IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received December 15, 2020, accepted January 18, 2021, date of publication February 1, 2021, date of current version February 9, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3055731

DLFT: Data and Layout Aware Fault Tolerance Framework for Big Data Transfer Systems

PREETHIKA KASU[1], PRINCE HAMANDAWANA[1], AND TAE-SUN CHUNG[2]
[1]Department of Software and Computer Engineering, Ajou University, Suwon 16499, South Korea
[2]Department of Artificial Intelligence, Ajou University, Suwon 16499, South Korea

Corresponding author: Tae-Sun Chung (tschung@ajou.ac.kr)

This work was supported in part by the Institute of Information & Communications Technology Planning & Evaluation (IITP) Grant funded by the Korean Government (MSIT) under Grant 2020-0-01592, and in part by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education under Grant 2019R1F1A1058548.

ABSTRACT Various scientific research organizations generate several petabytes of data per year through computational science simulations. These data are often shared by geographically distributed data centers for data analysis. One of the major challenges in distributed environments is failure; hardware, network, and software might fail at any instant. Thus, high-speed and fault tolerant data transfer frameworks are vital for transferring such large data efficiently between the data centers. In this study, we proposed a bloom filter-based data aware probabilistic fault tolerance (DAFT) mechanism that can handle such failures. We also proposed a data and layout aware mechanism for fault tolerance (DLFT) to effectively handle the false positive matches of DAFT. We evaluated the data transfer and recovery time overheads of the proposed fault tolerance mechanisms on the overall data transfer performance. The experimental results demonstrated that the DAFT and DLFT mechanisms exhibit a maximum of 10% and a minimum of 2% recovery time overhead at 80% and 20% fault points respectively. However, we observed minimum to negligible overhead with respect to the overall data transfer rate.

INDEX TERMS Big data, geo-distributed data centers, fault tolerance, bloom filter, parallel file system.

I. INTRODUCTION

Modern scientific experimental facilities such as CERN [1], LIGO [2], and ORNL [3] generate terabytes to petabytes of data every day. Additionally, every single entity in today's world has some digital component or counterpart, which is capable of generating data. Devices such as mobile phones, (security) cameras, smart home gadgets, and telemetry devices continuously generate data or digital content. Internet users generate more than 2.5 quintillion bytes of data per day geographically, and this number has been accelerated by modern technologies such as IoT, AI, and machine learning.

To provide a better quality of service to customers in terms of the response time and availability based on the location, service providers distribute their data centers geographically worldwide. This results in a significant increase in the demand for data transfer among data centers in such geo-distributed data center systems. However, how can the available inter-data center network bandwidth be fully utilized to satisfy real-time computational requirements?

The associate editor coordinating the review of this manuscript and approving it for publication was Xiaowen Chu.

Although networks are achieving terabit speed and storage capacities are being expanded to exabytes, there is an evident mismatch between the speed of network and storage. This gives rise to a major challenge in achieving higher end-to-end data transfer rates. To reduce the impedance mismatch between the network and storage as well as improve the scalability, data centers deploy parallel file systems (PFSes).

PFSes use dedicated servers to service metadata and I/O operations. To improve throughput, PFSes scale up the number of I/O servers to achieve higher performance. Typically, large-scale storage systems use tens to hundreds of I/O servers equipped with tens to hundreds of disks. Storage systems share resources between different clients. This enables clients to compete for the same resource. As the contention for these resources increases, there can be a serious gap between the expected and observed I/O performance; additionally, some servers or their disks might become overloaded. This type of load imbalance is a serious problem in PFSes [4]. To answer these problems, researchers have proposed efficient bulk data transfer frameworks [5] that can avoid temporarily congested servers during data transfer [6].

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

The traditional disk file system uses an inode data structure to retrieve file information such as the storage location of file data. Lustre [7], a PFS, also uses the inode data structure. However, the inodes on Metadata Target (MDT) points to one or more Object Storage Target (OST[1]) objects. These objects are implemented as files on the OSTs. When a client opens a file, the file open operation transfers a set of object identifiers and their layout from the Metadata Servers (MDS) to the client; this enables the client to directly interact with the object storage server (OSS) node where the object is stored and perform I/O in parallel across all OST objects in the file without further communication with the MDS.

One of the major challenges in the distributed environment is fault handling; hardware, software, and network might fail at any point of time. It is very costly in terms of time and additional network traffic to retransmit the whole data from the beginning while transmitting several terabytes of data. Distributed data transfer tools need to handle faults efficiently to reduce retransmission overhead upon recovery.

Big data transfer frameworks exploit the storage layout of files and enhance the data transfer rates by transferring the file objects in parallel. Owing to this object nature of data as well as parallel processing, the data transfer frameworks can transfer the objects of the same file from the source to sink in an out-of-order manner. If there is any fault in the end-to end path, this out-of-order nature of data transfer will lead to data corruption issues. This type of behavior forces the retransmission of the entire file data (objects) after recovering from the fault, thereby causing unnecessary congestion.

Owing to this out-of-order nature of object transmission, checkpoint-based logging [8] or logging the index of the last transferred object is not enough for resuming the transfers after the fault. Another approach is maintaining a log of all objects that were successfully sent and written at the sink end PFS. This type of logging mechanism [9] will affect the overall space occupied by the log files. Additionally, the time consumed for logging successful objects while transferring as well as that for retrieving the successfully completed objects after fault directly affect the overall performance of data transfer.

In this study, our main objective is to design fault tolerance framework to minimize the time, space, and retrieval overhead while not negatively impacting the data transfer performance. This paper makes the following contributions.

- Bloom filter-based data aware fault tolerance mechanism (DAFT) for efficiently managing the faults with out-of-order nature of the object transmission.
- Data and layout aware fault tolerance mechanism (DLFT) for efficiently handling the false positive object membership matches while recovering from the fault.
- We have analyzed the overhead of DAFT and DLFT frameworks with respect to data transfer performance and space overhead. For evaluating our implementation, we have used a Lustre filesystem which communicates over an InfiniBand (IB) network. From our evaluation results, we have observed negligible overhead (< 1%) with respect to the data transfer time and space (≈500 KB (KiloBytes)). However, we observed a recovery time overhead of 2-10% according to the fault point of data transfer.

[1] An OST manages a single storage device, and multiple OSTs are managed by the OSS

TABLE 1. Big Data Applications.

Sector	Application
Banking and Securities	Monitoring financial market, Trade, and Risk analytics.
Communications media and Entertainment	Create content for different target audiences, Recommend content on demand, and Measure content performance.
Healthcare	To aid in short-term public health monitoring and long-term epidemiological research programs.
Education	Learning and Management System, Customized and Dynamic Learning Programs, and Career Prediction.
Social Media	To analyze the customer behavior pattern.
Weather	Weather forecasting, Study global warming, Understanding the patterns of natural disasters, and Crises analysis.
Transportation	Route planning, Congestion management, and traffic control.
Government Sector	Welfare Schemes, Cyber Security, fraud detection, and environmental protection.

The remainder of this paper is organized as follows. Section II describes the background followed by the motivation of our work. Section III reviews the design and implementation aspects of the bloom filter. Section IV reviews the data aware and data and layout aware fault tolerance design and implementation details. The experimental results are presented in Section V. We conclude our paper in Section VI.

II. BACKGROUND AND MOTIVATION

A. BACKGROUND

1) BIG DATA APPLICATIONS

Big Data has become a game-changer in most, if not all, types of modern industries over the last few years. More and more organizations, both big and small, are leveraging from the benefits provided by big data applications. Table 1 summarizes the applications of big data in different sectors.

2) FAULT TOLERANCE AND DATA TRANSFER FRAMEWORKS

Reliability and high-performance are the major challenges of big data transfer frameworks while moving large volumes of data between geographically distributed data centers. To achieve high reliability the fault tolerance must be accomplished. To cope with the problem of fault tolerance

in cloud computing environments, researchers have proposed different fault tolerance methods.

Fault Tolerance

Adaptive framework for reliable cloud computing environment [10] has developed an adaptive model/framework to handle the faults in the cloud environment. This adaptive model enable the fault tolerance support using both check pointing and replication techniques. The proposed framework implement algorithms for deciding/choosing fault tolerance (FT) method, i.e. replication or check pointing approaches. The proposed framework has been evaluated on the basis of throughput, overheads and availability. As a result, this framework shows an improved performance in the cloud environments as compared to the existing algorithms. However, this type of fault tolerance model/framework is not suitable for the data transfer frameworks that focus on transferring objects instead of the entire file sequentially.

JCSR (Joint Checkpoint Scheduling and Routing) [11] provide reliability optimization in the cloud environment. A peer-to-peer check pointing method has been used to enable fault tolerance. As this method also uses checkpoint record, this framework is also not suitable for object based big data transfer frameworks.

Fault-tolerant workflow scheduling (ICFWS) algorithm [12] for cloud systems combine resubmission and replication strategies together to play their respective advantages for fault tolerance while trying to meet the soft deadline of workflow. Though, this algorithm outperforms some well-known fault tolerance methods in cloud environment, this method will have a negative impact on the overall data transfer time due to resubmission and replication strategies. So, this framework is also not suitable for systems with high performance requirements.

Owing to the object nature of the data transfer and high performance requirements, it is not possible to use checkpoint, resubmission and replication strategies as fault tolerance methods in our framework. In this study, we have proposed novel bloom filter based probabilistic fault tolerance mechanisms to minimize the time, space and retrieval overhead while not negatively impacting the data transfer performance.

Data Transfer Frameworks

To address the challenges encountered in transferring data between the data centers, researchers have proposed different big data transfer frameworks.

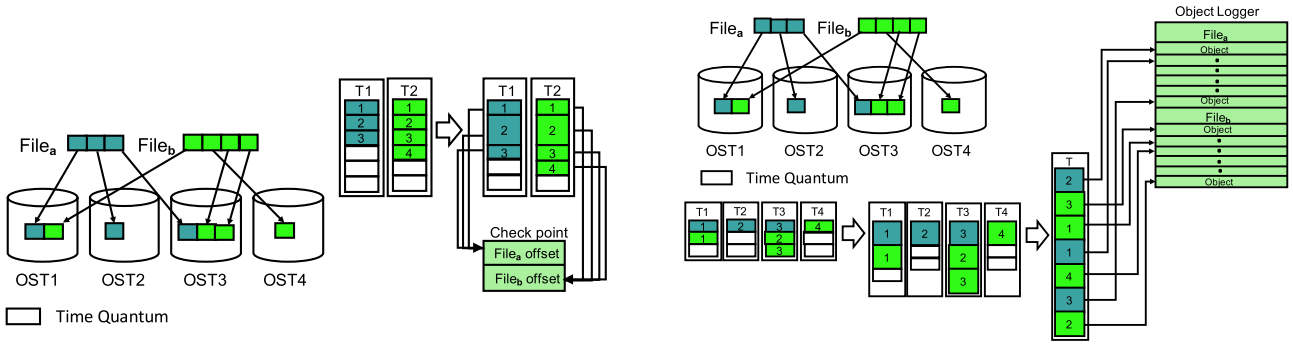
GridFTP [13], [14] is a well-known and robust protocol for fast data transfer on the grid. It is an extension of the File Transfer Protocol and defines a general-purpose mechanism for secure, reliable, and high-performance data movement. This framework utilizes the parallel data transfer mechanism by employing multiple TCP (Transmission Control Protocol) streams to aggregate the overall bandwidth. This framework also utilizes the striping feature to support multi-host to multi-host data transfer. Another important aspect of GridFTP is its ability to recover from failed transfers by salvaging the partial transfers and resuming mid-file. While transferring data, the Reliable File Transfer (RFT) service of GridFTP provides an interface to write the restart markers to the database to ensure that it can survive local faults. Upon recovery from a fault, the GridFTP server sends these restart markers to the client and the client restarts the transfer from based on these markers. Because GridFTP transfers the logical file data sequentially, the transfers can be recovered from the markers. However, our work considers transferring the data as objects based on their layout; hence, the marker-based recovery mechanism cannot be implemented in our framework.

BBCP [15] is an alternative for GridFTP for transferring large amounts of data. This tool can break a transfer into multiple simultaneous streams, thereby transferring data significantly faster than single-streaming utilities such as SCP (Secure Copy Protocol) and SFTP (SSH File Transfer Protocol). Because BBCP transfer the entire file data sequentially, the checkpoint-based fault recovery mechanism is used with this framework. However, this type of recovery mechanism is not suitable for the data transfer frameworks that focus on transferring objects instead of the entire file sequentially.

XDD [16] was designed to provide the software infrastructure required to move large datasets with high levels of performance and reliability. This tool provides several options to facilitate better file transfer, such as impedance matching, configurable device access schemes, threads, and I/O scheduling policies. 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 destination. However, recovering from faults, if any, is not considered in this tool. Hence, the entire dataset needs to be retransmitted after recovering from faults.

RAMSYS (Resource-Aware Asynchronous Data Transfer) [17], a resource-aware high-speed data transfer software, utilizes a multistage end-to-end data transfer pipeline, where each stage is fully resource-driven and implements a flexible number of components using predefined functions, such as storage I/O, network communication, and request handling. RAMSYS relies on the asynchronous paradigm to maximize the concurrency of components, thereby offering improved scalability and resource utilization in modern multi-core systems. However, this framework does not consider error recovery upon faults.

All the above-mentioned data transfer frameworks focus on transferring large data in a fast and secure manner from source to destination over the network. Also, as these frameworks transfer the logical file data sequentially, it is possible to resume from the failed transfers using a checkpoint based restart marker or an offset record. However, our work focuses on an entirely different scenario from the prior fault tolerance studies. In this work, we aim to support fault tolerance functionality when the workload is transferred as objects rather than files. Due to the object nature of the data transfer, it is possible to transfer one logical file's objects in random order.



(a) Checkpoint record

(b) Object index logging

FIGURE 1. Checkpoint and object index logging.

Hence, checkpoint based restart marker or an offset record based fault tolerance mechanisms are not sufficient to resume from the failed transfers. Therefore, in this study, we propose data aware and data and layout aware fault tolerance mechanisms to handle fault tolerance in object-based big data transfer frameworks.

B. MOTIVATION

Traditional big data transfer tools [13]–[15] consider workloads in terms of the logical files, irrespective of their physical distribution, of the file data in the storage. Owing to this, data transfer tools read or write the file data sequentially until the entire file is processed. During the transfer, the storage server consumes more time to service new I/O requests if the number of I/O requests exceeds the storage server capabilities. This type of storage congestion negatively impacts the overall data transfer rate.

A PFS is a type of distributed file system that distributes the file data across multiple storage servers and provides concurrent access to such file data. Owing to this, PFSes historically have targeted high-performance computing (HPC) environments that require access to large files, massive quantities of data, simultaneous access from multiple computer servers, or multiple tasks of a parallel application. In this study, we used the open-source file system Lustre as our PFS.

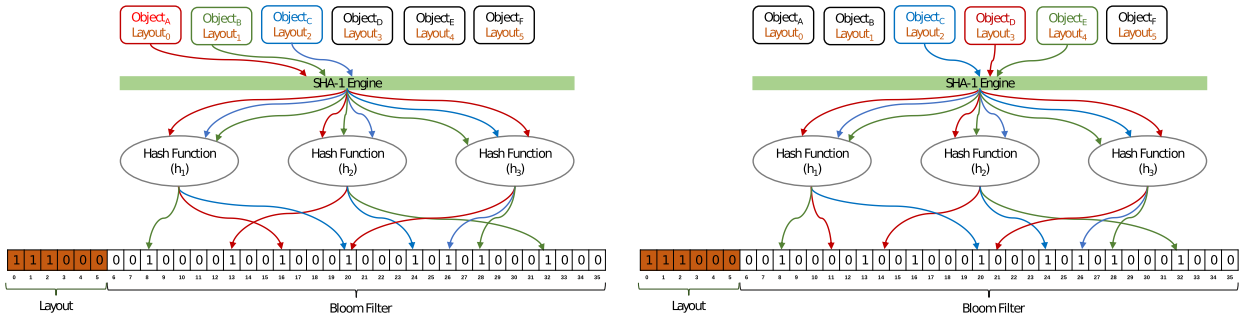
Fig. 1(a) and Fig. 1(b) depict how the Lustre file system stripes File$_a$ and File$_b$ across multiple OSTs. Considering the distributed nature of logical file data, multiple I/O threads can be assigned to process the data transfer and improve the big data transfer performance. Employing multiple I/O threads without any knowledge about the physical distribution of files might still cause disk contention issues as multiple threads compete for the same OSS or OST. This contention degrades the data transfer performance of the application. These types of resource contention issues are addressed by [6] and [18], which employ layout aware scheduling algorithms to process data transfers. Owing to these layout aware scheduling algorithms, objects of different logical files might be transferred in parallel. Although this type of mechanism significantly improves the data transfer performance, complex fault tolerance mechanisms must be devised to recover data transfers upon fault.

Big data transfer tools should handle faults efficiently to reduce the retransmission overhead upon recovery. Because the existing big data transfer tools transfer the file data sequentially, simple fault tolerance mechanisms such as checkpoint records can be adopted. As shown in Fig. 1(a), all objects of File$_a$ and File$_b$ are transferred in sequence. Thread T_1 transfers the first object of File$_a$ and records the file offset information. After completing the second object, the thread overwrites the checkpoint record with the updated file offset information. This process is repeated for all files in the dataset. During this process, if the transfer is resumed from a fault, then the transfer tool checks for the existence of checkpoint record for the target file; if yes, it starts transferring the objects from the offset found in the checkpoint record.

Data transfer frameworks, which target high data transfer performance, employ multiple I/O threads by exploiting the physical layout of files. Thus, objects of one logical file are transferred out-of-order. It can be observed from Fig. 1(b) that the second object of File$_a$ is transferred before the first object. Similarly, we can also observe the out-of-order object transfer for File$_b$. Because objects are transferred in an out of order manner, it is not possible to recover the completed object information by a checkpoint record, as shown in Fig. 1(a). Instead, the information about all objects of all logical files, which are successfully transferred, as shown in Fig. 1(b), should be recorded appropriately.

The amount of space occupied by log files for maintaining information regarding the successfully transferred objects corresponding to the logical files in the dataset is one of the major challenges. In previous research [19], we proposed fault tolerance mechanisms to address the space, computation, and recovery overheads of the object logging mechanisms on the data transfer rate. Among the proposed mechanisms, we concluded that the universal logger mechanism combined with bit binary methods (Bit8 and Bit64) has the minimum overhead with respect to space and recovery time. This study is an extension of our previous research [19] where we answer the following questions.

- How can the data transfer performance and recovery time overhead of the proposed bloom filter-based probabilistic fault tolerance mechanisms be minimized?



(a) Before Fault

(b) After Fault

FIGURE 2. An illustrative example of bloom filter.

- How can the space occupied by the fault tolerance mechanisms be minimized using bloom filter-based probabilistic data structures?
- How can the false positive matches of bloom filter-based probabilistic data structures be reduced?

III. BLOOM FILTER DESIGN

In this section, we describe the design and implementation aspects of the bloom filter.

A. BLOOM FILTER DATA STRUCTURE

A bloom filter is a simple space-efficient probabilistic data structure for representing a set to support constant-time membership queries [20], [21]. A bloom filter representing a set of n elements consists of an array of m bits that are initially set to 0. The bloom filter uses k independent hash functions $h_1, h_2 \ldots, h_k$ to generate k hashed positions in the range $1, \ldots, m$. By inserting an object, k out of m bits are set to 1. By querying for an object, k hashed positions are tested against 1, and if any one of the k bits is 0, it implies that the object is not in the set. If all k bits are set to 1, we assume that the object is in the set, and hence, the bloom filter may yield false-positive errors.

The probability of an element not present in a set or the false positive probability can be estimated in a straightforward manner [22]. After all objects of a data set are hashed into the bloom filter, the probability that a specific bit is still 0 for k number of hash functions and large m can be defined by:

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

Therefore, the probability that it is 1 is

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

To test the membership of an object that is not in the set, the probability of the false-positive error can be defined by

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

The false-positive error probability of the bloom filter depends on the number of objects in the set (n), total bloom filter size (m), and the number of hash functions (k). In practice, k must be an integer; a smaller, suboptimal k is preferred because it reduces the number of hash functions to be computed. For a given m and n, the number of hash functions required to minimize the false-positive errors is

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

B. HASH FUNCTION OPTIMIZATION

Hash functions are the core operations of bloom filters that require multiple independent hash functions for generating a bloom filter. Appropriately designed hash functions, such as MD5 and SHA-1, are computationally expensive. To optimize the computation overhead of hash functions, state-of-the-art techniques have attempted to generate multiple independent hash values with only one or two hash functions [23]–[25]. In our experiments, we used two hash functions, i.e., murmur [26] and DJB2 on the SHA-1 hashed input data. These two hash functions are used to generate additional hash functions. Specifically, the k hash functions are calculated as Eq.(5):

$$g_i(x) = h_1(x) + i * h_2(x) \mod m \tag{5}$$

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

C. ILLUSTRATION OF BLOOM FILTER

An illustrative example of bloom filter used in DLFT is as shown in the Fig.2. In this illustration, we have considered the number of objects in the set $n = 6$, number of hash functions $k = 3$, and bloom filter size as $m = 30$. For the purpose of illustration, we have depicted the case where objects $\{A, B, C\}$ were transferred to the sink end PFS before the fault and to illustrate success, fail, and false positive match scenarios, we considered the objects $\{C, D, E\}$ upon recovery.

As shown in Fig.2, bloom filter array is divided into two parts; layout and bloom filter. Layout segment of the bloom filter array is populated based on the object layout information upon successful object transfer. Whereas, bloom filter segment of the array is used to randomly map the objects into k positions by employing k independent hash functions. Upon initiating the data transfer, all of the $(n + m)$ bits of the boom filter array are initialized to 0.

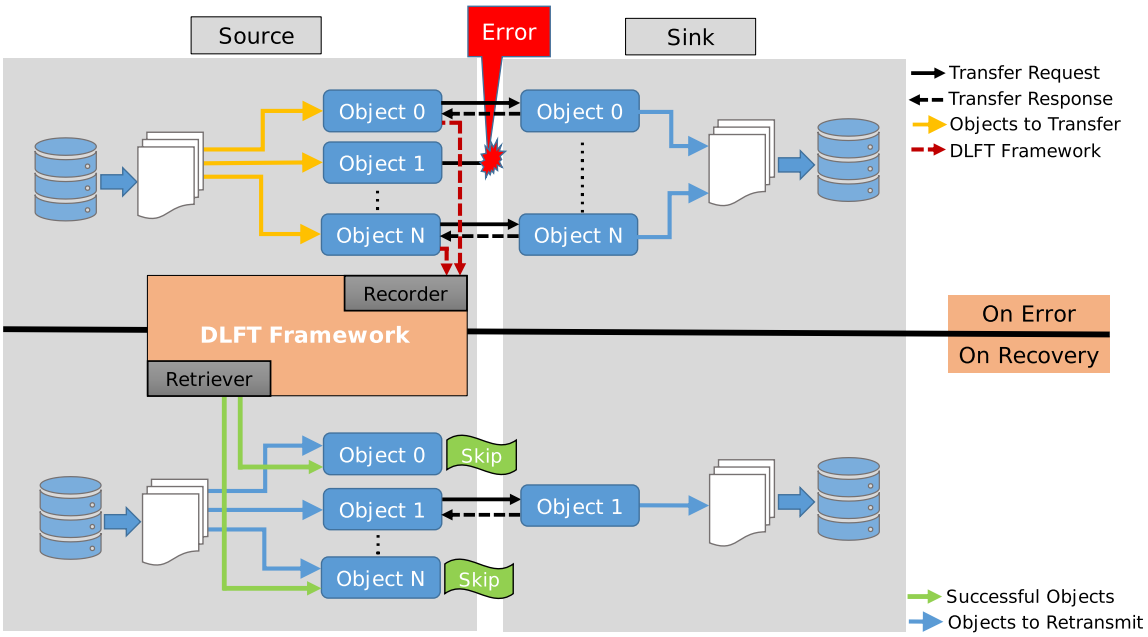


FIGURE 3. Fault tolerance system architecture.

1) BEFORE FAULT

Fig.2(a) depicts before fault scenario. SHA-1 engine is used to compute the block hash on the object data to uniquely represent the object. In this example, we have considered objects $\{A, B, C\}$ were inserted to bloom filter. Hash functions $\{h_1, h_2, h_3\}$ are employed on hashed object data to randomly map the objects into k positions. Employing $\{h_1, h_2, h_3\}$ hash fucntions on hashed $Object_A$ data, bits $\{13, 16, \text{and } 20\}$ of the bloom filter array are set to 1. Also, as the layout of $Object_A$ is *0*, bit 0 of the bloom filter array is set to 1. Similarly, for $Object_B$ and $Object_C$, bits $\{1, 8, 28, 32\}$ and $\{2, 20, 24, 26\}$ are set to 1 respectively.

2) AFTER FAULT

Fig.2(b) depicts after fault scenario. In this example, we have considered objects $\{C, D, E\}$ for querying their membership with the bloom filter. Object is assumed to be the member if all the k bits along with its layout bit is set in the bloom filter array. For $Object_C$, the resultant hash positions $\{20, 24, \text{and } 26\}$ along with its layout bit $\{2\}$ is set, thus the bloom filter return "Positive" for the query. The membership query of $Object_D$ returns "Negative" since the bit at position $\{11\}$ is not set. Though, the bits at positions $\{8, 28 \text{ and } 32\}$ are all set, the membership query of $Object_E$ would return "Negative" as the layout information at position $\{4\}$ is 0. Without the layout information, the membership query of $Object_E$ would return "False Positive" as the bits at positions $\{8, 28 \text{ and } 32\}$ are all set. Thus, using layout information in conjunction with the bloom filter, we have avoided the false positive matches of the bloom filter.

IV. DESIGN AND IMPLEMENTATION

In this section, we describe the design and implementation aspects of proposed fault tolerance mechanisms. First, we describe fault tolerant big data transfer system architecture for handling software, hardware, or common communication errors using the bloom filter-based probabilistic data structure. Then, we focus on the design and implementation details of DAFT. Next, we discuss the design and implementation aspects of DLFT for avoiding the false-positive matches of the bloom filter using the object layout information. Finally, we conclude by analyzing object logging based fault tolerance, DAFT and DLFT fault tolerance mechanisms.

A. SYSTEM ARCHITECTURE

Fig. 3 depicts the proposed fault tolerant big data transfer system architecture. On initiating data transfer, source end of the data trasfer tool prepares the list of the objects to be transferred to the sink end and initiates the transfer. Sink end acknowledges the source end for the objects that are sucessfully written to the sink end PFS. Upon receiving the acknowledgement from sink end, DLFT framework recorder component records the successfully completed objects information using the proposed probabilistic bloom filter-based fault tolerance mechanism.

If a transfer is resumed from a previous failed transfer, then the source end retrieves the completed object information using retriever component of the DLFT framework and exclude those objects from the list of the objects to be transferred to the sink end. For the remianing objects, source end initiates the transfer. This process is repeated until all objects of the dataset are successfully transferred to the sink endpoint.

B. DAFT (DATA AWARE FAULT TOLERANCE)

The data aware fault tolerance system is implemented using one *master* and communication (*comm*) threads, along with a configurable number of *I/O* and bloom filter (*B/F*) threads. The *master* thread schedules the transfer of file objects and the *I/O* threads read or write the object data from or to

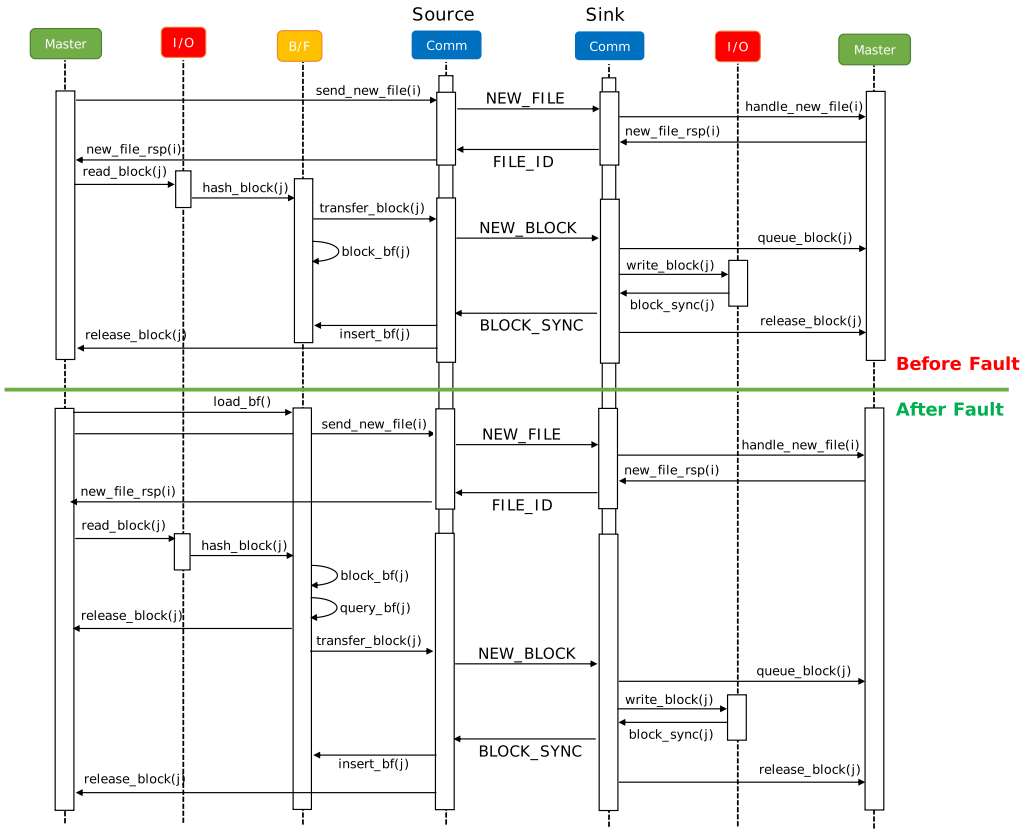


FIGURE 4. Data aware fault tolerance sequence diagram.

the PFS. The *B/F* thread hashes the object data using cryptographic hash functions to uniquely represent the object data, and computes the k hash positions of the m-*bit* bloom filter array on the hashed object data. The *comm* thread manages the communication between the source and sink endpoints. The *master*, *I/O*, and *B/F* threads get blocked while waiting for a resource; however, the *comm* thread continuously handles the communication between the source and sink endpoints.

Fig. 4 depicts the data transfer sequence between the source and sink endpoints. Upon initiating the data transfer, the source and sink endpoints initialize the threads required for communication, along with the required locks, wait queues, and work queues, and allocate buffers for the data transfer. As shown in Fig. 4, the source *master* thread generates a NEW_FILE request for each file in the target dataset, and enqueues the request in the work queue of the *comm* thread. The source *comm* thread dequeues the request and transfers it to the sink. At the sink end, upon receiving the NEW_FILE request, the *comm* thread enqueues the request to the work queue of the *master* thread and wakes it up. Based on the target file information in the request, the sink *master* thread opens the file and adds the file descriptor to the FILE_ID request and enqueues the request on the work queue of the *comm* thread. The sink-end *comm* thread dequeues the request and sends it to the source. On receiving the FILE_ID request, the source *comm* thread enqueues the request on the wait queue of the source *master* thread. Upon receiving the FILE_ID request, the source-end *master* thread splits the file according to the object size, generates a NEW_BLOCK request, and enqueues the request on the wait queue of the *I/O* thread. Upon receiving the NEW_BLOCK request, the *I/O* thread reserves the buffer and determines the OST [27], [28] to be used for reading the object data, and issues pread() to read the object data into the buffer reserved for communication. After completing the read operation, the *I/O* thread enqueues the NEW_BLOCK request on the wait queue of the *B/F* thread.

- **Before Fault:** If an object is scheduled to be transferred for the first time, the source *B/F* thread deques the request and computes the block hash on the object data and enqueues the request on the work queue of the *comm* thread. After enqueuing the request, the *B/F* thread progresses to compute the k hash positions of the m-*bit* bloom filter array on the hashed object data.
- **After Fault:** If a transfer is resumed from a previous partial transfer, then the source *B/F* thread computes the k positions of the m-*bit* bloom filter array based on the hashed object data. According to the computed k positions, the source *B/F* thread queries the object membership with the bloom filter. If any bit at these k positions is 0, then the *B/F* thread determines that the object was not transferred previously, and schedules the transfer by enqueuing a NEW_BLOCK request on the

work queue of the *comm* thread. If all bits at these k positions are set, then the *B/F* thread considers that block to be complete and releases the block resources.

The *comm* thread dequeues the NEW_BLOCK request and transfers it to the sink. On receiving the NEW_BLOCK request at the sink end, the *comm* thread attempts to reserve the buffer. If the *comm* thread acquires the buffer successfully, it initiates a read operation. If it fails to acquire the buffer, it enqueues the request on the work queue of the *master* thread. The *master* thread waits until a buffer is available, and then enqueues the request back on the work queue of the *comm* thread, which issues the read operation. Upon the successful completion of the read operation, the sink *comm* thread enqueues the request on the wait queue of the *I/O* thread. The *I/O* thread dequeues the request and determines the appropriate OST based on the object offset and issues a pwrite() system call to write the object data to the corresponding OST. Upon the successful completion of the write operation, the *I/O* thread enqueues a BLOCK_SYNC request on the work queue of the *comm* thread. The sink-end *comm* thread dequeues the BLOCK_SYNC request and notifies the source endpoint, and releases the resources allocated for the block transfer. Upon receiving BLOCK_SYNC request, source-end *comm* thread dequeues the BLOCK_SYNC request and enqueues the request on the wait queue of the *B/F* thread and releases the resources allocated for the block transfer. The *B/F* thread updates the bloom filter by setting the previously computed k positions to 1. This process is repeated until all objects of the dataset are successfully transferred to the sink endpoint.

While the proposed data-aware fault tolerance mechanism with bloom filters has a substantial space advantage over the existing fault tolerance mechanisms, this method result in false-positive object membership errors owing to its probabilistic nature of detecting the object membership. To avoid false-positive object membership matches of the bloom filter, we proposed a DLFT framework, which has been described in IV-C.

C. DLFT (DATA AND LAYOUT AWARE FAULT TOLERANCE)

The main objective of the bloom filter-based fault tolerance mechanism is to build a space-efficient data structure for retrieving the objects that were successfully transferred to the sink end. While maximizing the space efficiency, the bloom filter-based fault tolerance mechanism sacrifices the correctness. Owing to the false-positive nature of the bloom filter data structure, some object membership queries result in errors. Hence, some blocks that are not transferred to the sink end are falsely considered to be transferred. This results in data corruption, and thus, the transferred data would be unusable for further analysis.

A bloom filter is a bit vector (B) of m-bits with k independent hash functions ($h_1,\ldots, h_k$) that maps each element in the dataset ($S = \{x_1,\ldots, x_n\}$) to $R_m = \{0,1,\ldots,m\text{-}1\}$. We assume that each hash function, h_k, uniformly maps each object in the dataset to a random number over the range R_m with equal probability. Initially, all m-bits of bit vector B are 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 if 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 bit vector B are set to 1 by other objects due to hash collisions, then it causes false positive errors.

To avoid this type of false-positive object membership queries of the bloom filter, the object layout information (n-bits) is prepended in the bloom filter as additional information of the object. Consequently, the total size of B is increased to $(n+m)$-bits, and initially, these $(n+m)$-bits are initialized to "0".

- **Insert.** For each object $x_i \in S$, compute $h_1(x_i),\ldots, h_k(x_i)$ and set $B[n+h_1(x_i)] = B[n+h_2(x_i)] = \ldots. = B[n+h_k(x_i)] = 1$, along with $B[i] = 1$.
- **Query.** To check if 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.

The data transfer sequence between the source and sink endpoints for DLFT is illustrated in Fig. 5. The resource initialization, NEW_FILE, FILE_ID, and NEW_BLOCK request processing are similar to those in the DAFT system, as described in IV-B.

- **Before Fault:** Upon scheduling an object transfer, the source *B/F* thread computes the hash on the object data and enqueues the request on the work queue of the *comm* thread. After enqueuing the request, the *B/F* thread progresses to compute the k hashed positions of the *(n+m)-bit* bloom filter array on the hashed object data. The k independent hash functions, $h_1,\ldots, h_k$, take an initial value *"n"* and compute the k hash positions in the range $\{n,n+1,\ldots,n+m-1\}$; then, the bloom filter corresponding to that logical file is composed.
- **After Fault:** If a transfer is resumed from a previous partial transfer, then the source *B/F* thread computes the k positions of the *(n+m)-bit* bloom filter array on the hashed object data. Based on the computed k hashed positions, the source *B/F* thread queries the object membership with the bloom filter.
  - If any bit at these k positions in bloom filter is 0, then the *B/F* thread determines that the object was not transferred previously and schedules the transfer by enqueuing the NEW_BLOCK request on the work queue of the *comm* thread.
  - If all bits at these k positions as well as the block layout bit in the bloom filter are set, then the *B/F* thread considers that block to be complete and releases the resources allocated for block transfer.

After processing the NEW_BLOCK request at the sink end, the *I/O* thread enqueues the BLOCK_SYNC request

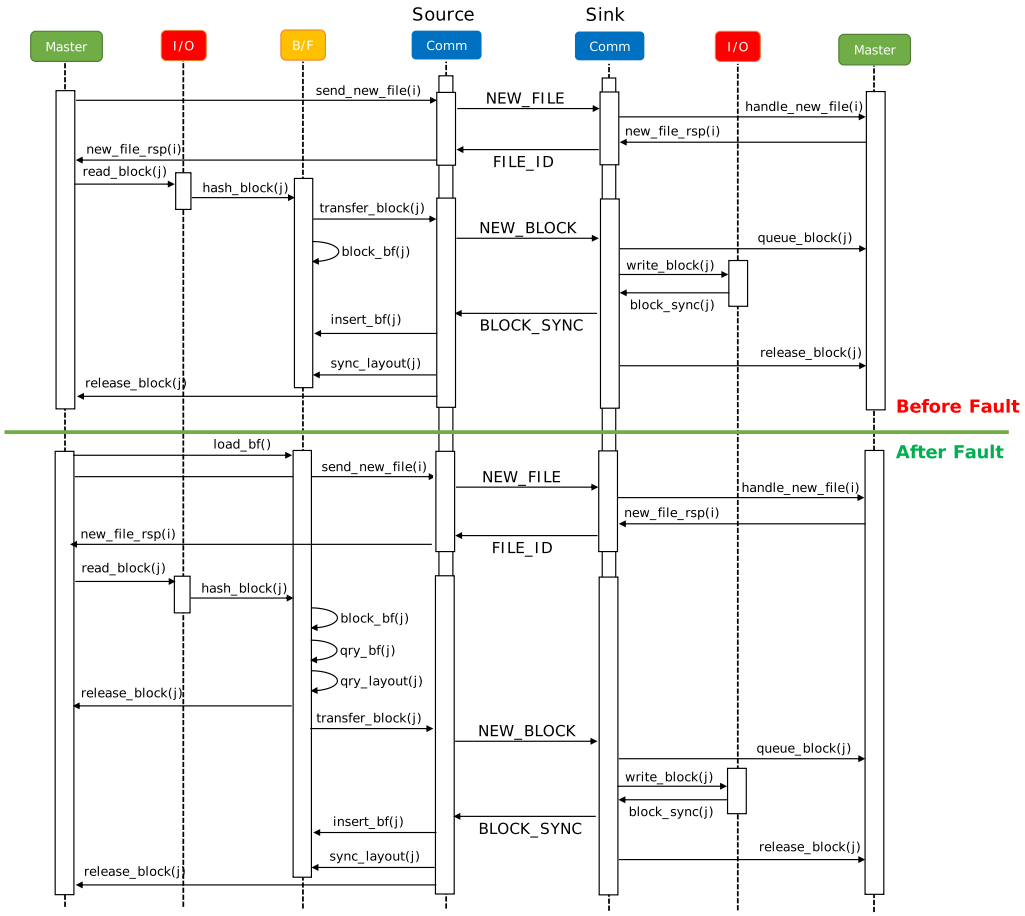


FIGURE 5. Data and layout aware fault tolerance sequence diagram.

on the work queue of the *comm* thread. The sink-end *comm* thread dequeues the BLOCK_SYNC request and notifies the source endpoint. Upon receiving the BLOCK_SYNC request, the source-end *comm* thread enqueues the request with the wait queue of the *B/F* thread, which synchronizes the block layout information along with the previously computed k hash positions with the bloom filter; it also releases the resources allocated for the block transfer. This process is repeated until all objects of the dataset are successfully transferred to the sink endpoint.

D. OBJECT LOGGING BASED FAULT TOLERANCE VS DAFT VS DLFT

In our previous research [19], we have proposed novel object logging based fault tolerance (FT) mechanisms for efficiently handling the faults when the workload is transferred as objects rather than files. In this section, we analyze object logging FT, DAFT and DLFT fault tolerance mechanisms.

In the object logging based FT mechanisms, objects which are successfully written to the sink PFS are considered as successful and log the information of those objects in the FT log file. Upon successful completion of all the objects of one logical file, the log information corresponding to that file will be erased. If there is any fault during the transfer, the proposed mechanisms search for the completed objects and schedule only those objects which were not transferred previously. From our experimental results, we have concluded that, object logging based FT mechanisms efficiently handle the faults without negatively impacting the data transfer performance. However, owing to the object layout logging nature of the FT, updates to the data is gone unnoticed and end up having old data at the sink end. In this work, our proposed DAFT and DLFT fault tolerance mechanisms efficiently handle faults as well as updates to the data.

DAFT method of FT address the issues observed with object logging based fault tolerance mechanims using bloom filter based probabilistic data structure. This method of fault tolerance considers the object data rather than the object layout for identifying the transferred objects. DAFT uniquely represent the object data using *k-bits*. If an object is scheduled to be transferred for the first time, the source end of the transfer computes the block hash on the object data and computes the k hash positions of the *m-bit* bloom filter array on the hashed object data. Upon successful object transfer, DAFT method of FT updates the bloom filter corresponding to that logical file (f_{bloom}) with the computed k hash positions. If there is any fault during the transfer, DAFT method of fault tolerance determines if the object is already transferred or not by computing the k hash positions, on the hashed object data, and comparing with the f_{bloom}. If the object is

not transferred previously, schedules the same for the transfer. This method of fault tolerance mechanism has a substantial space advantage over object logging based fault tolerance mechanisms. However, this method result in false-positive object membership errors owing to its probabilistic nature of detecting the object membership. To avoid false-positive object membership matches of the bloom filter, we have proposed a DLFT framework.

DLFT framework address the false positive object membership matches of the DAFT mechanism by complementing DAFT with object layout information. Upon successful object transfer, DLFT framework logs the object layout information along with its corresponding k hash positions of the bloom filter as described above. If there is any fault during the transfer, DLFT searches for the completed object information by comparing k hash positions as well as the block layout bit in the bloom filter are set or not. If not set, then schedules the object for the transfer. Our experimental results conclude that with small to negligible overhead, the use of the object layout information in conjunction with DAFT can help in avoiding false-positive matches of the bloom filter.

V. EVALUATION

In this section, we describe our simulation environment and present the experimental results along with their analysis.

A. TESTBED AND WORKLOAD SPECIFICATIONS

1) TESTBED

For our experiments, we used a private testbed with two nodes (source and sink) connected using the InfiniBand (IB) network interface. We used Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.10GHz servers with 32 cores and 16 GB DRAM. The source and sink hosts operate on Linux kernel 3.10.0-1062. Additionally, both nodes have separate Lustre file systems 2.9.0 [28] with one OSS and 4 OSTs, mounted over 1 TB drives each. By default, our Lustre file system configuration includes a stripe count of four with a stripe size of 1 MB. To fairly evaluate our implementation, we ensured that the storage server bandwidth is not over-provisioned with respect to the network bandwidth between the source and sink servers (i.e., the network does not encounter a bottleneck).

2) WORKLOAD

To analyze the file size distribution, we used the distribution data of Lustre Atlas 1 & 2 filesystems [29] hosted by the Oak Ridge Leadership Computing Facility [3]. Fig. 6 plots the number of files vs. the file size. It can be observed from this plot that 91.55% of the files are less than 4 MB and 84.17% are less than 1 MB. Additionally, less than 10% of the files are greater than 4 MB, and these files occupy most of the file system space. Hence, for the purpose of our evaluation, we used two groups of files with different sizes; one for small workloads with 10,000 files of size 1 MB and the other for big workloads with 100 files of size 1 GB. For the purpose of evaluation, we used a pre-configured source file system, with big and small workloads, by stripping the data across 4 OSTs using stripe count as 4 and stripe size as 1MB.

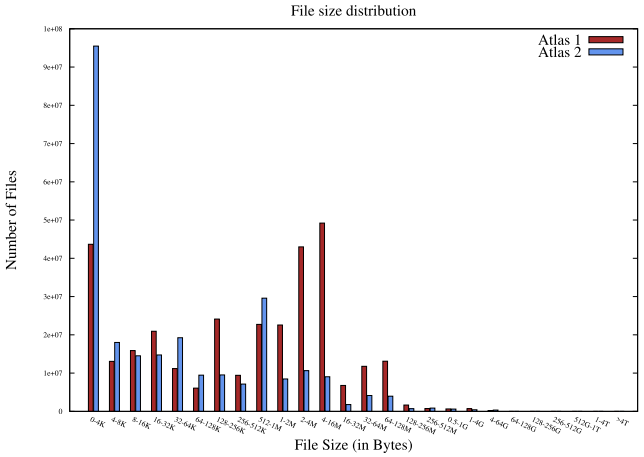


FIGURE 6. File size distribution.

3) THREAD CONFIGURATION

For an optimal evaluation environment, in all our experiments, we configured the data transfer framework with four I/O threads, four B/F threads (only source), one master thread, and one comm thread at both the source and sink endpoints.

4) RECOVERY TIME

Because there is no direct method for evaluating the recovery time of failed transfers, we estimated the recovery time of the DAFT and DLFT methods as follows.

$$ER_t = TBF_t + TAF_t - TT_t \tag{6}$$

where,

ER_t = Estimated Recovery Time
TBF_t = Time consumed before fault
TAF_t = Time consumed after fault
TT_t = Time consumed with no fault

5) BLOOM FILTER CONFIGURATION

As noted above, for all our experiments, we populated our file system with files with a stripe size of 1MB and used two groups of files with different sizes; one for small workloads (10,000 files of size 1 MB) and the other for big workloads (100 files of size 1 GB). Hence, the number of elements, n, to be inserted in the bloom filter is 102400 and 10000 for the big and small workloads, respectively. The required number of bits in the bloom filter, m, for a given n and the desired false positive probability, ϵ (Eq. (3)), can be computed by substituting the optimal value of k in the probability expressions, Eq.(7) and Eq.(8).

$$k = \frac{m}{n} ln2 \tag{7}$$

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

The salient feature of the bloom filter is that there is a clear tradeoff between m and the probability of a false positive. Inorder to balance the computational and storage overhead, we considered the optimal value of k as 7 and the false positive probability as *0.00001* and *0.0001*, respectively, for

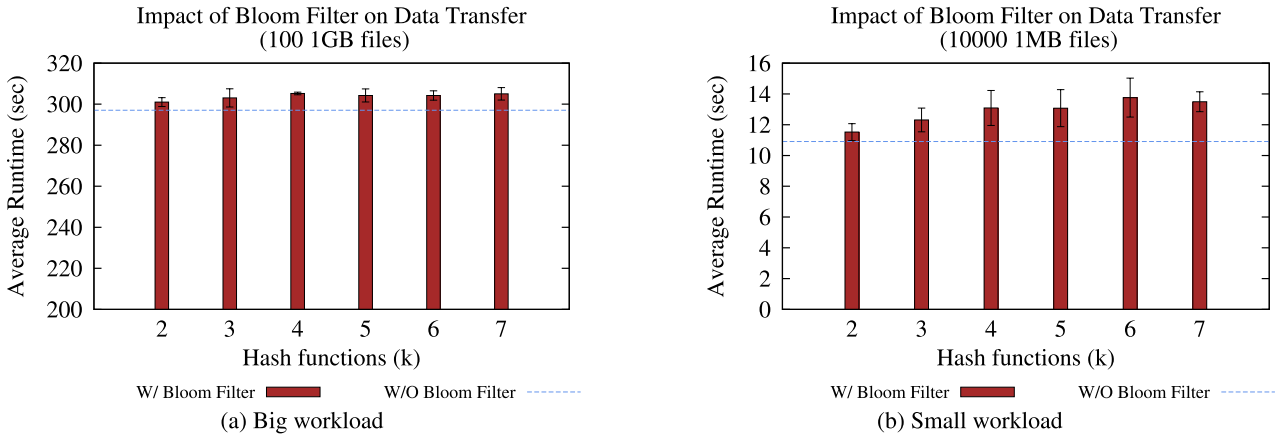


FIGURE 7. Data transfer time analysis for big and small workloads. The 99% confidence intervals are shown in the error bar.

big and small workloads. Based on the above considerations, we computed the number of bits in the bloom filter. As a result of these calculations, in all our experiments, the value of m was considered to be 28 *KB* for small workloads and 408 *KB* for big workloads.

B. PERFORMANCE EVALUATION

One of the major objectives while designing the proposed fault tolerance mechanisms involved the minimization of the bloom filter-based fault tolerance overhead on the overall data transfer time. In this section, we present the evaluation results for the proposed bloom filter-based DAFT on the overall data transfer performance. To evaluate the performance, we used the data transfer time, computational overhead, recovery overhead, and false-positive matches as the target performance factors.

1) DATA TRANSFER TIME

In this section, we analyze the impact of DAFT on the overall data transfer time. Fig. 7 depicts the DAFT data transfer time comparison for both big and small workloads. In these figures, the DAFT transfer time for varying number of hash functions are represented using a bar graph, whereas the line represents the transfer time without fault tolerance support.

From Fig. 7(a) and Fig. 7(b), it is evident that the DAFT method has a minimum to negligible impact on the overall data transfer time. Although it is expensive to compute the hash functions and generate the bloom filter, the overhead is nullified using a pipelining architecture, as described in IV-B. We utilized this pipelining technique to achieve high-performance data transfer with bloom filter-based DAFT. Our pipeline included the read, hash, bloom filter generation, and transfer operations. Our system design ensured that each operation overlaps with the operations of a different block.

Along with the average runtime, Fig. 7(a) and Fig. 7(b) illustrate the 99% confidence intervals represented as error bars. From these error bars, we can observe some variability for both workloads. This variability might be related to the file management overhead of the file system. Overall, from Fig. 7, we can conclude that the bloom filter-based DAFT method has a negligible impact on the overall data transfer time.

2) RECOVERY TIME

Minimizing the recovery time upon resuming from fault is another major objective of our fault tolerance framework. In this section, we evaluated the recovery time for both small and big workloads. For the effective evaluation of the recovery time, we created a simulation environment wherein we generated faults after transferring 20%, 40%, 60%, and 80% of the total data size. Although faults can occur at any transfer endpoint, in our simulation environment, we generated faults at the source end.

Fig. 8 and Fig. 9 depict the recovery time of both big and small workloads at varying fault points, respectively. In these figures, recovery time for varying number of hash functions are represented using a bar graph, whereas the line represents baseline data transfer time without failure and fault tolerance support. For effective comparision, we have represented time consumed before fault, after fault, and estimated recovery time.

From these graphs, it is evident that the recovery time is linearly proportional to the fault point. The later the fault, the higher the number of redundant blocks to be hashed for verifying the object membership to determine if the block has already been transferred. Hence, the later the fault, the higher the recovery time. These graphs also elucidate that the recovery time is independent of the number of hash functions being used for generating the bloom filter. For quantitative comparison, the percentage of recovery time against the number of hash functions was calculated. At 20% fault time, our fault tolerance framework experienced an overhead of approximately 2–3% for all hash functions; however, at 80% fault point, the overhead increased to 10–11%.

Based on our evaluation results, we can conclude that the recovery time does not affect the number of hash functions used for generating the bloom filter.

3) FALSE POSITIVE MATCHES OF DAFT

The bloom filter-based fault tolerance framework is prone to false-positive results of object membership queries.

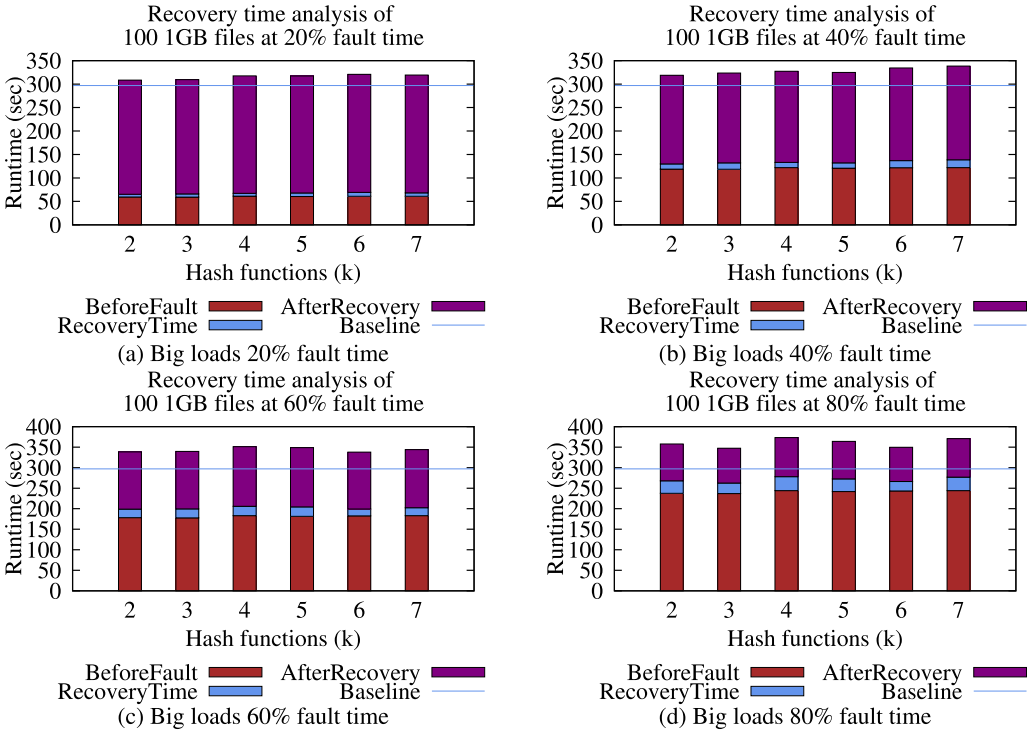


FIGURE 8. Recovery time analysis of DAFT at varying fault timing for big workloads.

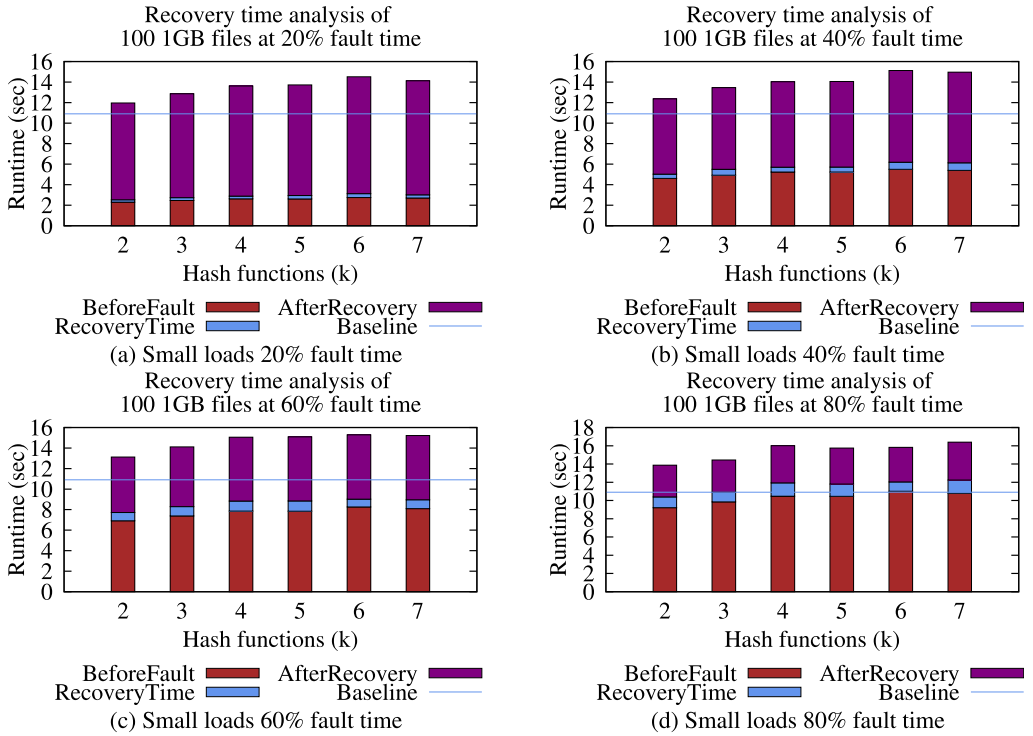
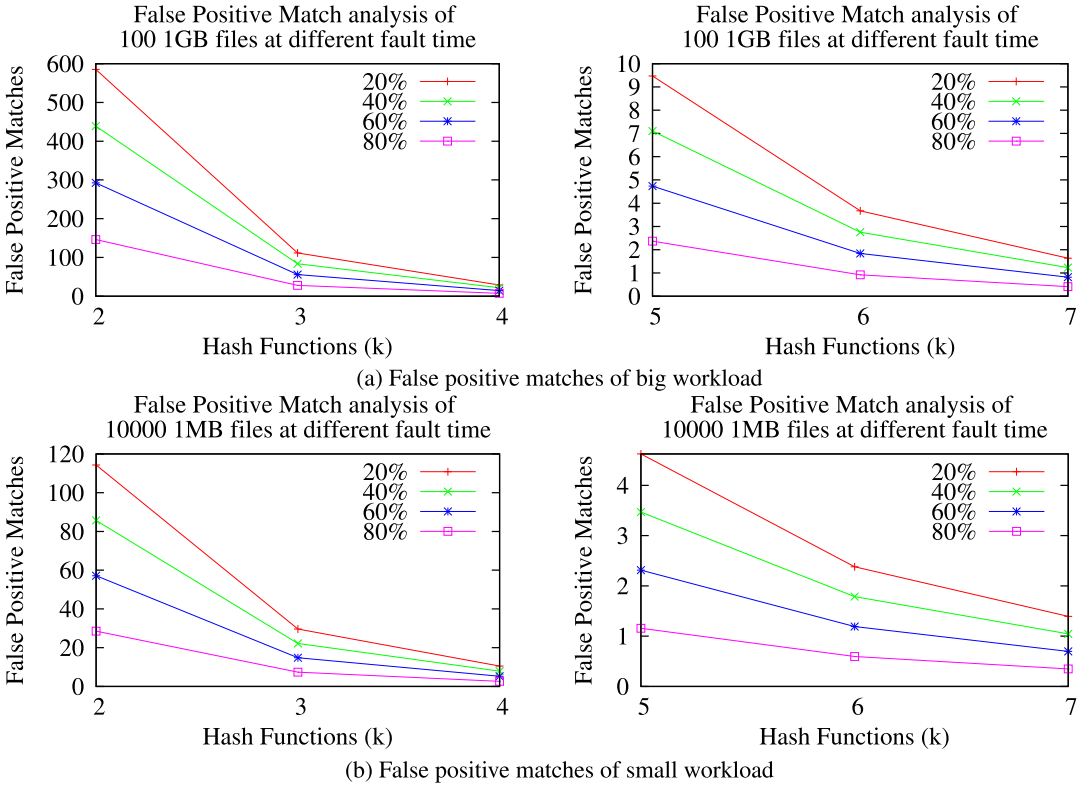


FIGURE 9. Recovery time analysis of DAFT at varying fault timing for small workloads.

Thus, some object membership queries result in errors, and some blocks that are not transferred to the sink end are falsely considered to be transferred. This results in data corruption; therefore, the transferred data would be unusable for further analysis. Reducing the false positive matches of the object membership queries is another major design aspect of this framework.

Fig. 10 shows the false positive matches for the big and small workloads at varying fault points and for different number of hash functions. For ease of analysis, we divided



(a) False positive matches of big workload

(b) False positive matches of small workload

FIGURE 10. False positive matches of DAFT at varying fault timing for big and small workloads.

the graph into two sets (k = {2,3,4} and k = {5,6,7}). From this graph, we can observe that the false-positive matches are inversely proportional to the number of hash functions. Additionally, it is evident that the later the fault point, the lower the number of false-positive matches. This is because the later the fault point, the lower the number of blocks to be transferred to the sink end.

We evaluated our framework using the bloom filter configuration described in SectionV-A5. From Fig. 10, it can be observed that the false-positive match patterns for both big and small workloads are significantly similar. It can be observed from the graphs that at 20% fault and $k = 2$, big workloads experience approximately 0.7% false-positive matches, whereas at a similar configuration, small workloads experience approximately 1.5% false-positive matches. As we increase the number of hash functions to 7, the number of false-positive matches reduces drastically. From the evaluation results, it can be observed that at 20% fault and $k = 7$, both big and small workloads experience as low as 2 false-positive matches. From this, we can conclude that increasing the number of hash functions will minimize the number of false-positive matches. However, these false-positive matches cannot be completely avoided owing to the probabilistic nature of the bloom filter. To negate the impact of the false-positive matches of DAFT, we proposed the DLFT framework, as described in Section IV-C.

4) RECOVERY TIME AND FALSE POSITIVE MATCHES OF DLFT

To evaluate the impact of the DLFT mechanism on the recovery time, we used a similar simulation environment as described in V-B2, and generated faults at 20%, 40%, 60%, and 80% of the total data size. Fig. 11 and Fig. 12 depict the DLFT recovery time for both big and small workloads, respectively. For effective comparison, the total recovery time is represented by two colors. $RecoveryTime_A$ denotes the DAFT recovery time and $RecoveryTime_L$ represents the object layout logging overhead. Also, to understand the impact of failures, the baseline data transfer time without failure is represented as a line, *Baseline*.

Big Workloads: From Fig. 11, it can be observed that at 20% and 40% fault points for a varying number of hash functions (k), the overhead incurred by object layout logging is significant. However, at 60% and 80% fault points, the layout logging overhead is considerably negligible. This is because the later the fault, the higher the DAFT recovery time. Although the object layout logging overhead is independent of the fault points, a higher DAFT recovery time (at 60% and 80% fault points) results in negligible object layout logging overhead.

Small Workloads: From Fig. 12, we can observe that small workloads also exhibit similar behavior as big workloads for a varying number of hash functions (k). At 20% and 40% fault points, the object layout logging overhead is visible; however, this overhead is not significant at 60% and 80% fault points.

Accordingly, we can conclude that with small to negligible overhead, the use of the object layout logging mechanism in conjunction with DAFT can help in avoiding false-positive matches of the bloom filter, irrespective of the number of hash functions used. According to the experimental results,

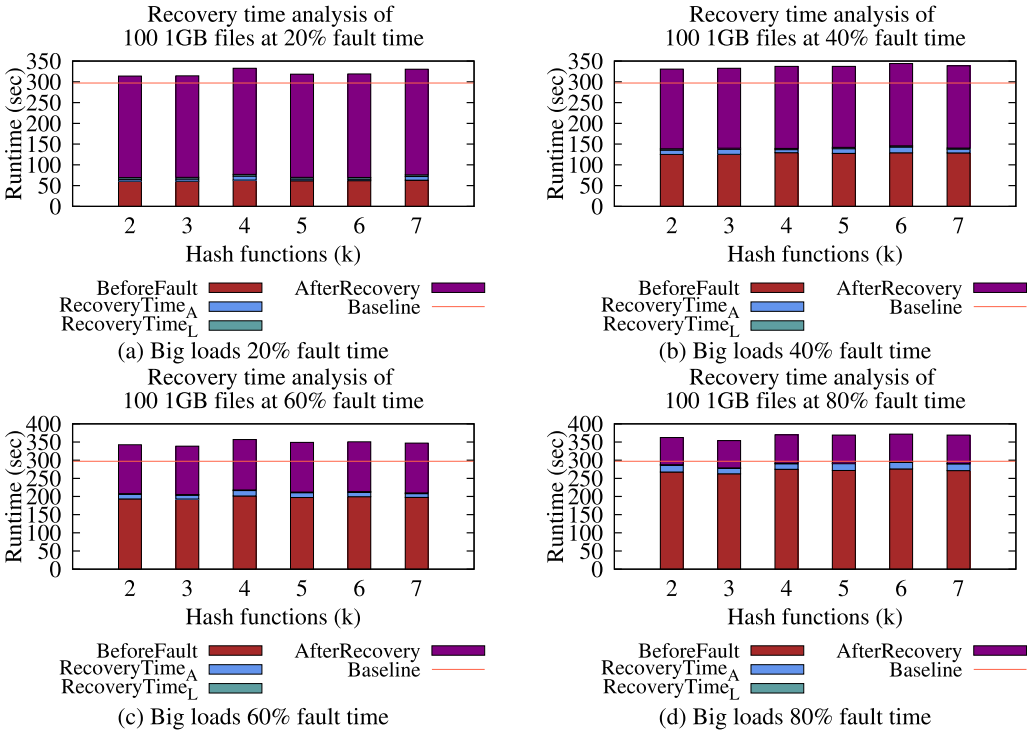


FIGURE 11. Recovery time analysis of DLFT at varying fault timing for big workloads.

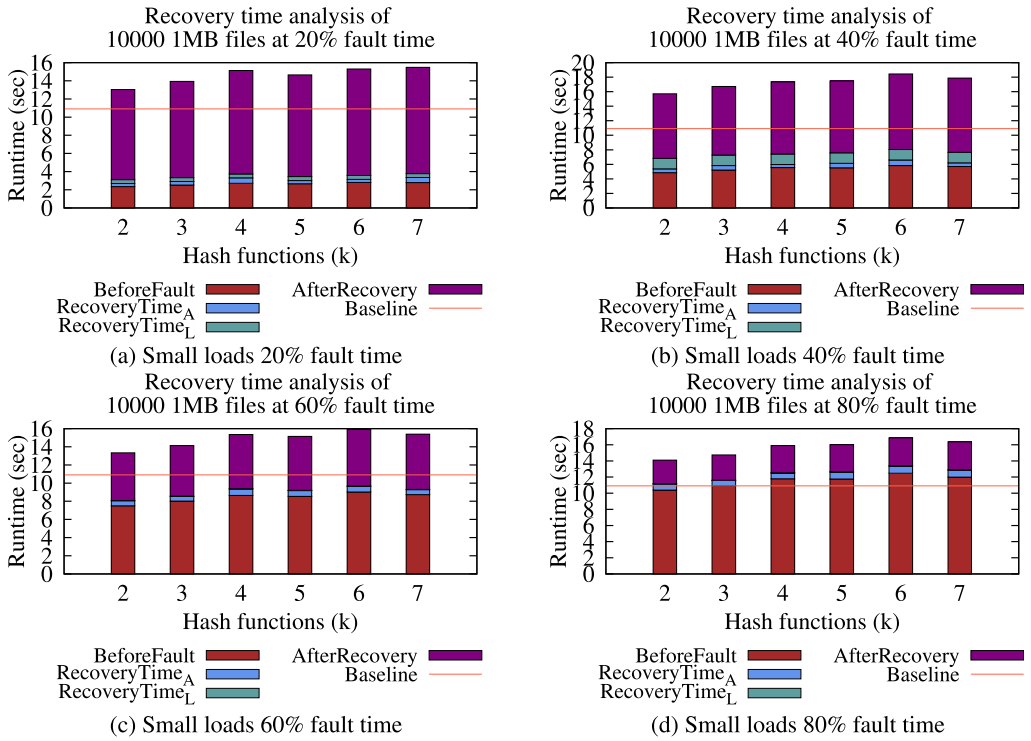
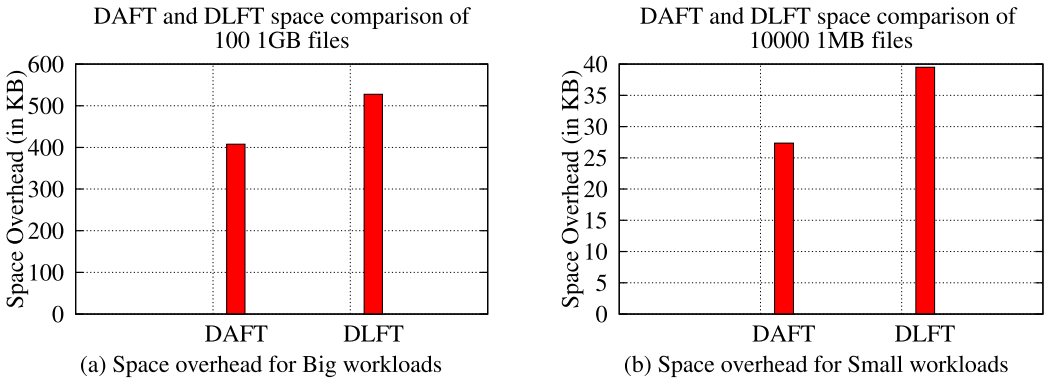


FIGURE 12. Recovery time analysis of DLFT at varying fault timing for small workloads.

no false-positive matches were observed with the DLFT mechanism.

5) SPACE OVERHEAD ANALYSIS

An important aspect of using the bloom filter-based probabilistic fault tolerance mechanism is to reduce the amount of space occupied by the fault tolerance framework during data transfer. SectionV-A5 describes the required storage space for computing the bloom filter at varying number of hash functions. Fig. 13 depicts the space overhead of DAFT and DLFT for both big and small workloads. From Fig. 13(a) and Fig. 13(b), we can observe that the amount of space



(a) Space overhead for Big workloads

(b) Space overhead for Small workloads

FIGURE 13. Space overhead analysis of DAFT and DLFT for both big and small workloads.

occupied by the DAFT and DLFT frameworks is negligible (in the order of few KB (KiloBytes)) for both big and small workloads. Owing to the additional object layout information, the DLFT space overhead was found to be 30% and 44% higher than DAFT for big and small workloads, respectively.

VI. CONCLUSION

The PFS distributes file data across multiple storage servers and provides concurrent access to the file data. Big data transfer tools with higher data transfer performance requirements employ multiple I/O threads to process data transfers. Thus, objects of different logical files might be transferred in parallel. Although such data transfer tools significantly improve the data transfer performance, they introduce additional complexity in the efficient management of faults because traditional file offset-based fault tolerance mechanisms are not suitable owing to the out-of-order nature of data transfer. In this study, we proposed DAFT mechanisms that employed a bloom filter-based probabilistic data structure for efficiently managing faults with out-of-order object transmission. We evaluated and compared the data transfer performance overhead of DAFT for varying number of hash functions (k = 2 to 7) on the overall data transfer rate and concluded that the proposed fault tolerance mechanisms do not negatively impact the data transfer performance. Moreover, to evaluate the recovery time overhead of DAFT, we created a simulation environment to generate faults at 20%, 40%, 60%, and 80% points of data transfer. The evaluation results demonstrated that the recovery time of DAFT increases with the fault point of data transfer. For big workloads, we observed a minimum overhead of 2% for $k = 2$ and 2.5% for $k = 7$ at 20% fault point. Meanwhile, at 80% fault point, the recovery time overhead increased to 10% for $k = 2$ and 10.7% for $k = 7$. Similar recovery time overhead was observed with small workloads as well.

Although the proposed DAFT mechanism significantly improved the data transfer tool performance upon encountering faults, its probabilistic nature of data structure resulted in false-positive object matches. Thus, some objects that were supposed to be transferred after fault were considered as already transferred. This results in data integrity issues. We evaluated the false-positive object matches for both big and small workloads at predefined fault points. The experimental results demonstrated that at 20% fault point and number of hash functions $k = 2$, 0.7% and 1.5% false-positive matches were observed for big and small workloads, respectively. As the number of hash functions was increased to $k = 7$, the number of false object matches was reduced to as low as 2 for both big and small workloads.

By increasing the number of hash functions, the number of false-positive matches can be reduced, but they cannot be completely avoided. Therefore, we proposed the DLFT framework that combines DAFT with the object layout information to avoid false-positive object matches. According to the experimental results, no false-positive matches were observed with the DLFT mechanism. We also evaluated the recovery time and space overhead of the DLFT framework on the overall data transfer performance. The experimental results demonstrated constant and negligible recovery time overhead at all fault points for both big and small workloads. Furthermore, 30% and 44% space overheads were observed in comparison to DAFT for big and small workloads, respectively.

To summarize, the proposed DAFT and DLFT mechanisms complement the existing big data transfer tools with fault tolerance support without negatively impacting the data transfer performance.

REFERENCES

[1] *CERN*. Accessed: Apr. 4, 2020. [Online]. Available: https://home.cern/

[2] *LIGO*. Accessed: Apr. 4, 2020. [Online]. Available: https://www.ligo.caltech.edu/

[3] *ORNL*. Accessed: Apr. 4, 2020. [Online]. Available: https://www.ornl.gov/

[4] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou, "Scalable performance of the panasas parallel file system," in *Proc. 6th USENIX Conf. File Storage Technol.*, 2008, pp. 1–17. [Online]. Available: http://dl.acm.org/citation.cfm?id=1364813.1364815

[5] Y. Kim, S. Atchley, G. R. Vallée, and G. M. Shipman, "LADS: Optimizing data transfers using layout-aware data scheduling," Oak Ridge Nat. Lab., Oak Ridge, TN, USA, Tech. Rep. ORNL/TM-2014/251, Jan. 2015.

[6] Y. Kim, S. Atchley, G. R. Vallee, S. Lee, and G. M. Shipman, "Optimizing end-to-end big data transfers over terabits network infrastructure," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 1, pp. 188–201, Jan. 2017.

[7] *Lustre Wiki*. [Online]. Available: https://en.wikipedia.org/wiki/Lustre_(file_system)

[8] R. Koo and S. Toueg, "Checkpointing and rollback-recovery for distributed systems," *IEEE Trans. Softw. Eng.*, vol. SE-13, no. 1, pp. 23–31, Jan. 1987.

[9] S. Park, Y. Kim, and S. R. Maeng, "Lightweight logging and recovery for distributed shared memory over virtual interface architecture," in *Proc. 2nd Int. Symp. Parallel Distrib. Comput.*, Nov. 2003, p. 54.
[10] M. Amoon, "Adaptive framework for reliable cloud computing environment," *IEEE Access*, vol. 4, pp. 9469–9478, 2016.
[11] J. Zhao, Y. Xiang, T. Lan, H. H. Huang, and S. Subramaniam, "Elastic reliability optimization through peer-to-peer checkpointing in cloud computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 2, pp. 491–502, Feb. 2017.
[12] G. Yao, Y. Ding, and K. Hao, "Using imbalance characteristic for fault-tolerant workflow scheduling in cloud systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 12, pp. 3671–3683, Dec. 2017.
[13] W. Allcock, J. Bresnahan, R. Kettimuthu, and M. Link, "The globus striped GridFTP framework and server," in *Proc. ACM/IEEE SC Conf. (SC)*, Nov. 2005, pp. 54–64, doi: 10.1109/SC.2005.72.
[14] G. Alliance. *The Globus Toolkit*. Accessed: Oct. 10, 2020. [Online]. Available: http://http://www.globus.org/toolkit/
[15] A. Hanushevsky. *BBCP*. [Online]. Available: http://www.slac.stanford.edu/~abh/bbcp/
[16] B. W. Settlemyer, J. D. Dobson, S. W. Hodson, J. A. Kuehn, S. W. Poole, and T. M. Ruwart, "A technique for moving large data sets over high-performance long distance networks," in *Proc. IEEE 27th Symp. Mass Storage Syst. Technol. (MSST)*, May 2011, pp. 1–6.
[17] T. Li, Y. Ren, D. Yu, and S. Jin, "RAMSYS: Resource-aware asynchronous data transfer with multicore SYStems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 5, pp. 1430–1444, May 2017, doi: 10.1109/TPDS.2016.2619344.
[18] Y. Kim, S. Atchley, G. Schmuck, Vallée, and M. G. Shipman, "LADS: Optimizing data transfers using layout-aware data scheduling," in *Proc. 1st USENIX Conf. File Storage Technol.*, Berkeley, CA, USA: USENIX Association, 2015, pp. 67–80.
[19] P. Kasu, T. Kim, J.-H. Um, K. Park, S. Atchley, and Y. Kim, "FTLADS: Object-logging based fault-tolerant big data transfer system using layout aware data scheduling," *IEEE Access*, vol. 7, pp. 37448–37462, 2019.
[20] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970.
[21] A. Broder and M. Mitzenmacher, "Survey: Network applications of Bloom filters: A survey," *Internet Math.*, vol. 1, no. 4, pp. 485–509, Jan. 2004.
[22] *Bloom Filter*. [Online]. Available: https://en.wikipedia.org/wiki/Bloom_filter
[23] A. Kirsch and M. Mitzenmacher, *Less hashing, same performance: Building a Better Bloom Filter*, vol. 4168. Berlin, Germany: Springer, Jan. 2006, pp. 456–467, doi: 10.1007/11841036_42.
[24] J. Lu, T. Yang, Y. Wang, H. Dai, L. Jin, H. Song, and B. Liu, "One-hashing Bloom filter," in *Proc. IEEE 23rd Int. Symp. Qual. Service (IWQoS)*, Jun. 2015, pp. 289–298.
[25] L. Luo, D. Guo, R. T. B. Ma, O. Rottenstreich, and X. Luo, "Optimizing Bloom filter: Challenges, solutions, and comparisons," 2018, *arXiv:1804.04777*. [Online]. Available: http://arxiv.org/abs/1804.04777
[26] *Murmur*. Accessed: Aug. 29, 2020. [Online]. Available: https://en.wikipedia.org/wiki/MurmurHash
[27] *LUSTRE*. Accessed: Sep. 17, 2020. [Online]. Available: http://doc.lustre.org/lustre_manual.xhtml
[28] F. Wang, S. Oral, G. M. Shipman, O. Drokin, T. Wang, and I. Huang, "Understanding Lustre filesystem internals," Oak Ridge Nat. Lab., Oak Ridge, TN, USA, Tech. Rep. ORNL/TM-2009/117, 2009.
[29] *Atlas*. Accessed: Oct. 7, 2020. [Online]. Available: https://github.com/ORNL-TechInt/Atlas_File_Size_Data

PREETHIKA KASU is currently pursuing the Ph.D. degree with the Department of Software and Computer Engineering, Ajou University, Suwon, South Korea. Her doctoral research interests include fault tolerance, distributed file and storage, parallel I/O, and high-performance computing.

PRINCE HAMANDAWANA received the B.Sc. degree (Hons.) in computer science from the National University of Science and Technology (NUST), Bulawayo, Zimbabwe, in 2010. He is currently pursuing the Ph.D. degree in computer engineering with the Database and Dependable Computing (DBDC) Laboratory, Ajou University, Suwon, South Korea. He worked as a Network Engineer in some of the Zimbabwean leading service providers, Econet Wireless, from 2008 to 2011, and Liquid Telecom, from 2011 to 2016. His research interests include distributed and parallel storage systems, and GPU assisted cluster-wide data deduplication. He is also a member of the Database and Dependable Computing (DBDC) Laboratory, Ajou University.

TAE-SUN CHUNG received the B.S. degree from KAIST, Daejeon, South Korea, in 1995, and the M.S. and Ph.D. degrees in computer science from Seoul National University, Seoul, South Korea, in 1997 and 2002, respectively. He is currently working as a Professor with the Department of Artificial Intelligence, Ajou University, Suwon, South Korea. His research interests include flash memory storages, XML databases, and database systems.