, where each object represents a maximum transmission unit (MTU) of data. Object-based big data transfer systems utilize file(s) layout information to load balance the OSTs. Therefore, a thread can be assigned to an object of any file on any OST without requiring that all the objects of a particular file be transferred before objects of another file. Due to this enhanced layout-aware scheduling and parallel processing, object-based big data transfer systems avoid OST contention and hence, improve data transfer rates [9–11,30].

Figure 1 depicts an illustration of OST contention in file and object-based big data transfer systems. In this example, we consider two files, $File_a$ and $File_b$, that are distributed across four different OSTs (OST_1 to OST_4). As shown in Figure 1a, traditional big data transfer systems assign T_1 and T_2 I/O threads for transferring $File_a$ and $File_b$, respectively. On initiating the data transfer, owing to the serial and file based data transfer nature of the traditional big data transfer systems, both T_1 and T_2 threads attempt to access the first object of $File_a$ and $File_b$, respectively. However, as both the objects are placed in OST_1, one of the threads has to wait until the other thread completes its job. As shown in Figure 1a, thread T_2 experiences the delay and hence consumes more time to read the first object of $File_a$. Similarly, threads T_1 and T_2 contend for resource OST_3 while processing the third object of $File_a$ and $File_b$. Due to this, as shown in Figure 1a, thread T_2 execution is delayed and hence consumes more time to process the third object of $File_a$. This kind of resource contention has a negative impact on the overall data transfer rates in traditional big data transfer systems.

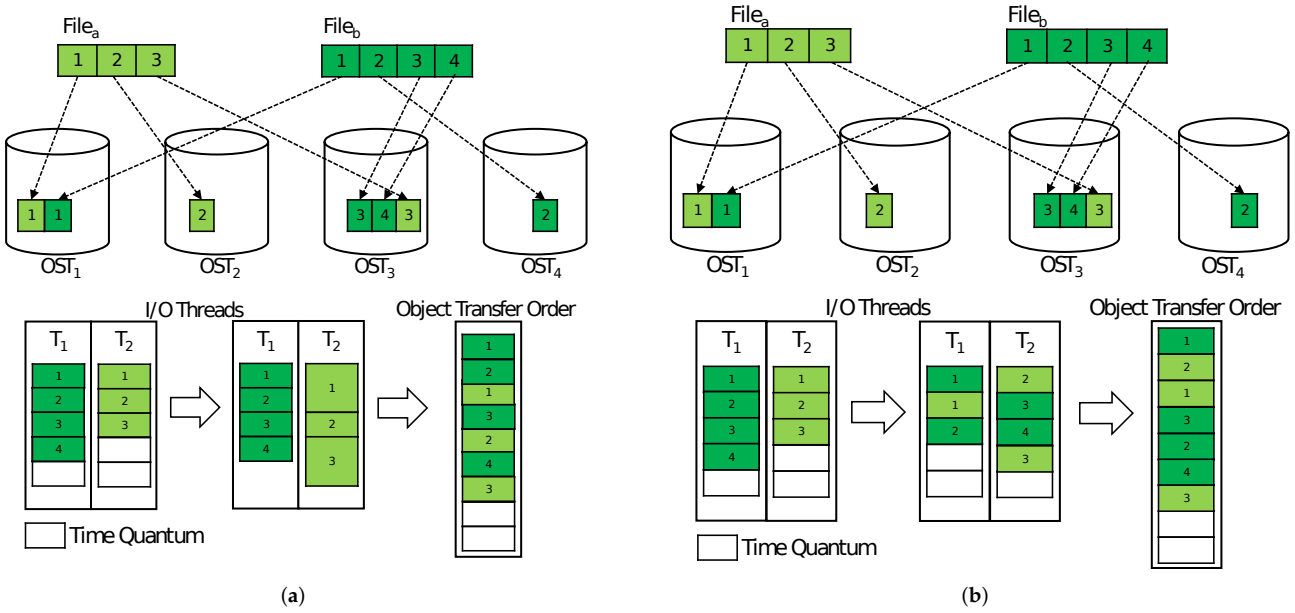

Figure 1. Illustration of OST contention in file and object-based transfer systems. (**a**) File-based transfer system. (**b**) Object-based transfer system.

On the other hand, object-based big data transfer systems ensure that no two threads compete for the same resource at any given time by scheduling the objects using their enhanced layout-aware and congestion-aware scheduling algorithms [9–11]. For example, as shown in Figure 1b, threads T_1 and T_2 contend for resource OST_1 while processing the first object of $File_a$ and $File_b$. To avoid such a contention, the layout-aware scheduler schedules the second object of $File_a$ ahead of the first object. A similar mechanism is used to schedule all other objects of the dataset. Though this method of object scheduling avoids resource contention, it also results in out-of-order object transfers. From Figure 1b, we can observe that the second object of $File_a$ is transferred first, followed by the first object. Similarly, we can observe the out-of-order object transfer for $File_b$ too. A similar kind of

out-of-order object transfer mechanism can be observed for all other files in the dataset. Due to these improved layout-aware data scheduling algorithms and data parallelism, the data throughput of object-based big data transfer systems is much higher than the traditional big data transfer frameworks [9–11,30].

Despite the fact that object-based big data transfer systems significantly increase data transfer rates, the out-of-order nature of data transmission, as well as parallel processing, necessitates complex sorting algorithms, as well as higher storage and memory requirements, to support end-to-end data integrity. Therefore, in this study, we propose a TPBF-based integrity verification framework for handling end-to-end data integrity in object-based big data transfer systems.

2.1.2. End-to-End Data Integrity

Figure 2 depicts the scope of the network and end-to-end data integrity verification during large scale data transfers. As shown in Figure 2, various components, such as source and sink host machines, storage systems, and network elements, are involved in the end-to-end path of the data transfer. Network data integrity accounts for the data corruption issues over the network and will be handled by the network stack. In addition to the network, data corruption can also happen at storage systems during file read or write operations due to hardware, storage media, firmware, and controller malfunction. This leads to undetected silent data corruption errors. Recovering from such errors will incur significant overheads, and it may not always be possible for the storage devices to fully recover from these errors [31]. Thus, end-to-end data integrity verification accounts for the integrity verification between source and sink storage devices.

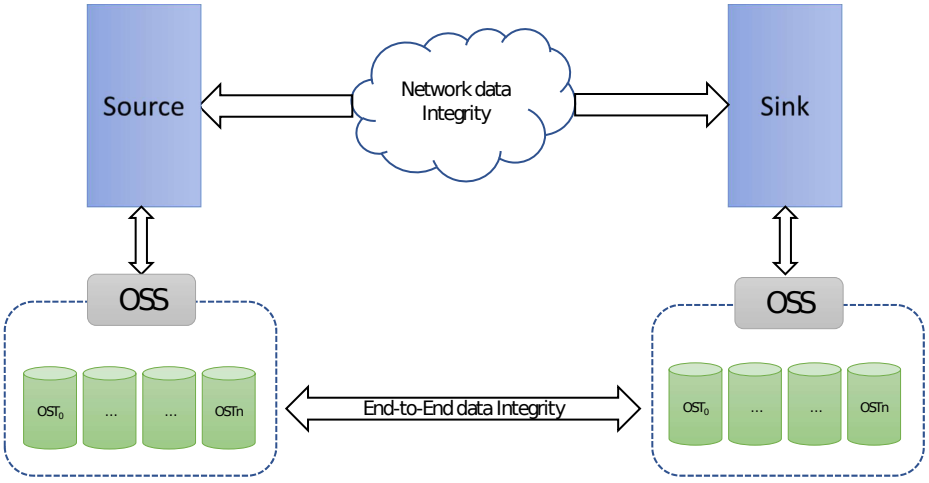

Figure 2. End-to-end data integrity.

While many intermediate data transfer components, such as file systems [32–35], support integrity verification, they are insufficient to ensure end-to-end data integrity. The only way to ensure that the data transferred to the sink-end PFS is the same as the data created by a user or application at the source-end PFS is to perform end-to-end data integrity verification. In this study, we propose methods for ensuring block, file, and dataset level end-to-end data integrity verification.

2.1.3. Big Data Transfer Frameworks

Big data transfer frameworks face two primary issues when transferring huge amounts of data across geographically diverse data centers: high performance and reliability. Researchers have proposed different big data transfer frameworks to address high-performance and reliability requirements.

One of the most popular protocols for rapid data transmission on the grid is GridFTP [26,27]. This protocol is an extended version of the FTP (file transfer protocol),

which provides a general purpose mechanism for transmitting data in a reliable, secure, and high-performance manner. This framework employs a parallel data transfer mechanism to aggregate the overall bandwidth by employing multiple transmission control protocol (TCP) streams. Striping is also used in this framework to facilitate multi-host-to-multi-host data transmission. GridFTP also handles file-level data corruption errors by incorporating on-the-fly checksum-based data integrity verification. However, this serial and file-based method of data integrity verification falls short as far as large-scale datasets are concerned. If the dataset contains large files, then the checksum computation either fails owing to the size of the data or takes an extremely long time to process. Hence, the parallel checksum approach is needed and preferred. In this study, we consider transferring the data as objects based on their layout; hence, the serial and file-based data integrity verification mechanisms cannot be implemented in our framework.

The BaBar Copy Program (BBCP) [25] is a potential alternative to GridFTP for transmitting huge volumes of data between data centers. Similar to GridFTP, this tool also divides a transfer into several concurrent streams. As a result, data is transferred at substantially higher rates than using single stream utilities, such as SCP (secure copy protocol) and SFTP (SSH file transfer protocol). Since BBCP sequentially transfers the whole file data, it supports network data integrity verification (data integrity verification only during network transfer). However, this is not really end-to-end, since it does not account for all the data integrity issues along the path between the host and destination endpoints.

The eXtreme DD Toolset (XDD) [12] offers the software infrastructure required for transferring large scale datasets with high reliability and performance. It has several configurable features that make it easier to transfer files efficiently, including threads, device access methods, impedance matching, as well as I/O scheduling principles. Owing to its improved I/O scheduling policies, objects of the same file are transferred in an out-of-order manner from the source to the destination. Similar to our framework, this tool also considers data transmission as objects rather than files. However, the data integrity verification was not addressed in this tool.

Layout aware data scheduling (LADS) [9–11] leverages storage architecture at each endpoint to optimize throughput while not affecting the functionality of shared resources for other users. Therefore, LADS considers the physical view of the files rather than their logical view. Owing to the physical view of the files, as well as concurrent processing, this framework transfers the objects of the same logical file from the source endpoint to the sink endpoint in an out-of-order fashion. However, similar to XDD, this object-based big data transfer tool also does not provide any solution to handle data integrity verification.

Table 1 summarizes the level of data integrity verification supported by different data transfer tools. According to this table, the GridFTP big data transfer framework supports both network data integrity and file-level end-to-end integrity verification. However, the serial and file-based nature of the data integrity verification of GridFTP will have a significant negative effect on the overall data transfer performance. Although, the BBCP data transfer tool supports network data integrity verification, it does not support any type of end-to-end integrity verification. Furthermore, the table shows that none of the object-based data transfer tools supports data integrity verification.

Table 1. Summary of data integrity verification.

Data Transfer Tool	Network Integrity	End-to-End Integrity		
		Object	File	Dataset
Grid FTP	Yes	No	Yes	No
BBCP	Yes	No	No	No
XDD	No	No	No	No
LADS	No	No	No	No

2.2. Motivation

Object-based big data transfer systems, as described in Section 2.1.1, exploit the storage layout architecture to enhance the data transfer rates. Consequently, while transferring data, the physical distribution of the workload is taken into account rather than the logical nature of the files. Thus, the workload is transferred in the form of objects, not files. Owing to the object nature of the workload, as well as the parallel processing, objects of different logical files are transferred in parallel, and objects of the same logical file are transferred out-of-order, as shown in Figure 1b. Although this type of mechanism significantly improves the data transfer performance, complex methodologies need to be employed for ensuring data integrity during the data transfer.

To ensure the authenticity of the data, big data transfer frameworks should handle the data integrity verification during the data transfer. Traditional big data transfer tools, such as GridFTP [26,27] and BBCP [25], transfer the file data sequentially, as shown in Figure 1a. Due to this, it is feasible to support data integrity verification using traditional serial and file-based checksum techniques. For example, GridFTP generates a file-level root hash by concatenating all the block-level checksums into a hash list and then computing the top hash from that or by employing a Merkle tree. However, object-based big data transfer systems transfer objects of the same logical file in an out-of-order fashion. Due to this, object checksums required for data integrity verification are also generated in an out-of-order manner. Therefore, generating a single and consistent file or dataset level signature for data integrity verification necessitates the need for complex sorting algorithms, as well as higher memory and storage requirements due to the sheer size of the data.

The major challenges for supporting data integrity with object-based big data transfer systems are memory, storage, and computational overheads. In our research, we aimed to address the following issues:

- How can the impact of the data integrity framework on the overall data transfer rate of object-based big data transfer systems be minimized?
- How can the memory and storage requirements of the data integrity framework be reduced?

To address the aforementioned challenges, we propose a two-phase Bloom-filter-based end-to-end data integrity verification framework for reducing the memory, storage, and computational overheads of object-based big data transfer systems.

3. Related Work

Ensuring the correctness of the transferred data is one of the primary concerns of data transfer frameworks. Many studies have been performed on the design and implementation of data integrity verification and their optimization in data storage [34,36,37], cloud storage [38–42], file systems [32–35,43], databases [32,44,45], and data transfer systems [18,19,46,47].

GridFTP [26,27], a widely used protocol for scientific data transfer, has built-in data integrity verification support. This file transfer service computes a 128-bit checksum by reading the file at the destination after it is written to the disk, and the same is compared against the source-end checksum to verify file data integrity. It supports file-level pipelining to minimize the integrity verification overhead. File-level pipelining transfers the file data by overlapping with the checksum calculation of the previously transferred file in multi-file transfers. However, this pipelining approach fails when the dataset comprises mixed file sizes, where the file transfer takes longer than the checksum computation of the previously transferred file or vice versa. To optimize this, Jung et al. [18,46] proposed block-level pipelining (with various block sizes) to overlap the processes of data transfer and checksum computation. As GridFTP transfers the logical file data sequentially, concatenating all block-level checksums to a hash list and calculating the top hash based on it (or) using a Merkle tree to generate the root hash is possible. However, our work considers transferring the data as objects based on their layout; hence, the objects of the same logical file are transferred out-of-order. Therefore, to generate a hash list or a Merkle tree, storing the

checksums of all objects is necessary. However, this method is not feasible for datasets containing large file sizes. Hence, this method cannot be implemented using our data transfer framework.

I/O overhead is reduced and improved pipelining is achieved by using FIVER [47] for data integrity verification. FIVER focuses on overlapping checksum calculation and transfer procedures for the same file, unlike prior research that focused on overlapping operations for distinct files or blocks. In addition, as opposed to reading each file from storage twice (once for file transfer and the second time for checksum computation), FIVER reads the files only once and shares the I/O between checksum and transfer operations. It offers the same level of integrity protection as other existing techniques, but with a significant performance advantage. However, similar to GridFTP, FIVER also transfers logical file data sequentially. Hence, this method of data integrity is not suitable for our data transfer framework.

Xiong et al. [19] proposed a scalable parallel dataset checksumming tool called *fsum*. It is built upon the principle of parallel tree walk and work stealing patterns to maximize parallelism and overcome the limitations of traditional serial and file-based checksumming tools. This method of data integrity generates a single and consistent dataset-level signature by aggregating chunk-level checksums. However, owing to the parallel nature of execution, checksums are generated in different orders. Therefore, sorting is necessary to generate root hashes. However, it is not scalable because the root process has to gather checksums from all processes and store them in memory as a hash list or a Merkle tree at some point. To avoid these drawbacks due to sorting, a Bloom filter is used to aggregate the chunk-level checksums. This method of data integrity is both memory and computation efficient than other approaches. However, owing to its false-positive detections, data integrity errors went unnoticed when duplicate contents were present in the dataset. In this study, our proposed two-phase Bloom filter structure effectively handles the end-to-end integrity verification for all types of datasets by reducing the storage footprint and false-positive detections when compared with the *fsum* tool.

4. Data Integrity Verification Framework

In this section, we present the design and implementation details of the proposed data integrity verification framework. First the design aspects of the Bloom filter are described. Next, we present two-phase Bloom filter (TPBF) design and implementation details. Finally, we conclude by analyzing the memory requirements of the proposed two-phase Bloom filter.

4.1. Bloom Filter Design

In this section, we first describe the Bloom filter data structure. Subsequently, we focus on the hash function optimization details to reduce computational overhead. Next, we discuss the design details of the data and the layout-aware Bloom filter (DLBF) [48] used for verifying the file-level data integrity. Finally, we conclude by illustrating an example of DLBF for insert and query operations.

4.1.1. Bloom Filter Data Structure

A Bloom filter is a compact data structure which supports constant time insert and query operations [49–51]. The Bloom filter representing a set of n elements is composed of m-bits. Initially, all these m-bits of the Bloom filter array are set to 0. Bloom filter utilizes k independent hash functions, for inserting an element or for querying the membership of an element. These hash functions generate k hash values that are uniformly distributed over the range, $1 \ldots m$. When an element is inserted, the bits at these k-positions in the Bloom filter array are set to 1. When an object membership query is performed, the Bloom filter array values at these k-positions are compared, and, if any of the bits at these positions is equal to 0, then the element is presumed to be not present in the set. Otherwise, we assume

that the element is in the set. However, if any of the bits at these k-positions were set as a result of hash collisions, this assumption leads to false-positive errors.

The probability that an object is not existing in the set, often known as the false-positive error probability, is calculated as below. For a given number of hash functions, k, and larger m, the probability, p, that a specific bit will remain 0, after all the entries in a dataset have been hashed into the Bloom filter, may be described as

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

As a result, the probability that the bit is 1 is represented as

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

While testing the membership of an object that is not present in set, the false-positive error probability, ϵ, can be represented as

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

The number of objects in the dataset (n), hash functions (k), and total filter size (m) all influence the Bloom filter's false-positive error probability (ϵ). Practically k would be an integer; a suboptimal smaller k is preferable since it minimizes the number of hash functions to be computed and thus reduces the overall computational overhead of the Bloom filter. The number of hash functions needed to reduce false-positive errors for a given m and n can be represented as

$$k = \frac{m}{n} ln2. \quad (4)$$

4.1.2. Hash Optimization

Multiple independent hash functions are used for building a Bloom filter. Thus, hash functions are the core computational operations of the Bloom filter. Therefore, minimizing the computational overhead of these hash functions is very much necessary for optimizing the overall Bloom filter computation. For optimizing hash function computational overhead, researchers have proposed strategies to generate distinct hash values using a fewer number of hash functions [52–54]. Using this approach, in our experiments, we employed *murmur* [55] and *DJB2* [56] as base hash functions, and additional hash values were generated using these two hash algorithms. Specifically, Equation (5) was used to compute the additional k hash values:

$$h_i(x) = h_1(x) + i * h_2(x) \mod m \quad (5)$$

where:

i $= 0 \leq i \leq k - 1$
m = Bloom filter size

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

Space efficiency and insertion order independence are the two major design considerations in using a Bloom filter data structure for supporting data integrity verification with object-based big data transfer systems. However, the probabilistic nature of the Bloom filter results in false-positive errors [57,58]. Therefore, in this section, we present a modified Bloom filter, the data and layout aware Bloom filter (DLBF), for efficiently handling the false-positive errors of a standard Bloom filter.

A data- and layout-aware Bloom filter is generated by mapping the object of an arbitrary size to a fixed size using an SHA-1 engine as depicted in Figure 3. This block hash is deterministic and serves as the input to the hash functions.

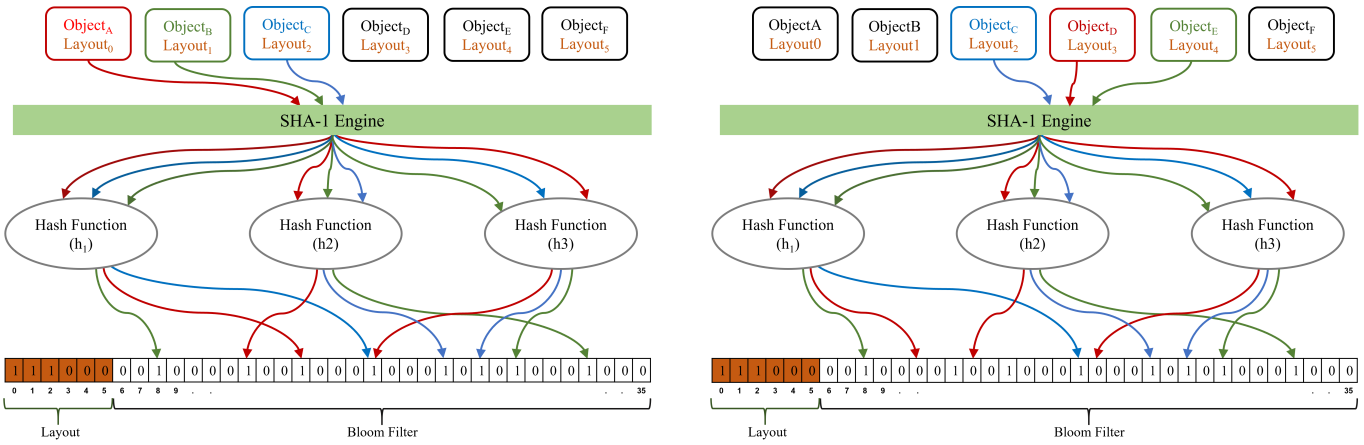

Figure 3. Illustrative example of data- and layout-aware Bloom filter data structure. (**a**) Insert operation. (**b**) Query operation.

The Bloom filter, as described in Section 4.1.1, is an m-bit vector (B) with k independent hash functions ($h_1, \ldots, h_k$) that translates every element in a dataset ($S = \{x_1, \ldots, x_n\}$) to a range ($R_m = \{0, 1, \ldots, m-1\}$). We suppose that every hash function h_k uniformly maps each element in the dataset to a random integer with equal probability across the range R_m. All m-bits of bit vector B are initially set to "0".

- **Insert:** For each object $x_i \in S$, compute $h_1(x_i), \ldots, h_k(x_i)$ and set $B[h_1(x_i)] = B[h_2(x_i)] = \ldots = B[h_k(x_i)] = 1$.
- **Query:** To check whether an object, x_i, is in S, compute $h_1(x_i), \ldots, h_k(x_i)$. If $B[h_1(x_i)] = B[h_2(x_i)] = \ldots = B[h_k(x_i)] = 1$, the answer is yes; otherwise, the answer is no. However, if $h_1(x_i), \ldots, h_k(x_i)$ in the bit vector B are set to 1 as a result of hash collisions, then it results in false-positive errors.

To avoid such false-positive membership query results of the Bloom filter, the object layout data (n-bits) is prepended as additional information about the object. As a result, the Bloom filter's total size is extended to (n+m) bits, and all these bits are initially set to "0".

- **Insert:** For each object $x_i \in S$, compute $h_1(x_i), \ldots, h_k(x_i)$, and set $B[n + h_1(x_i)] = B[n + h_2(x_i)] = \ldots = B[n + h_k(x_i)] = 1$, and also set the object layout information bit, $B[i] = 1$. Where, 'i' represents the layout of the object.
- **Query:** To check whether an object, x_i, is in S, compute $h_1(x_i), \ldots, h_k(x_i)$. If $B[n + h_1(x_i)] = B[n + h_2(x_i)] = \ldots = B[n + h_k(x_i)] = 1$ and $B[i] = 1$, the answer is yes; otherwise, the answer is no.

As shown above, by using object layout information along with the standard Bloom filter, we are able to avoid the false-positive errors of the Bloom filter data structure.

4.1.4. Illustration of Data and Layout Aware Bloom Filter

Figure 3 shows an example of a data- and layout-aware Bloom filter. In this example, we considered the total number of objects in the dataset $n = 6$, the number of hash functions $k = 3$, and the total size of the Bloom filter as $m = 30$. For demonstration purposes, we illustrate the example where objects A, B, and C are inserted into the Bloom filter and objects C, D, and E to exhibit success, fail, as well as false-positive match queries.

The Bloom filter array is categorized into two sections, as illustrated in Figure 3, the Bloom filter and the layout sections. Following successful object transfer and integrity veri-

fication, the layout section of the Bloom filter array is filled with object layout information. On the other hand, the Bloom filter section of the array is used to map the objects into k positions randomly by utilizing k independent hash functions. All $(n + m)$ bits of the filter array are set to zero when the data transfer is initialized.

- **Insert:** The insert operation is shown in Figure 3a. To uniquely represent the object, the SHA-1 engine is employed to calculate the block hash on the dataset. In this illustrative example, objects A, B, and C are inserted into the Bloom filter. Hash functions $\{h_1, h_2,$ and $h_3\}$ are employed on the hashed object data to uniformly map the objects into k random positions. The Bloom filter bits at positions {13, 16, and 20} are set to 1 using the $\{h_1, h_2,$ and $h_3\}$ hash functions on hashed $Object_A$ data. Additionally, bit {0} of the Bloom filter array is set to 1 as the layout of $Object_A$ is zero. Similarly, bits at positions {1, 8, 28 32}, and {2, 20, 24, 26} are set to 1 for $Object_B$ and $Object_C$, respectively.
- **Query:** The query operation is shown in Figure 3b. We considered objects C, D, and E for membership query operation. We presume the object membership if all the k bits in the Bloom filter section, along with the layout bit in the layout sections, are set to 1. For $Object_C$ the Bloom filter returns "Positive" for membership query as the hash positions {20, 24, and 26} along with its layout bit at position {2} is set to 1. The membership query of $Object_D$ returns "Negative" as the bit at position {11} is not set. On the other hand, $Object_E$ membership query results in "Negative", despite the fact that the bits at positions {8, 28, and 32} are all set. This is due to the fact that the object layout bit at position {4} is not set. Without the layout information, $Object_E$ membership query may result in "False Positive" since the bits at positions {8, 28, and 32} are all set. Hence, we prevented false-positive matches of the Bloom filter by utilizing the object layout information in conjunction with the Bloom filter.

4.2. System Architecture

Figure 4 depicts the system architecture of the proposed data integrity verification framework for object-based big data transfer systems. *File Handler* and *Object Handler* are the two core components of the data integrity verification framework. The *File Handler* manages the file-level activities, such as scheduling the file objects for transfer and performing the file level data integrity verification. On the other hand, the *Object Handler* handles the core operations of data transfer, as well as object-level data integrity management.

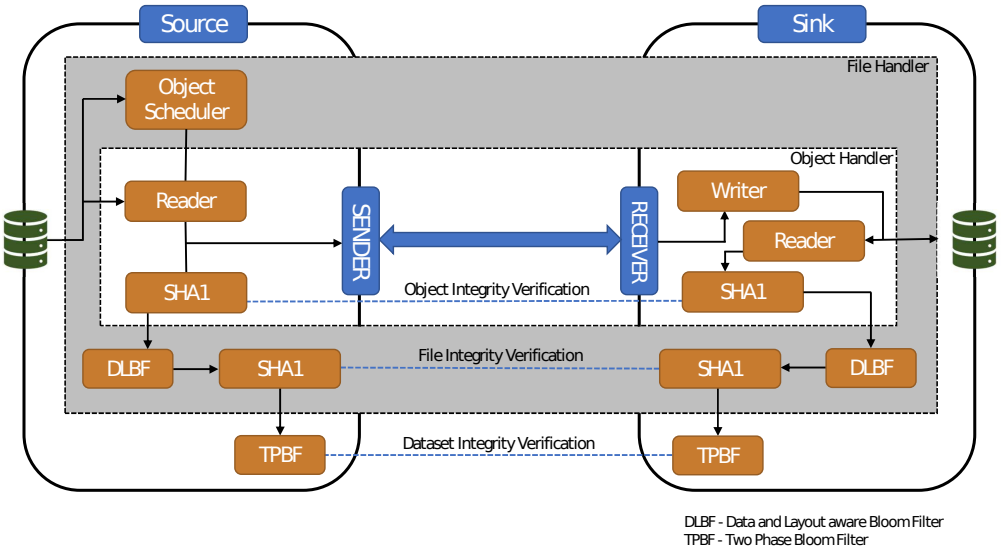

Figure 4. Two-phase Bloom-filter-based end-to-end data integrity framework architecture.

On initiating the data transfer, the source end *File Handler* prepares the list of the files to be transferred and schedules the file(s) objects for transfer. The sink-end *File Handler*

acknowledges the objects and files that were successfully written to the sink-end PFS after ensuring the data integrity. As shown in Figure 4, DLBF and TPBF are used to maintain information about the successfully transferred objects and files in the dataset, respectively. On successful object transfer and integrity verification, DLBF is populated with k-hash positions, computed on the object signature, along with its layout information. Similarly, upon successful file transfer and integrity verification, TPBF is populated with k-hash positions computed on the file level signature. If a fault occurs during the data transfer and the transfer is resumed from the fault point, TPBF and DLBF are used to retrieve the successfully transferred files and file(s) objects, respectively. These successfully transferred files and objects will be excluded from the re-transfer. For the remaining objects, the source-end *File Handler* schedules the transfer. This process is repeated until all objects and files in the dataset are successfully transferred to the sink endpoint without any integrity errors.

4.3. Design and Implementation

The design and implementation details of the proposed two-phase Bloom-filter-based data integrity verification framework is described in this section.

4.3.1. Communication Protocol

The communication protocol of the proposed data integrity framework is as shown in Figure 5 and Listing 1 lists the communication messages between the source and sink endpoints. On initiating the data transfer,

1. The source endpoint sends a CONNECT request to the sink endpoint, and the sink endpoint responds with SUCCESS if the connection is successful.
2. The source endpoint compiles a list of files to be transferred and then issues a NEW_FILE request for each file. The sink endpoint opens the file based on the information in the NEW_FILE request and adds the file descriptor to the FILE_ID response.
3. The source endpoint schedules all the objects of a file and initiates object transfer using NEW_OBJECT request. The sink endpoint receives the object data and writes the same to the sink-end PFS. On successful write operation, the sink endpoint compares the block hash with the hash received in the NEW_OBJECT request and responds with OBJECT_SYNC.
4. On successful integrity verification, both source and sink endpoints aggregate the file-based data and the layout-aware Bloom filter; otherwise, the source endpoint schedules the object for re-transfer.
5. Steps 3 and 4 are repeated for all the objects in the file.
6. On transferring all the objects of a file successfully, the sink endpoint compares the file hash with the hash received in the last object's NEW_OBJECT request and responds with FILE_CLOSE response.
7. On successful integrity verification, both source and sink endpoints aggregate the dataset level two phase Bloom filter; otherwise, the source endpoint schedules the file for re-transfer.
8. Steps 2 to 7 are repeated for all the files in the dataset.
9. After successfully transferring all of the files in the dataset, dataset level integrity verification is performed. If the integrity check is successful, the source endpoint will send a DISCONNECT request; otherwise, steps 2 to 9 will be repeated.

Listing 1. Communication message type.

```
typedef enum msg_type {
CONNECT = 0,     //Connection Request
SUCCESS,         //Connection accepted
NEW_FILE,        //New File request
FILE_ID,         //Sink File ID.
NEW_OBJECT,      //Ready for object transfer
OBJECT_SYNC,     //Sync with Sink PFS
FILE_CLOSE,      //File close
DISCONNECT       //Ready to disconnect
} msg_type_t;
```

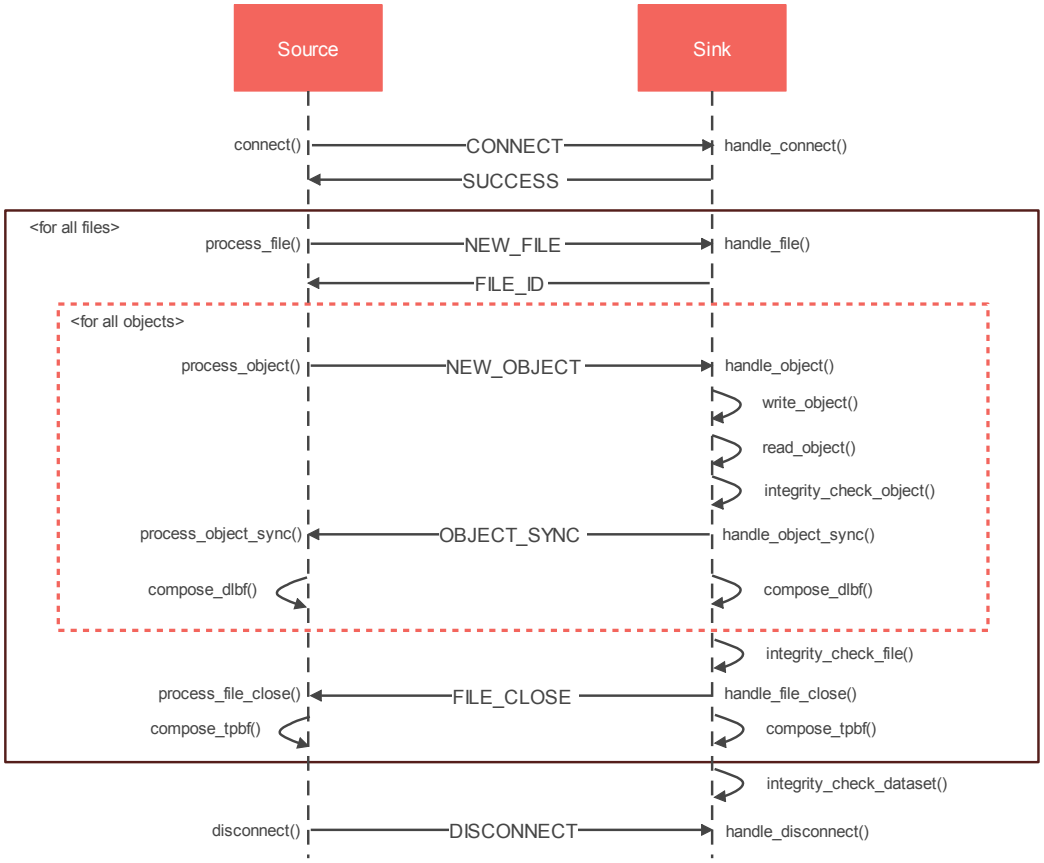

Figure 5. Communication sequence of two-phase Bloom-filter-based end-to-end data integrity framework.

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

The design and implementation aspects of DLBF are described in this section. Figure 6 depicts the flowchart of the proposed data integrity framework. On initiating the data transfer, the source-end *object handler* reads the object data from storage and computes a block hash, to uniquely represent the object, using the SHA1 engine and issues a transfer request to the sink. Upon receiving the object data, the sink-end *object handler* writes the object data to the storage and, on successful write operation, the *object handler* reads the object data back from the storage and then computes the object hash using the SHA1 engine. This computed hash value is compared against the received hash value from the source endpoint to validate the block data. If both the hash values are the same, then it marks the block as successful and acknowledges the source endpoint about the successful object transfer. Otherwise, it marks the block as corrupted and then sends a retransmit request to the source endpoint. Upon successful completion of the block transfer, both source and sink

endpoints update the file level, data- and layout-aware Bloom filter (DLBF) using the block hash. On receiving the retransmit request from the sink endpoint, the source endpoint schedules the block again for transfer. This procedure is repeated until all the objects of the logical file have been successfully transmitted to the sink endpoint.

Algorithm 1 depicts the pseudo-code for generating the data- and layout-aware Bloom filter (procedure *GenerateDLBF*). As shown in Algorithm 1, *k*-hash functions are used for generating the *k*-hash bit positions. On successful object transfer and integrity verification, the *k*-hash bit positions along with the object layout bit of DLBF are set to 1. On transmitting all the objects of a logical file, a file level signature is generated by hashing file level DLBF.

Algorithm 1 Two-phase Bloom Filter

procedure GENERATEDLBF((S))
  for each $s_i \in \mathcal{S}$ **do** ▷ For all objects in a file
    $Obj_{sig}(s_i) \leftarrow SHA1(s_i)$ ▷ Map object of arbitrary size to fixed size
    for each $j \leftarrow 1$ to $j \leq k$ **do** ▷ k hash functions
      $pos \leftarrow h_j(Obj_{sig}(s_i))$ ▷ Calculate k hash bit positions
      $S_{bf}(pos) = 1$ ▷ Set k hash positions of DLBF to 1
    end for
    $S_{bf}(i) = 1$ ▷ Set the layout bit of DLBF to 1
  end for
  $S_{sig} \leftarrow SHA1(S_{bf})$ ▷ File signature
  return S_{sig}
end procedure

procedure GENERATETPBF((N))
  for each $f_i \in \mathcal{N}$ **do** ▷ For all files in a dataset
    $File_{sig}(f_i) \leftarrow GenerateDLBF(f_i)$ ▷ Generate file level DLBF
    for each $j \leftarrow 1$ to $j \leq k$ **do** ▷ k hash functions
      $pos \leftarrow h_j(File_{sig}(f_i))$ ▷ Calculate k hash bit positions
      $N_{bf}(pos) = 1$ ▷ Set k hash positions of TPBF to 1
    end for
  end for
  $N_{sig} \leftarrow SHA1(N_{bf})$ ▷ Dataset signature
  return N_{sig}
end procedure

4.3.3. Two-Phase Bloom Filter (TPBF)

This section describes the design and implementation aspects of the TPBF. As shown in Figure 6, after successfully transferring all of the objects of a logical file, both the endpoints compute the file level hash from the aggregated data- and layout-aware Bloom filter (DLBF). This hash value is verified to ensure the file level data integrity. If the computed hash value matches at both the endpoints, then the file transfer is considered successful and the dataset level-two-phase Bloom filter (TPBF) is updated at both endpoints using the hash value of DLBF. Otherwise, it marks the file as corrupted and schedules it for retransmission. This procedure is repeated until all of the files in the dataset have been successfully transmitted to the sink endpoint.

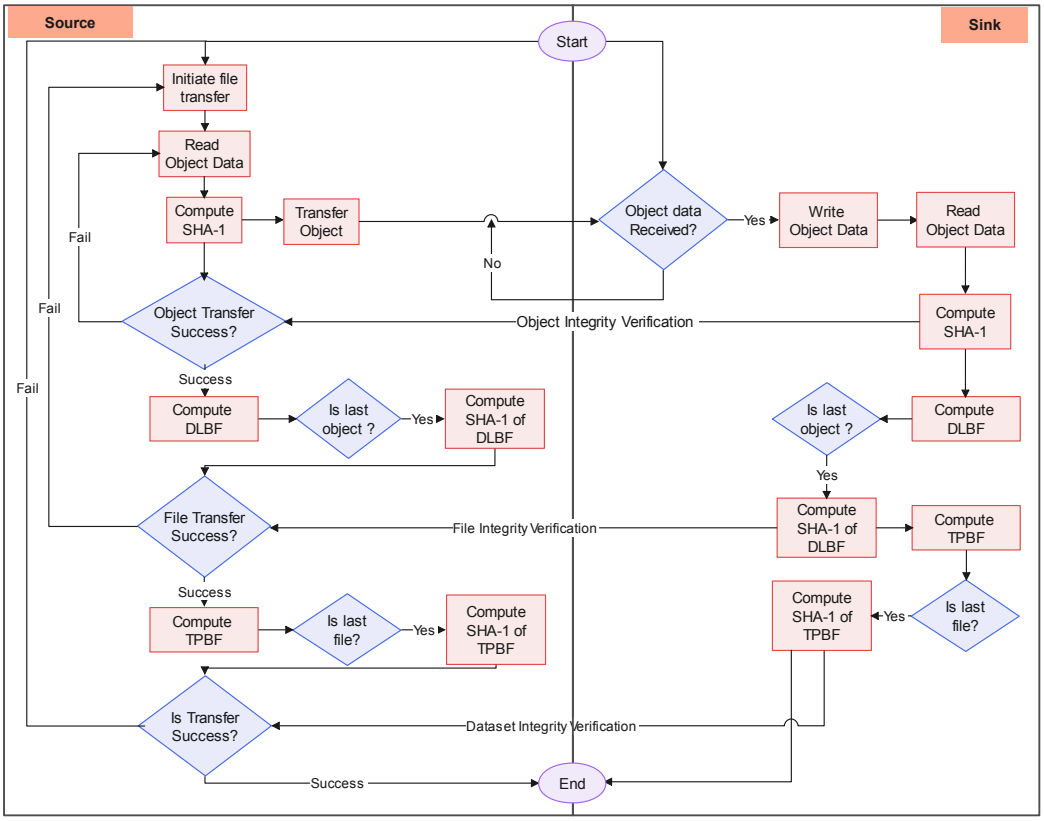

Figure 6. Flow chart of two-phase Bloom-filter-based end-to-end data integrity framework.

The pseudo-code for generating the two-phase Bloom filter (procedure *GenerateTPBF*) is represented by the Algorithm 1. As shown in Algorithm 1, k-hash functions are used for generating k-hash bit positions. On successful file transfer and integrity verification, the k-hash bit positions of TPBF are set to 1. After transferring all of the files in the dataset, the dataset level signature is computed from the aggregated two-phase Bloom filter. This signature is used to verify the dataset level data integrity.

4.4. Memory Overhead Analysis

In this section, we analyze and compare the memory overhead of the proposed two-phase Bloom filter with that of a state-of-the-art Bloom filter-based data integrity solution [19].

Given the number of elements to be inserted, n, the desired false-positive probability, ϵ, and the number of hash functions, k, the number of bits required for the Bloom filter, m, can be computed by substituting the value of k from Equation (4) in the probability expression, Equation (3).

$$\epsilon = (1 - e^{(\frac{m}{n} ln2)\frac{n}{m}})^{\frac{m}{n} ln2}. \tag{6}$$

which can be simplified as

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

Thus, the optimal number of bits required are

$$m = -\frac{nln\epsilon}{ln2^2} \tag{8}$$

Consider a dataset with a total number of files, N, each with S objects or blocks. For this dataset, for a given false-positive probability, the minimum number of bits required for the standard Bloom-filter-based data integrity is

$$m = -\frac{(N * S) ln\epsilon}{ln2^2} \tag{9}$$

Assuming $S == N$ and ignoring the constant values, we can approximate the number of bits required as follows:

$$m \approx (N^2) \tag{10}$$

Whereas, for the two-phase Bloom filter, the total number of bits required are the sum of bits required for both DLBF and TPBF Bloom filters. Substituting n at each phase of the Bloom filter in Equation (7), the total number of bits required are

$$m = -(\frac{(S * C) ln\epsilon}{ln2^2}) + -(\frac{(N) ln\epsilon}{ln2^2}) \tag{11}$$

where

C = number of active file transfers

Assuming $S == N$ and $C \ll N$, and ignoring the constant values, we can approximate the number of bits required as

$$m = -\frac{(2N) ln\epsilon}{ln2^2} \approx N \tag{12}$$

From Equation (12), we can observe that the number of bits required for the two-phase Bloom filter are linearly proportional to the number of elements in the dataset, while Equation (10) suggests the number of bits required by the state-of-the-art Bloom-filter-based data integrity solution has a quadratic relation with the number of elements in the dataset. From this, we can conclude that the proposed two-phase Bloom filter efficiently reduces memory and storage requirements.

5. Evaluation

This section describes the testbed and workload specifications and provides the experimental findings, along with their interpretation. First, we assess the overhead of the proposed data integrity verification framework on the overall data transfer performance. Subsequently, we explore the effectiveness of the suggested two-phase Bloom-filter-based data integrity verification framework by comparing the computational time overhead, memory overhead, storage overhead, and the number of false-positive errors with that of a state-of-the-art solution, which is based on fsum [19]. All of our trials were carried out under the same conditions.

5.1. Testbed and Workload Specifications

5.1.1. Testbed

For our evaluations, we employed a testbed with source and sink nodes interfaced by an InfiniBand (IB) network interface. Servers powered by Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.10GHz, having 16GB DRAM and 32 cores with Linux kernel version 3.10.0-1062, were employed at both source and sink endpoints. Furthermore, both these servers employed the Lustre file system (version 2.9.0) [59] with a single OSS (object storage server) and four OSTs (object storage targets), mounted over a 1TB drive. The Lustre file system at both the endpoints was configured with a stripe count (the stripe count defines the number of OSTs a file is written across) of four and a stripe size (stripe size refers to the size of the stripe written as a single block to an OST) of 1 megabyte (MB).

5.1.2. Workload

We utilized the Lustre Atlas 1 and 2 file system distribution data [60] provided by the "Oak Ridge Leadership Computing Facility" [1] to examine the file size distribution. The file size versus number of files is shown in Figure 7. It may be seen from the statistics that 91.55 percent of the files were under 4 MB and 84.17 percent were under 1 MB. Moreover, only around 10 percent of the files were larger than 4 MB, even though such files took up the majority of the file system capacity. As a result, we employed three sets of files with varied sizes for our assessment. The first group (D1) had large workloads (100 1 GB files); the second group (D2) contained smaller workloads (100,000 1 MB files); and the third group (D3) contained mixed workloads with 50 1 GB files, 10,000 4 MB files, and 10,000 1 MB files. For evaluating our framework, we populated the source file system with D1, D2 and D3 workloads by striping the data across all the source end OSTs. As described in Section 5.1.1, a stripe count of four and a stripe size of 1 MB were used for striping the data to OSTs.

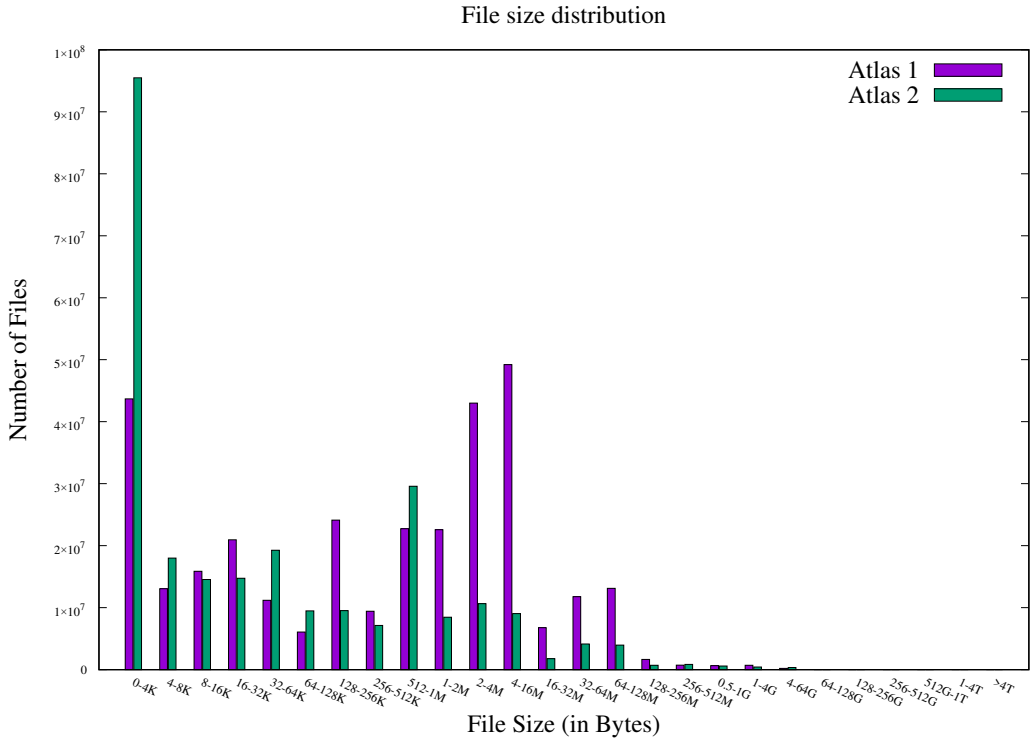

Figure 7. File size distribution.

5.1.3. Bloom Filter Configuration

The false-positive rate, ϵ, is a function of the Bloom filter size, m as expressed in Equation (9). Thus, increasing the Bloom filter size reduces the false-positive error rate and vice versa. Therefore, a trade-off between the Bloom filter size and the false-positive error rate is an important design aspect of the Bloom filter. To optimize the memory and storage overheads of the Bloom filter, the false-positive error rate was considered as $\ll 0.0001\%$ for all workloads. Based on this configuration, the Bloom filter size was calculated using Equation (8) and the optimum number of hash functions was calculated using Equation (4).

5.2. Performance Evaluation

5.2.1. Data Transfer Time

Using a two-phase Bloom-filter-based data integrity verification framework, we investigated the effect on the overall data transfer time. The data transmission time for all workloads, as described in Section 5.1.2, is shown in Figure 8. In this figure, we compared the total data transfer time of the proposed two-phase Bloom filter (TPBF) with that of the

standalone (without integrity) data transfer time. We represented the total data transfer time for both standalone and TPBF solutions as bar graphs. Error bars were also used to denote the 99% confidence intervals.

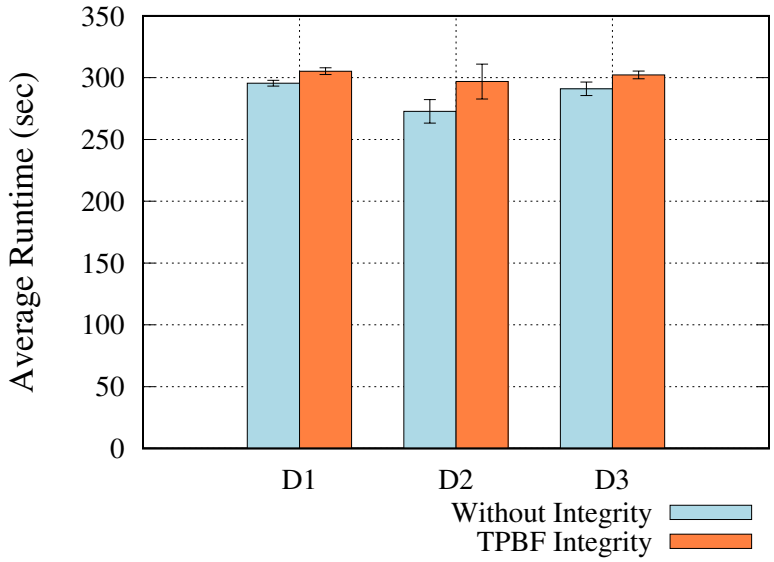

Figure 8. Data transfer time.

From Figure 8, we observe that, the TPBF method of data integrity verification exhibited overheads of 3%, 7%, and 4%, respectively, for D1, D2, and D3 workloads. Despite the fact that TPBF computation is expensive, we optimized the overhead by using hash optimization and pipelining techniques. Hash optimization, as stated in Section 5.1.2, minimizes the computational overhead of hash functions and hence the Bloom filter. Along with the hash optimization, we also utilized a pipelining technique to achieve high-performance data transfer with two-phase Bloom-filter-based data integrity. Our pipeline included the read(write), hash, Bloom filter generation, and data transfer operations. Each function in our system was designed to overlap with the activities of a different block. On the whole, based on the results of Figure 8, we can infer that the TPBF method of data integrity had a minimum to insignificant impact on the overall data transmission time.

5.2.2. Computational Overhead

One of the primary goals of the suggested TPBF method of data integrity is to minimize the total data transmission time by reducing the computational overhead. The computational overhead of the proposed TPBF method of data integrity is estimated as below

$$DI_t = TP_t - SA_t \tag{13}$$

where,

DI_t = Estimated data integrity overhead
TP_t = TPBF average runtime
SA_t = Standalone average runtime

Figure 9 depicts the computational overhead of the proposed data and layout aware TPBF method of data integrity with a state-of-the-art solution, *fsum* for all the workloads, as described in Section 5.1.2. From Figure 9, we observe that the overhead of the proposed data integrity verification framework was much less than that of the state-of-the-art solution. However, we observed slightly higher overhead for the D2 workloads when compared with the D1 and D3 workloads. This variation may have been due to the Lustre file system's file management (metadata retrieval) overhead. Based on this, we can conclude that the

proposed data- and layout-aware TPBF method of data integrity had fewer computational requirements than the state-of-the-art solution.

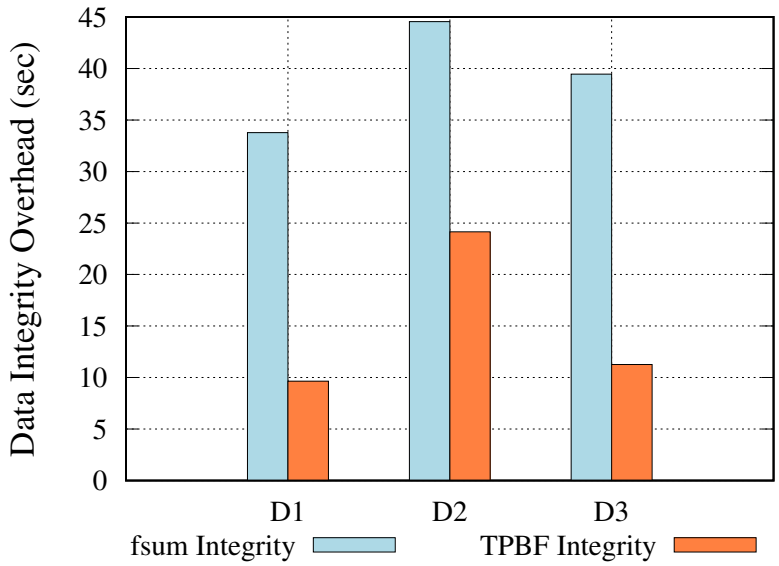

Figure 9. Computational overhead comparison of fsum and TPBF.

5.2.3. Memory Overhead

Memory efficiency is a prominent aspect of the Bloom filter data structure. Section 4.4 describes the memory overhead analysis of the proposed TPBF method of data integrity. In this section, we discuss the memory load requirements for the TPBF data integrity verification framework.

Figure 10 depicts the memory load comparison for all workloads, as described in Section 5.1.2. In this figure, we compared the total memory load of the proposed two-phase Bloom filter with that of the standalone method (without integrity). The bar graph is used to represent the memory load for both the standalone and TPBF methods. From Figure 10, we observe that the TPBF method of data integrity exhibited an overhead of 10% for all workloads. However, this was quite a lot less compared with the traditional serial and file-based data integrity mechanisms, which needed ≈2 GiB of memory to store all the chunk level checksums (assuming each chunk is represented using a 160-bit SHA-1 hash value). Overall, from Figure 10, we can conclude that the suggested TPBF method of the data integrity verification framework had a negligible influence on the overall memory requirements.

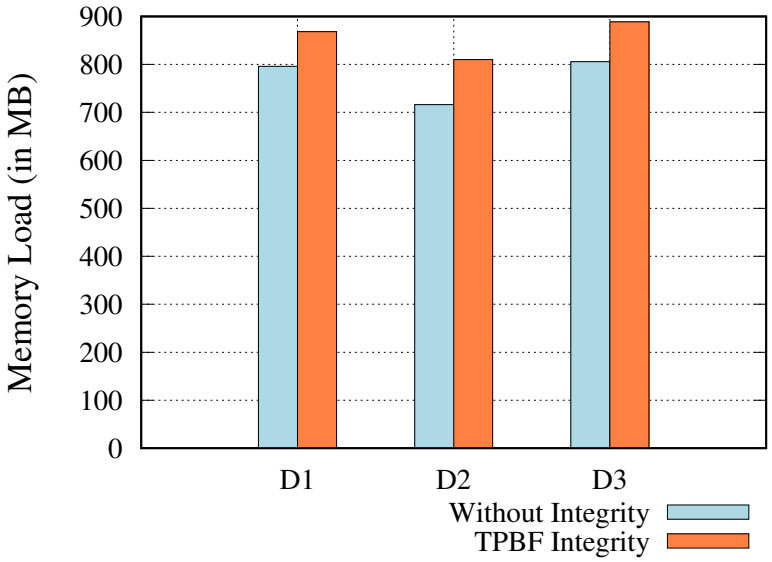

Figure 10. Memory usage.

5.2.4. Storage Overhead

Another important aspect of adopting the TPBF technique for data integrity is reducing the amount of storage occupied by the data integrity framework metadata during data transfer. Figure 11 depicts the storage overhead of the state-of-the-art data integrity solution, fsum, along with the TPBF method of data integrity for all workloads, as described in Section 5.1.2. For analysis, we calculated the theoretical storage overhead for the state-of-the-art data integrity solution, *fsum*, by configuring the Bloom filter with a similar false-positive error rate as the TPBF method of data integrity.

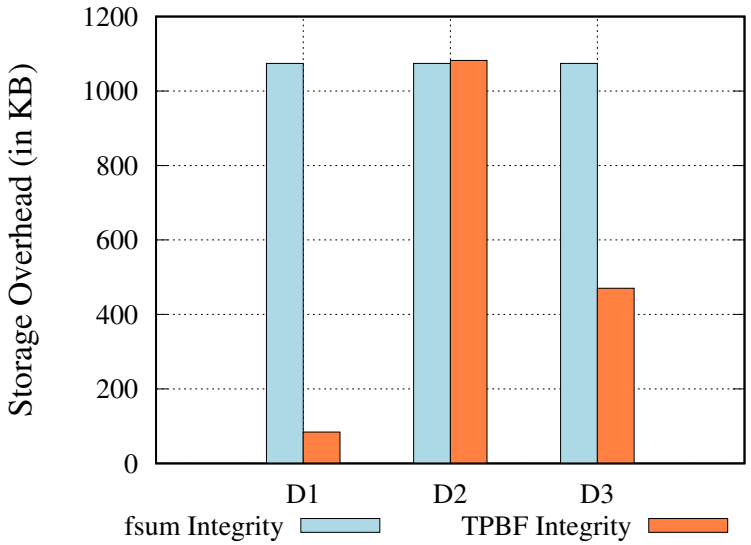

Figure 11. Storage overhead comparison of fsum and TPBF.

From Figure 11, we can observe that, for big and mixed workloads (D1 and D3), the storage overhead of the proposed TPBF method of data integrity was considerably lower than that of the state-of-the-art solution. This is because the storage overhead of the TPBF method of data integrity is directly proportional to the number of files in the dataset. In contrast, the state-of-the-art solution has a quadratic relationship with the number of files in the dataset, as described in Section 5.1.2. However, we observed a slightly higher storage overhead for small workloads (D2) than that of the state-of-the-art solution. This is because the file size and the stripe size of the small workload were the same, that is, 1 MB. Hence,

the total number of objects to be transferred matched the total number of files. As a result, the storage overhead associated with the TPBF data integrity approach was equivalent to that associated with the state-of-the-art solution. In general, as seen in Figure 11, the TPBF method of data integrity resulted in significant savings with respect to the storage requirements.

5.2.5. False-Positive Matches

Owing to the probabilistic nature of the Bloom filter, the TPBF method of data integrity framework is prone to false-positive matches for membership queries. As a result, certain items that have not been moved to the sink end are incorrectly assumed to have been sent. As a consequence, data gets corrupted, rendering it unsuitable for further processing. Reducing the false-positive matches of the object membership queries is another important design aspect of this framework. In this section, we present the average false-positive matches of the proposed TPBF method of the data integrity framework for all the workloads defined in Section 5.1.2. Additionally, to assess the suggested solution's efficacy, we compared the false-positive matches of the TPBF method of data integrity verification framework to the state-of-the-art solution, *fsum*.

To analyze false-positive matches, we created and simulated faults after transmitting 20%, 40%, 60%, and 80% of the total data. Due to the fact that these faults might occur at any point in the end-to-end data transfer path, it is possible to simulate these faults at either end of the data transfer. However, in our verification, we generated these faults at the source endpoint. In addition, as the number of false-positive errors varied from iteration to iteration, we presented the average number of false-positives detected over multiple iterations of the experiments.

Figure 12 depicts the average false-positive matches for all workloads, as described in Section 5.1.2, at varying fault points. We can see from Figure 12 that the average number of false-positive matches was quite low for all the workloads. In addition, we can see that the later the fault point, the lower the frequency of the false-positive matches. This is because the later the fault point, the lower was the number of objects to be transferred to the sink end.

Figure 13 illustrates the average false-positive errors of the proposed TPBF method of data integrity with a state-of-the-art solution, *fsum*, for mixed workloads ($D3$), as described in Section 5.1.2. From this Figure 13, we observe that the proposed data and layout-aware TPBF method of data integrity outperformed the state-of-the-art solution, *fsum*, and was 80% more efficient in avoiding false-positive matches at all simulated fault points. With this, we can conclude that the TPBF data integrity verification framework was superior in avoiding false-positive errors compared to the state-of-the-art solution.

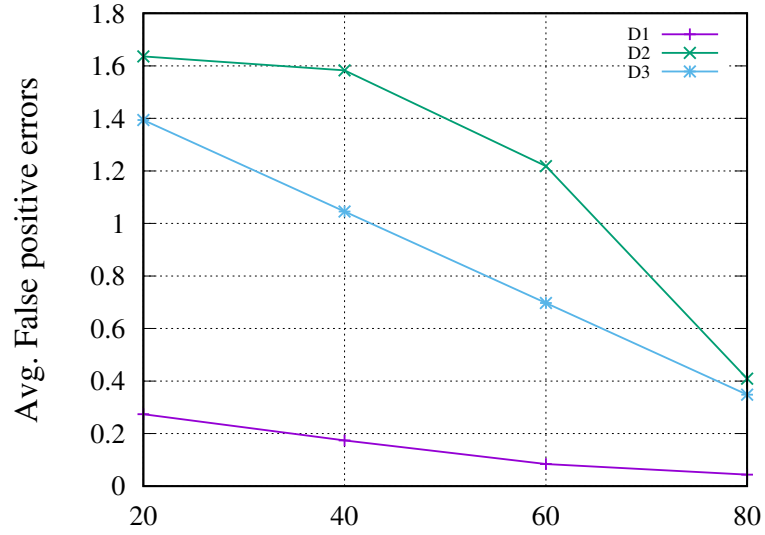

Figure 12. False-positive match analysis at different fault points.

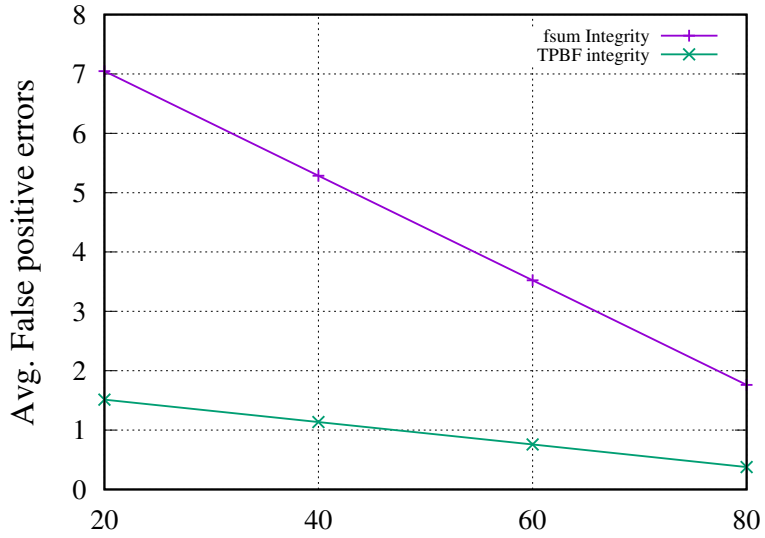

Figure 13. False-positive errors comparison of fsum and TPBF.

6. Conclusions

Object-based data transfer systems outperform existing data transfer tools with respect to data transfer rates. However, this method of data transfer, due to its out-of-order nature of object transfer, has a significant effect on the memory, storage, and computational overhead in ensuring the correctness of the transferred data. In the present work, we implemented an end-to-end data integrity verification framework using a two-phase Bloom filter to effectively handle data integrity errors by minimizing the memory, storage, and computational overhead. We analyzed and compared the impact of the TPBF method of data integrity on the overall data transfer performance and determined that the suggested framework had no apparent effect on data transfer performance for all kinds of workloads. Moreover, to evaluate the computational and memory overhead, we compared the data integrity overhead of the suggested framework with a state-of-the-art solution, *fsum*. The experimental results demonstrated that the proposed framework had fewer computational and memory requirements than the state-of-the-art solution. We also evaluated the storage overhead, and compared it with the state-of-the-art data integrity solution, *fsum*. The experimental results demonstrated 90% and 50% lower storage requirements than *fsum* for the D1 and D3 workloads, respectively. However, we observed similar storage requirements as fsum for the D2 workloads.

Although the proposed TPBF method of data integrity verification framework is very effective concerning computational, memory, and storage requirements, the probabilistic nature of the data structure leads to false-positive errors. Therefore, to assess the suggested framework's efficiency in decreasing false-positive errors, we created a simulation environment that generated faults at 20%, 40%, 60%, and 80% of data transfer points. The experimental outcomes revealed that the proposed TPBF data integrity verification framework outperformed the state-of-the-art solution, *fsum*, and was 80% more efficient in avoiding false-positive matches at all simulated fault points.

In summary, the proposed two-phase Bloom-filter-based data integrity framework complements existing object-based big data transfer systems with end-to-end data integrity support without negatively impacting data transfer performance.

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 and editing, P.K., P.H. and T.-S.C. All authors have read and agreed to the published version of the manuscript.

Funding: This research was supported by the MSIT (Ministry of Science and ICT), Korea, under the ITRC (Information Technology Research Center) support program (IITP-2021-0-02051) supervised by the IITP (Institute for Information and Communications Technology Planning and Evaluation) and the BK21 FOUR program of the National Research Foundation of Korea funded by the Ministry of Education (NRF5199991014091).

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. ORNL. Available online: https://www.ornl.gov/ (accessed on 20 April 2021).
2. CERN. Available online: https://home.cern/ (accessed on 20 April 2021).
3. LIGO. Available online: https://www.ligo.caltech.edu/ (accessed on 20 April 2021).
4. Data Never Sleeps 5.0. Available online: https://www.domo.com/learn/infographic/data-never-sleeps-5 (accessed on 25 May 2021).
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. Welch, B.; Unangst, M.; Abbasi, Z.; Gibson, G.; Mueller, B.; Small, J.; Zelenka, J.; Zhou, B. Scalable Performance of the Panasas Parallel File System. In Proceedings of the 6th USENIX Conference on File and Storage Technologies, FAST'08, San Jose, CA, USA, 26–29 February 2008; pp. 1–17.
8. Lofstead, J.; Zheng, F.; Liu, Q.; Klasky, S.; Oldfield, R.; Kordenbrock, T.; Schwan, K.; Wolf, M. Managing Variability in the IO Performance of Petascale Storage Systems. In Proceedings of the SC'10: 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, New Orleans, LA, USA, 13–19 November 2010; pp. 1–12. [CrossRef]
9. 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]
10. Kim, Y.; Atchley, S.; Vallée, G.R.; Shipman, G.M. *LADS: Optimizing Data Transfers using Layout-Aware Data Scheduling*; Technical Report ORNL/TM-2014/251; Oak Ridge National Laboratory: Oak Ridge, TN, USA, 2015.
11. 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]
12. Settlemyer, B.; Dobson, J.M.; Hodson, S.W.; Kuehn, J.A.; Poole, S.W.; Ruwart, T.M. A Technique for Moving Large Data Sets over High-Performance Long Distance Networks. In Proceedings of the IEEE Symposium on Massive Storage Systems and Technologies, MSST'11, Denver, CO, USA, 23–27 May 2011; pp. 1–6.
13. Stone, J.; Partridge, C. When the CRC and TCP checksum disagree. *ACM SIGCOMM Comput. Commun. Rev.* **2001**, *30*, 309–319. [CrossRef]

14. Meylan, A.; Cherubini, M.; Chapuis, B.; Humbert, M.; Bilogrevic, I.; Huguenin, K. A Study on the Use of Checksums for Integrity Verification of Web Downloads. *ACM Trans. Priv. Secur.* **2020**, *24*, 4. [CrossRef]
15. Hash Functions: CSRC. Available online: https://csrc.nist.gov/projects/hash-functions (accessed on 25 February 2022).
16. RFC 1321—The MD5 Message-Digest Algorithm. Available online: https://datatracker.ietf.org/doc/html/rfc1321 (accessed on 25 February 2022).
17. 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]
18. 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]
19. 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 City, UT, USA, 14 November 2016; pp. 55–60.
20. Lustre: A Scalable, High-Performance File System Cluster. Available online: https://cse.buffalo.edu/faculty/tkosar/cse710/papers/lustre-whitepaper.pdf (accessed on 25 February 2022).
21. Xie, B.; Chase, J.; Dillow, D.; Drokin, O.; Klasky, S.; Oral, S.; Podhorszki, N. Characterizing output bottlenecks in a supercomputer. In Proceedings of the SC'12: International Conference on High Performance Computing, Networking, Storage and Analysis, Salt Lake City, UT, USA, 11–15 November 2012; pp. 1–11. [CrossRef]
22. Schwan, P. Lustre: Building a File System for 1,000-node Clusters. In Proceedings of the Linux Symposium, Ottawa, ON, Canada, 23–26 July 2003; p. 9.
23. Introduction to InfiniBand. Available online: https://network.nvidia.com/sites/default/files/pdf/whitepapers/IB_Intro_WP_190.pdf (accessed on 25 February 2022).
24. Wu, J.; Wyckoff, P.; Panda, D. PVFS over InfiniBand: Design and performance evaluation. In Proceedings of the 2003 International Conference on Parallel Processing, Kaohsiung, Taiwan, 6–9 October 2003; pp. 125–132. [CrossRef]
25. Hanushevsky, A. BBCP. Available online: http://www.slac.stanford.edu/~abh/bbcp/ (accessed on 24 September 2021).
26. Allcock, W.; Bresnahan, J.; Kettimuthu, R.; Link, M.; Dumitrescu, C.; Raicu, I.; Foster, I. The Globus Striped GridFTP Framework and Server. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC'05, Seattle, WA, USA, 12–18 November 2005; pp. 54–64. [CrossRef]
27. Alliance, G. The Globus Toolkit. Available online: http://http://www.globus.org/toolkit/ (accessed on 24 September 2021).
28. Malensek, M.; Pallickara, S.; Pallickara, S. Alleviation of Disk I/O Contention in Virtualized Settings for Data-Intensive Computing. In Proceedings of the 2015 IEEE/ACM 2nd International Symposium on Big Data Computing (BDC), Limassol, Cyprus, 7–10 December 2015; pp. 1–10. [CrossRef]
29. Kim, Y.; Atchley, S.; Vallée, G.R.; Shipman, G.M. LADS: Optimizing Data Transfers using Layout-Aware Data Scheduling. In Proceedings of the 1st USENIX Conference on File and Storage Technologies, FAST'15, Santa Clara, CA, USA, 16–19 February 2015.
30. 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]
31. Bairavasundaram, L.N.; Goodson, G.; Schroeder, B.; Arpaci-Dusseau, A.C.; Arpaci-Dusseau, R.H. An Analysis of Data Corruption in the Storage Stack. In Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST 08), San Jose, CA, USA, 26–29 February 2008.
32. 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.
33. Lustre, ZFS, and Data Integrity. Available online: https://wiki.lustre.org/images/0/00/Tuesday_shpc-2009-zfs.pdf (accessed on 25 February 2022).
34. 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]
35. Improvements in Lustre Data Integrity—Opensfs. Available online: https://www.opensfs.org/wp-content/uploads/2011/11/Improvements-in-Lustre-Data-Integrity.pdf (accessed on 25 February 2022).
36. 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; Association for Computing Machinery: New York, NY, USA, 2005; pp. 26–36. [CrossRef]
37. Kumar, M.; Meena, J.; Singh, R.; Vardhan, M. Data outsourcing: A threat to confidentiality, integrity, and availability. In Proceedings of the 2015 International Conference on Green Computing and Internet of Things (ICGCIoT), Greater Noida, India, 8–10 October 2015; pp. 1496–1501. [CrossRef]
38. 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]
39. Sravan Kumar, R.; Saxena, A. Data integrity proofs in cloud storage. In Proceedings of the 2011 Third International Conference on Communication Systems and Networks (COMSNETS 2011), Bangalore, India, 4–8 January 2011; pp. 1–4. [CrossRef]

40. George, A.S.; Nargunam, A.S. Multi-Replica Integrity Verification in Cloud: A Review and A Comparative Study. In Proceedings of the 2021 International Conference on Communication, Control and Information Sciences (ICCISc), Idukki, India, 16–18 June 2021; Volume 1, pp. 1–5. [CrossRef]
41. 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]
42. Wang, H.; Zhang, J. Blockchain Based Data Integrity Verification for Large-Scale IoT Data. *IEEE Access* **2019**, *7*, 164996–165006. [CrossRef]
43. Ma, A.; Dragga, C.; Arpaci-Dusseau, A.C.; Arpaci-Dusseau, R.H. Ffsck: The Fast File System Checker. In Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST 13), San Jose, CA, USA, 12–15 February 2013; pp. 1–15.
44. Abu-Rayyan, L.; Hacid, H.; Leoncé, A. Towards an End-User Layer for Data Integrity. In Proceedings of the 2019 IEEE/WIC/ACM International Conference on Web Intelligence (WI), Thessaloniki, Greece, 14–17 October 2019; pp. 317–320.
45. Arasu, A.; Eguro, K.; Kaushik, R.; Kossmann, D.; Meng, P.; Pandey, V.; Ramamurthy, R. Concerto: A High Concurrency Key-Value Store with Integrity. In Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD'17, Chicago, IL, USA, 14–19 May 2017; Association for Computing Machinery: New York, NY, USA, 2017; pp. 251–266. [CrossRef]
46. 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]
47. 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]
48. 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]
49. Bloom, B.H. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* **1970**, *13*, 422–426. [CrossRef]
50. Broder, A.; Mitzenmacher, M. Survey: Network Applications of Bloom Filters: A Survey. *Internet Math.* **2003**, *1*, 485–509. [CrossRef]
51. Bloom Filter. Available online: https://en.wikipedia.org/wiki/Bloom_filter (accessed on 20 April 2021).
52. Kirsch, A.; Mitzenmacher, M. Less Hashing, Same Performance: Building a Better Bloom Filter. *Random Struct. Algorithms* **2008**, *33*, 187–218. [CrossRef]
53. 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.
54. 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]
55. Murmur Hash. Available online: https://en.wikipedia.org/wiki/MurmurHash (accessed on 25 May 2021).
56. Hash Functions. Available online: http://www.cse.yorku.ca/~oz/hash.html (accessed on 25 May 2021).
57. Tarkoma, S.; Rothenberg, C.E.; Lagerspetz, E. Theory and Practice of Bloom Filters for Distributed Systems. *IEEE Commun. Surv. Tutor.* **2012**, *14*, 131–155. [CrossRef]
58. Jiang, M.; Zhao, C.; Mo, Z.; Wen, J. An improved algorithm based on Bloom filter and its application in bar code recognition and processing. *EURASIP J. Image Video Process.* **2018**, *2018*, 139. [CrossRef]
59. George, A.; Mohr, R.; Simmons, J.; Oral, S. *Understanding Lustre Internals Second Edition*; Oak Ridge National Lab. (ORNL): Oak Ridge, TN, USA, 2021. [CrossRef]
60. Atlas. Available online: https://github.com/ORNL-TechInt/Atlas_File_Size_Data (accessed on 20 April 2021).